Essential ActionScript 3.0

Colin Moock

O'REILLY®

Beijing · Cambridge · Farnham · Koln · Paris · Sebastopol · Taipei · Tokyo

ActionScript 3.0 для Flash

Подробное руководство

Колин Мук

O'REILLY®

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2009

ББК 32.988.02-018
УДК 004.738.5
М90

Мук К.
М90 ActionScript 3.0 для Flash. Подробное руководство. — СПб.: Питер, 2009. — 992 с.: ил.

ISBN 978-5-91180-808-2

«Это новая книга Колина Мука». Данной фразы будет достаточно, чтобы в книжные магазины выстроились очереди из флэшеров. Не было и нет автора, который бы по авторитету и влиянию мог равняться Муку. Он был первым, кто написал достойную книгу по ActionScript сразу же после появления этого языка в далеком 2001 году. ActionScript стремительно развивался, превращаясь в полноценный объектно-ориентированный язык. И неизменную помощь в освоении очередных революционных нововведений начинающим и опытным флэшерам по всему миру оказывали книги Колина Мука.

Казалось бы, еще совсем недавно всех поразил ActionScript 2.0, в котором появились все особенности объектно-ориентированного языка. Однако по своей сути это оставался старый добрый ActionScript 1.0, что проявлялось в формальности многих возможностей. Эти недостатки исчезли в ActionScript 3.0, который стал мощнее, удобнее, строже, быстрее. Данная книга, будучи лучшим руководством по ActionScript 3.0, объединяет в себе достоинства своих предшественниц — «Essentials ActionScript 2.0» и «ActionScript for Flash MX: The Definition Guide». В ней рассматриваются как фундаментальные основы языка и ключевые идеи объектно-ориентированного программирования, так и конкретные возможности по управлению содержимым Flash-приложений. Уникальный авторский стиль, множество реальных примеров, грамотный перевод — все это позволит освоить ActionScript 3.0 быстро и легко.

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-0-596-52694-8 (англ.)
ISBN 978-5-91180-808-2

Краткое содержание

Часть I. ActionScript с нуля

Часть II. Отображение и интерактивность

Часть III. Практическое применение ActionScript

Оглавление

Часть I. ActionScript с нуля

Часть II. Отображение и интерактивность

Часть III. Практическое применение ActionScript

Предисловие

Мы мечтаем о мире, в котором любое цифровое взаимодействие — в классе, офисе, квартире, аэропорту или машине — превращается в мощную, простую, эффективную и занимательную операцию. Для реализации подобных возможностей широко применяется приложение Flash Player, которое превратилось в сложную платформу, поддерживаемую различными браузерами, операционными системами и устройствами.

Один из основных стимулов для появления новаторских решений корпорации Adobe и разработки приложения Flash Player состоит в том, что разработчики постоянно расширяют границы реализуемых возможностей — эти возможности впоследствии становятся достоянием других разработчиков.

Если бы с помощью машины времени мы могли вернуться в 2001 год, мы бы обратили внимание на широкое распространение Интернета и первые свойства сайтов, содержащих не только страницы, но и интерактивные приложения. Эти приложения в основном использовали HTML-формы и полагались на веб-серверы, обрабатывавшие полученную информацию. Группа специалистов работала над реализацией более быстрого механизма взаимодействия, используя преимущества обработки данных на стороне клиента с помощью языка ActionScript технологии Flash. Одним из самых ранних примеров успешных интерактивных приложений была система бронирования номеров для отеля Broadmoor Hotel — вместо многостраничных HTML-форм в ней использовался одноэкранный, полностью интерактивный интерфейс, позволивший увеличить объем бронирования через Интернет на 89 %.

Очевидно, что скорость имеет значение. Она обеспечивает гораздо более эффективное и впечатляющее взаимодействие. Однако в 2001 году существовало множество проблем с производительностью, возможностями языка сценариев, простотой отладки и ограничениями дизайна в браузерах (которые создавались для просмотра страниц, а не для выполнения других приложений).

В своей компании мы провели множество «мозговых атак» и активно беседовали с разработчиками, и в итоге было решено приступить к реализации данного направления, получившего название «Многофункциональные интернет-приложения» (RIA — Rich Internet Applications). Для лучшей поддержки RIA-приложений мы должны были создать следующее.

- Гораздо более быструю виртуальную машину в приложении Flash Player для языка ActionScript 3.0.
- Платформу разработки под названием Flex, значительно упрощающую разработку RIA-приложений.
- Среду, которая позволяла бы раскрыть все возможности многофункциональных интернет-приложений, — теперь она называется Adobe Integrated Runtime (AIR). В эпоху активного развития интернет-проектов мы зацепились за идею реализации этого будущего мира многофункциональных веб-приложений.

Мы продолжали инвестировать средства в создание целого ряда технологий и готовились к моменту, когда эта инновация станет востребованной. Затраты на иннова-

цию восполнились в полном объеме, и я счастлив наблюдать за появлением многофункциональных интернет-приложений вместе с технологией Web 2.0. Применяя множество технологий и платформ, разработчики создают приложения, которые предоставляют доступ к потенциалу Интернета, позволяют использовать преимущества HTML, Flash, Flex и AJAX и размещать одну часть логики приложения на клиенте, а другую — на сервере.

Вместе с приложением Flash Player 9 была разработана новая виртуальная машина, позволяющая значительно увеличить скорость выполнения приложений на языке ActionScript 3.0 и реализующая самые последние требования стандарта ECMA для языка (JavaScript основан на этом же стандарте). По соглашению с сообществом Mozilla Foundation был открыт исходный код этой современной реализации (проект Tamarin), что позволило создателям приложения Flash Player сотрудничать с инженерами Mozilla с целью оптимизации виртуальной машины и реализации самых последних стандартов. Этот базовый движок выполнения сценариев со временем будет встроен в браузер Firefox, обеспечивая совместимость сценариев в HTML и Flash.

Кроме того, появилась платформа Flex, которая обеспечивает быструю разработку приложений с помощью распространенных шаблонов для взаимодействия и управления данными, а вся платформа встроена в язык ActionScript 3.0. Платформа разработки Flex доступна бесплатно и содержит исходный код, благодаря чему вы можете точно узнать, как она работает. Для написания кода, использующего возможности платформы Flex, можно применять любой редактор. Существует также специальная среда IDE под названием Adobe Flex Builder.

Наблюдая за появлением инноваций в Интернете, мы решили объединить усилия корпорации Adobe и компании Macromedia. В то время как компания Macromedia занималась развитием RIA-приложений на основе технологии Flash, корпорация Adobe вносила изменения в алгоритм передачи электронных документов и в другие области. Со временем мы увидели, что компания Macromedia собирается добавить поддержку электронных документов в RIA-приложения, а корпорация Adobe — наоборот, ввести возможности RIA-приложений в электронные документы. Чтобы не идти разными путями к одной цели и не изобретать заново велосипед, мы объединили усилия для реализации нашего видения следующего поколения документов и RIA-приложений, совместив лучшую мировую технологию для электронных документов и самую распространенную технологию для RIA-приложений. Это невероятно мощная комбинация.

После объявления об объединении наших компаний мы создали команду «чистой комнаты», чтобы разработать планы относительно нашего следующего поколения программного обеспечения, заимствуя все, что узнали до этого момента, а также потенциал объединения технологий Flash, PDF и HTML в новой среде Adobe AIR для RIA-приложений.

Проект AIR на самом деле является нашей третьей попыткой создать эту новую среду. Первые две попытки были частью экспериментального проекта Central под кодовыми названиями Mercury и Gemini (космическая программа Соединенных Штатов) и проекта AIR под кодовым названием Apollo. Мы узнали многое из этих проектов, и, о чем я люблю напоминать команде, экипаж Apollo первым высадился на Луну.

Среда AIR позволит вам использовать существующие навыки веб-разработки (Flash, Flex, HTML, JavaScript, AJAX) для создания и распространения RIA-приложений в виде настольных приложений. Как и в случае с публикацией интернет-приложений, позволяющей любому разработчику с базовыми знаниями языка HTML создать сайт, среда AIR позволит любому разработчику с базовыми навыками веб-разработки создать настольное приложение.

Как разработчик, вы можете теперь входить в более близкий контакт с вашими пользователями. Использование браузера подразумевает быструю, отчасти непрочную связь с ними. Они открывают страницу и затем уходят. Среда AIR позволяет вам создавать приложения, которые могут поддерживать связь с пользователями длительное время. Как и любое настольное приложение, приложения AIR имеют значок на **Рабочем столе**, в меню **Пуск** операционной системы Windows или на панели инструментов операционной системы Mac OS X. Кроме того, когда вы выполняете веб-приложение, оно существует в отдельном от вашего компьютера мире. Вы не можете просто интегрировать локальные данные в свое приложение. Например, вы не можете просто перетащить контакты из программы Outlook в приложение с картой, чтобы получить маршрут к дому вашего друга. Хотя это можно сделать с помощью приложений AIR, которые создают мост между вашим компьютером и Интернетом.

Я верю, что среда AIR — это начало нового измерения. Эти приложения интересно создавать. Начав раньше, вы сможете реализовать в своих приложениях такие возможности, которых еще нет у других, особенно это касается улучшения представления вашего приложения на компьютере и установления связи между Интернетом и настольным приложением.

Основой RIA-приложений является язык ActionScript независимо от того, выполняются они во Flash Player в браузере, в виде настольного приложения в среде AIR или на мобильных устройствах. Каждое поколение языка ActionScript подробно описано Колином Муком (Colin Moock) в серии книг издательства O'Reilly — они стали справочниками, которыми пользуется большинство разработчиков на платформе Flash. С языком ActionScript 3.0 вы получаете беспрецедентную мощь для построения привлекательных приложений, а в этом справочнике вы найдете подробное описание эффективного использования этой мощи.

Я хочу увидеть созданные вами шедевры и с нетерпением ожидаю появления приложений следующего поколения. Продолжайте расширять границы возможного в Интернете, чтобы сделать приложения более привлекательными и эффективными для людей во всем мире, а мы будем делать все возможное, чтобы поставлять более выразительные и мощные инструменты, помогая вам в ваших усилиях.

Кевин Линч (Kevin Lynch),
ведущий разработчик архитектуры ПО, корпорация Adobe

Сан-Франциско, 2007

Введение

ActionScript является официальным языком платформы Adobe Flash. Изначально ActionScript задумывался как простой инструмент для управления анимацией, но со временем он превратился в полноценный язык программирования, который сегодня используется для создания разнообразного содержимого и приложений для Интернета, мобильных устройств и настольных компьютеров. Основополагающие принципы, заложенные в язык ActionScript, делают его удобным средством решения разнообразных задач для программистов, работающих в разных сферах, и разработчиков содержимого. Например, аниматору нужно написать всего несколько строк кода на языке ActionScript, чтобы остановить воспроизведение анимации. Дизайнеру интерфейса потребуется несколько сот строк кода на языке ActionScript, чтобы добавить интерактивность в интерфейс мобильного телефона. А с помощью нескольких тысяч строк кода ActionScript разработчик приложений может создать полноценную программу для чтения электронной почты, которая будет работать в веб-браузере или автономно.

В этой книге представлена исчерпывающая информация по основам программирования на языке ActionScript, а стиль изложения материала отличается чрезвычайной доходчивостью и точностью. Такая непревзойденная точность и глубина материала являются результатом десятилетнего изучения языка ActionScript, использования практического опыта в области программирования и постоянного контакта с инженерами компании Adobe. Каждое слово этой книги было тщательно проверено — в большинстве случаев по нескольку раз — ключевыми фигурами инженерно-технического персонала компании Adobe, включая команды, работающие над созданием программ Flash Player, Flex Builder и среды разработки Flash. Дополнительную информацию можно найти в разд. «Благодарности».

Для новичков

Если у вас нет опыта программирования, начните чтение книги с главы 1. В ней вы познакомитесь с фундаментальными понятиями языка ActionScript: *переменная*, *метод*, *класс* и *объект*. После этого можете приступать к последовательному изучению остальных глав. Каждая глава базируется на концепциях предшествующих глав, представляя новые темы в виде одного длительного повествования, которое поможет вам овладеть мастерством разработки с использованием языка ActionScript.

Если же вы являетесь дизайнером, который просто хочет узнать, как управлять анимацией в среде разработки Flash, то вполне возможно, что эта книга не для вас. В данном случае ответы на все вопросы вы сможете найти в документации от компании Adobe. Эта книга принесет пользу, когда вы захотите узнать, как добавить в ваше содержимое логику и программируемое поведение.

Профессиональное руководство

Если у вас уже есть опыт программирования на языке ActionScript, эта книга поможет устранить пробелы в ваших знаниях, переосмыслить важные понятия, описанные формальными терминами, и понять сложные темы благодаря простому и точному языку. Считайте данную книгу профессионалом языка ActionScript, сидящим с вами за одним рабочим столом. Вы можете попросить его объяснить тонкости событийной архитектуры языка ActionScript, или распутать лабиринт системы безопасности приложения Flash Player, или продемонстрировать возможности встроенной поддержки языка XML (E4X). Кроме того, в этом издании вы можете найти информацию по таким недокументированным возможностям языка ActionScript, как, например, пространства имен, встроенные шрифты, доступ к загруженному содержимому, распространение библиотек классов, сборка мусора и обновления экрана.

Эта книга — настоящее руководство разработчика, содержащее практические пояснения, предупреждения, основанные на глубоком изучении предметной области, и полезные примеры кода, демонстрирующие способы правильного решения той или иной задачи.

Структура книги

Эта книга разделена на три части.

В части I «ActionScript с нуля» представлено подробное описание основ языка ActionScript, включая объектно-ориентированное программирование, классы, объекты, переменные, методы, функции, наследование, типы данных, массивы, события, исключения, область видимости, пространства имен, язык XML. Завершается часть I рассмотрением архитектуры безопасности приложения Flash Player.

В части II «Отображение и интерактивность» представлены методы отображения содержимого на экране и обработки событий ввода. К рассматриваемым темам относится интерфейс прикладного программирования (API) среды выполнения Flash, управляющий отображением содержимого на экране, иерархическая обработка событий, реализация интерактивности с использованием клавиатуры и мыши, анимация, векторная и растровая графика, текст и операции загрузки содержимого.

В части III «Практическое применение ActionScript» основное внимание уделяется вопросам применения кода, написанного на языке ActionScript. В этой части освещаются такие темы, как объединение кода ActionScript с ресурсами, созданными вручную в среде разработки Flash, использование платформы Flex Framework в программе Flex Builder и создание библиотеки пользовательского кода.

В качестве примера на протяжении всей книги создается с нуля и подробно анализируется полнофункциональная программа — виртуальный зоопарк.

Чего нет в этой книге

Экосистема языка ActionScript очень обширна. Охватить весь материал в одной книге просто невозможно. К вопросам, которые заслуживают внимания, но не рассматриваются подробно в этом издании, относятся:

- язык MXML;
- платформа Flex Framework;
- Flex Data Services;
- встроенные компоненты среды разработки Flash;
- Flash Media Server;
- Flash Remoting;
- поддержка регулярных выражений в языке ActionScript.

Информацию по перечисленным вопросам можно найти в документации от корпорации Adobe или на веб-странице Adobe Developer Library издательства O'Reilly по адресу http://www.oreilly.com/store/series/adl.csp.

Привязка к среде разработки

В этой книге рассматриваются базовые концепции программирования на языке ActionScript, применимые к любой среде разработки с использованием языка ActionScript 3.0 и к любой среде выполнения, поддерживающей ActionScript 3.0. По возможности я старался избегать вопросов, касающихся разработки кода с применением какого-либо конкретного инструмента, и уделял основное внимание концепциям программирования, а не использованию того или иного инструмента разработки. И все-таки в гл. 29 описывается использование языка ActionScript в среде разработки Flash, а в гл. 30 рассматриваются основы работы с платформой Flex Framework в приложении Flex Builder. Кроме того, в гл. 7 описывается процесс компиляции программы с использованием различных средств разработки (среды разработки Flash, приложения Flex Builder и компилятора mxmlc).

Сейчас обратим внимание на сам язык ActionScript. В следующих разделах представлена вводная техническая информация, касающаяся языка ActionScript 3.0, которая представляет интерес для опытных программистов. Если у вас нет опыта программирования, ознакомьтесь с разд. «Соглашения об обозначениях» и приступайте к изучению гл. 1.

Обзор языка ActionScript

ActionScript 3.0 представляет собой объектно-ориентированный язык программирования, применяемый для создания приложений и управляемого с помощью сценариев мультимедийного содержимого для воспроизведения в клиентских средах выполнения Flash (например, в приложениях Flash Player и Adobe AIR).

Благодаря синтаксису, напоминающему синтаксис языков Java и C#, базовый язык ActionScript наверняка покажется знакомым опытным программистам. Например, следующая строка кода создает переменную типа `int` (этот тип означает целое число) с именем `width`, которой присваивается значение `25`:

```
var width:int = 25;
```

Следующий фрагмент кода демонстрирует цикл `for` с проходом до 10:

```
for (var i:int = 1; i <= 10; i++) {
  // Расположенный здесь код будет выполнен 10 раз
}
```

А следующий фрагмент кода создает класс с именем `Product`:

```
// Описание класса
public class Product {
  // Переменная экземпляра типа Number
  var price:Number;

  // Метод-конструктор класса Product
  public function Product ( ) {
    // Расположенный здесь код инициализирует экземпляры класса Product
  }

  // Метод экземпляра
  public function doSomething ( ):void {
    // Расположенный здесь код выполняется всякий раз при вызове
    // метода doSomething( )
  }
}
```

Базовый язык

Базовый язык ActionScript 3.0 основан на четвертой редакции спецификации языка ECMAScript, которая на момент написания этой книги (май 2007 года) находилась в стадии разработки.

Со спецификацией языка ECMAScript 4 можно ознакомиться по адресу http://developer.mozilla.org/es4/spec/spec.html. Спецификация языка ActionScript 3.0 находится по адресу http://livedocs.macromedia.com/specs/actionscript/3.

В будущем планируется реализовать язык ActionScript в полном соответствии со спецификацией языка ECMAScript 4. Помимо языка ActionScript, спецификация ECMAScript также лежит в основе JavaScript — популярного языка веб-браузеров. Ожидается, что в будущей версии браузера Firefox 3.0 будет реализована поддержка языка JavaScript 2.0 с использованием того же базового кода, который применяется для ActionScript. Этот код был передан организации Mozilla Foundation корпорацией Adobe в ноябре 2006 года (дополнительную информацию можно найти по адресу http://www.mozilla.org/projects/tamarin).

Спецификация языка ECMAScript 4 налагает ограничения на базовый синтаксис и грамматику языка ActionScript — код, применяемый для создания таких элемен-

тов, как выражения, инструкции, переменные, функции, классы и объекты. Кроме того, спецификация языка ECMAScript 4 определяет небольшой набор встроенных типов данных для работы с распространенными значениями (например, `String`, `Number` и `Boolean`).

Ниже перечислены некоторые ключевые возможности базового языка ActionScript версии 3.0.

- Первоклассная поддержка наиболее распространенных объектно-ориентированных конструкций, например классов, объектов и интерфейсов.
- Однопоточная модель исполнения кода.
- Проверка типов на этапе выполнения.
- Дополнительная проверка типов на этапе компиляции.
- Динамические возможности, позволяющие, например, создавать новые методы-конструкторы и переменные на этапе выполнения.
- Исключения, генерируемые на этапе выполнения.
- Поддержка языка XML в качестве одного из встроенных типов данных.
- Пакеты для организации библиотек кода.
- Пространства имен для уточнения идентификаторов.
- Регулярные выражения.

Все клиентские среды выполнения Flash, поддерживающие язык ActionScript 3.0, в целом реализуют возможности базового языка. В этой книге полностью рассматривается базовый язык, за исключением регулярных выражений.

Клиентские среды выполнения Flash

Для исполнения программ, разработанных с использованием языка ActionScript, могут использоваться три различные клиентские среды выполнения: Adobe AIR, Flash Player и Flash Lite.

- **Adobe AIR.** Среда выполнения Adobe AIR исполняет Flash-приложения, предназначенные для развертывания на компьютере пользователя. Эта клиентская среда выполнения поддерживает содержимое в формате SWF, а также содержимое, подготовленное с использованием языков HTML и JavaScript. Среда выполнения Adobe AIR должна быть установлена на компьютере конечного пользователя на уровне операционной системы.

  Дополнительную информацию можно получить по адресу http://www.adobe.com/go/air.
- **Flash Player.** Среда выполнения Flash Player исполняет Flash-содержимое и Flash-приложения, предназначенные для развертывания в Интернете. Это приложение является предпочтительной средой выполнения для содержимого в формате SWF, интегрированного в веб-страницу. Flash Player обычно устанавливается в качестве дополнительного модуля к браузеру, но при этом он может работать и в автономном режиме.
- **Flash Lite.** Среда выполнения Flash Lite исполняет Flash-содержимое и Flash-приложения, предназначенные для развертывания на мобильных устройствах. Из-за ограниченной производительности мобильных устройств среда

выполнения Flash Lite обычно отстает от Flash Player и Adobe AIR как по скорости, так и по набору возможностей.

Рассмотренные клиентские среды выполнения Flash предоставляют общий базовый набор функциональных возможностей вместе со специфическими возможностями, которые удовлетворяют функциональным требованиям и требованиям безопасности для каждой конкретной среды выполнения. Например, во всех перечисленных средах выполнения Flash (Adobe AIR, Flash Player и Flash Lite) используется один и тот же синтаксис для создания переменной. Однако Adobe AIR включает интерфейсы прикладного программирования для управления окнами и работы с файловой системой, Flash Player налагает особые ограничения безопасности для защиты личной информации конечного пользователя в Интернете, а Flash Lite позволяет управлять функцией вибрации телефона.

API среды выполнения

Каждая клиентская среда выполнения Flash предоставляет собственный предопределенный набор функций, переменных, классов и объектов, который называется *интерфейсом прикладного программирования* (Application Programming Interface, API) среды выполнения. API каждой клиентской среды выполнения Flash имеет собственное имя. Например, API клиентской среды выполнения Flash, определяемый приложением Flash Player, называется *Flash Player API*.

Все API клиентской среды выполнения Flash определяют общий базовый набор функциональности. Например, каждая клиентская среда выполнения использует одинаковый набор классов для отображения содержимого на экране и обработки событий.

К ключевым возможностям, реализуемым всеми API клиентской среды выполнения Flash, относятся:

- отображение графики и видео;
- иерархическая событийная архитектура;
- отображение и ввод текста;
- управление с помощью мыши и клавиатуры;
- сетевые операции для загрузки данных из внешних источников и взаимодействия с серверными приложениями;
- воспроизведение аудио;
- печать;
- взаимодействие с внешними локальными приложениями;
- инструменты программирования.

В этой книге рассматриваются первые пять возможностей из приведенного списка. Информацию по другим специфическим API клиентской среды выполнения Flash можно найти в соответствующей документации по продукту.

Компоненты

Помимо API клиентской среды выполнения Flash, корпорация Adobe предоставляет два различных набора *компонентов* для выполнения общих задач програм-

мирования и построения пользовательского интерфейса. Приложение Flex Builder и бесплатный инструментарий разработчика Flex 2 SDK включают платформу Flex Framework, определяющую полный набор элементов управления пользовательского интерфейса, например RadioButton, CheckBox и List. Среда разработки Flash предоставляет аналогичный набор компонентов пользовательского интерфейса. Компоненты среды разработки Flash объединяют код и созданные вручную графические элементы, которые могут быть изменены разработчиками и дизайнерами, работающими в этой среде.

И набор компонентов платформы Flex Framework, и набор компонентов среды разработки Flash написаны полностью на языке ActionScript 3.0. Компоненты пользовательского интерфейса платформы Flex Framework в основном обладают более широкими возможностями по сравнению с компонентами среды разработки Flash и поэтому имеют больший файловый размер.

Компоненты пользовательского интерфейса платформы Flex Framework не могут быть использованы в среде разработки Flash, однако компоненты пользовательского интерфейса среды Flash могут быть использованы (и с юридической, и с технической точки зрения) в приложении Flex Builder и откомпилированы с помощью компилятора mxmlc.

В этой книге не рассматривается использование или создание компонентов с применением языка ActionScript. Информацию по компонентам можно найти в соответствующей документации по продукту.

Файловый формат Flash (SWF)

Код на языке ActionScript должен быть скомпилирован в SWF-файл для воспроизведения в одной из клиентских сред выполнения Flash. SWF-файл может содержать как байт-код ActionScript, так и включенные мультимедийные элементы (графику, звук, видео и шрифты). Одни SWF-файлы содержат только мультимедийные элементы без кода, а другие — только код без мультимедийных данных. Программа на языке ActionScript может размещаться как в одном SWF-файле, так и в нескольких. Когда программа разбита на несколько SWF-файлов, один определенный SWF-файл содержит точку входа программы и по мере необходимости загружает другие SWF-файлы. Разбиение сложной программы на несколько SWF-файлов упрощает ее дальнейшее сопровождение и, что касается приложений, размещаемых в Интернете, может обеспечить более быстрый доступ к различным частям программы.

Инструменты разработки приложений на языке ActionScript

Корпорация Adobe предлагает следующие инструменты для разработки приложений на языке ActionScript.

- **Adobe Flash** (http://www.adobe.com/go/flash/). Визуальное средство для дизайна и программирования, применяемое с целью создания мультимедийного содержимого, включающего графику, видео, аудио, анимацию и интерактивность. Программа Adobe Flash используется программистами для создания приложений

и мультимедийного содержимого путем объединения кода на языке ActionScript с нарисованными от руки изображениями, анимацией и мультимедийными элементами. Приложение Adobe Flash также называется *средой разработки Flash*. На момент написания этой книги (в июне 2007 года) самой последней, девятой, версией среды разработки Flash была CS3.

- **Adobe Flex Builder** (http://www.adobe.com/products/flex/productinfo/overview/). Инструмент разработки для создания содержимого с использованием либо чистого языка ActionScript, либо *MXML* — языка, основанного на XML и применяемого для описания пользовательских интерфейсов. В состав приложения Flex Builder входит платформа разработки, называемая Flex Framework, которая предоставляет широкий набор средств программирования и библиотеку элементов управления пользовательского интерфейса с изменяемым дизайном и стилями. Приложение Flex Builder, основанное на приложении Eclipse — популярном инструменте программирования с открытым кодом, — может использоваться как в режиме ручного написания кода, так и в режиме визуальной разработки (подобный режим применяется в приложении Visual Basic от корпорации Microsoft).
- **Adobe Flex 2 SDK** (http://www.adobe.com/products/flex/sdk/). Бесплатный инструментарий разработчика без графического интерфейса (работа осуществляется через командную строку) для создания содержимого с использованием либо чистого языка ActionScript 3.0, либо языка MXML. Инструментарий разработчика Flex 2 SDK включает в себя платформу Flex Framework и консольный компилятор mxmlc (эти компоненты также входят в состав приложения Adobe Flex Builder). Инструментарий Flex 2 SDK позволяет разработчикам бесплатно создавать содержимое в любом редакторе программного кода по их желанию. (Список инструментов и средств разработки приложений на языке ActionScript с открытым кодом можно найти по адресу http://osflash.org.)

Файлы примеров

Официальный сайт, осуществляющий поддержку этой книги на английском языке, доступен по адресу http://moock.org/eas3.

Загрузить файлы примеров для данной книги на английском языке можно по адресу http://moock.org/eas3/examples.

Обратите внимание, что большинство примеров в книге представлено в контексте объемлющего основного класса, который должен быть откомпилирован как FLA-файл *класса документа* (средство разработки Flash) или как *заданный класс приложения* проекта (приложение Flex Builder).

Использование примеров кода

Данная книга призвана помочь решить поставленные перед вами задачи. Вы можете использовать код из этой книги в своих программах и документации. Если вы не

воспроизводите значительную часть кода, вам не нужно связываться с нами для получения разрешения. Например, написание программы, в которой используется несколько фрагментов кода из этой книги, не требует получения разрешения. Ответ на вопрос путем цитирования текста и примера кода из этой книги также не требует получения разрешения. Включение значительной части примеров кода из данной книги в вашу документацию по продукту требует получения разрешения. Продажа или распространение дисков с примерами из книг издательства O'Reilly также требует получения разрешения.

Мы будем признательны за указание ссылки на цитируемый источник, хотя и не требуем этого.

Если вы считаете, что использование вами примеров кода исходя из перечисленных ситуаций не является законным, можете в любой момент обращаться к нам за консультацией по электронной почте: permissions@oreilly.com.

Соглашения об обозначениях

Чтобы выделить различные синтаксические компоненты языка ActionScript, в этой книге применяются следующие обозначения.

- **Команды меню.** Команды меню записываются с использованием символа ▸, например File ▸ Open (Файл ▸ Открыть). Кроме того, шрифт используется для обозначения ссылок.
- `Моноширинный шрифт`. Обозначает примеры и фрагменты кода, имена переменных и параметров, а также имена функций, методов, классов и пакетов, имена файлов, типы данных, ключевые слова, объекты.
- `Шрифт с фиксированной шириной`. Применяется для отображения примеров кода.
- **`Полужирный шрифт с фиксированной шириной`**. Обозначает текст, который должен быть введен дословно при выполнении пошаговой процедуры. Кроме того, **`полужирный шрифт с фиксированной шириной`** иногда используется в примерах кода для создания логического ударения, например, чтобы выделить важную строку кода в большом примере.
- *`Курсивный шрифт с фиксированной шириной`*. Обозначает код, который должен быть заменен подходящим значением (например, *`укажитеВашеИмя`*).

Это совет. Советы содержат полезную информацию по рассматриваемой теме, зачастую выделяя важные концепции или лучшие практические решения.

Это предупреждение. Предупреждения помогают решить возникающие проблемы и избежать их в дальнейшем. Вы можете игнорировать предупреждения на свой страх и риск.

Это примечание. Оно содержит полезную прикладную информацию по рассматриваемой теме. Кроме того в примечании выполняется сравнение и сопоставление версии 2.0 языка ActionScript с версией 3.0 с тем, чтобы помочь вам перейти на ActionScript 3.0 и понять важные различия между двумя версиями языка.

В книге применяются следующие соглашения по кодированию и терминологии.

- Ключевое слово `this` записывается моноширинным шрифтом, поскольку оно является неявным параметром, передаваемым в методы и функции.
- Вообще, при обращении к идентификаторам внутри методов экземпляра ключевое слово `this` не используется. Тем не менее оно применяется для устранения неоднозначности, когда имена переменных и методов экземпляра совпадают с именами параметров и локальных переменных.
- При обсуждении методов-аксессоров и методов-мутаторов в этой книге не используются традиционные термины *аксессор*, *мутатор*, *читатель* и *писатель*. Вместо этого применяются неофициальные термины *метод-получатель* и *метод-модификатор*.
- При описании класса, содержащего статические переменные, статические методы, переменные экземпляра, методы экземпляра и метод-конструктор, в этой книге сначала перечисляются статические переменные, затем указываются статические методы, переменные экземпляра, метод-конструктор класса и, наконец, методы экземпляра.
- В этой книге имена констант записываются ПРОПИСНЫМИ БУКВАМИ.
- При обращении к статическим переменным и статическим методам в этой книге всегда указывается имя класса, в котором определены данные переменная или метод.
- Если не указано другое, вместо понятия *замыкание функции* используется более простое понятие *функция*. Описание различий между этими двумя понятиями можно найти в гл. 5.
- В этой книге подразумевается, что компиляция кода осуществляется с применением строгого режима. Более того, после гл. 7 для всех переменных, параметров и возвращаемых значений будет использоваться объявление типов.
- Для именования обработчиков событий в книге применяется следующий формат: `имяСобытияListener`, где `имяСобытия` — это строковое имя события.

Как с нами связаться

Мы приложили максимум усилий, чтобы протестировать и проверить информацию, содержащуюся в этой книге, но, возможно, вы обнаружите некоторые неточности (или даже места, где мы ошиблись!). Пожалуйста, информируйте нас о любых обнаруженных ошибках, а также высказывайте ваши предложения для будущих редакций книги, используя следующую контактную информацию:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

(800) 998–9938 (в Соединенных Штатах Америки или Канаде)

(707) 829–0515 (международный/местный)

(707) 829–0104 (факс)

Мы создали веб-страницу для этой книги, где представлен список опечаток, перечислены примеры и собрана другая дополнительная информация. Эта страница доступна по адресу http://www.oreilly.com/catalog/9780596526948.

Отправлять свои комментарии и технические вопросы, касающиеся этой книги, вы можете на следующий электронный адрес: bookquestions@oreilly.com.

Дополнительную информацию о наших книгах, конференциях, программном обеспечении, ресурсных центрах и сети издательства O'Reilly можно найти на нашем сайте по адресу http://www.oreilly.com.

Safari® Enabled

Если на обложке вашей любимой книги по компьютерным технологиям вы видите значок Safari® Enabled, это значит, что данная книга доступна в интернет-каталоге издательства O'Reilly — O'Reilly Network Safari Bookshelf.

Интернет-каталог Safari предлагает лучшее решение по сравнению с обычными книгами в электронном варианте. Это виртуальная библиотека, которая окажет неоценимую помощь в тех случаях, когда вам понадобится наиболее достоверная и актуальная информация. С ее помощью вы можете легко осуществлять поиск нужной информации среди тысяч ведущих книг по компьютерным технологиям, копировать и вставлять примеры кода, загружать главы и получать быстрые ответы на интересующие вас вопросы. Бесплатно ознакомиться с возможностями интернет-каталога Safari вы можете по адресу http://safari.oreilly.com.

Благодарности

Эта книга была написана благодаря оказанному доверию и активной поддержке как со стороны корпорации Adobe, так и со стороны издательства O'Reilly. Летом 2005 года на встрече со Стивом Вайссом (Steve Weiss), Лизой Фрэндли (Lisa Friendly) и Майком Чамберсом (Mike Chambers) я согласился написать книгу под названием «Essential ActionScript 3.0». Изначально предполагалось, что новая книга станет «небольшим обновлением книги "Essential ActionScript 2.0"». Но бурное развитие языка ActionScript 3.0 привело к тому, что книга «Essential ActionScript 3.0» переросла в самостоятельный продукт. Корпорация Adobe и издательство O'Reilly терпеливо и стойко наблюдали за бесконечным увеличением размеров книги и в итоге согласились перенести предельный срок публикации с девяти месяцев до двух лет. На протяжении всего времени я твердо верил, что мы сделали правильный выбор, и для меня большой честью является тот факт, что корпорация Adobe и издательство O'Reilly согласились с этим.

В процессе написания этой книги корпорация Adobe любезно предоставляла мне полный доступ к внутренним ресурсам и выделяла официальное инженерное время для технических рецензий. Я чрезвычайно благодарен Джу Ли Бардекину (Ju Lee Burdekin) и Фрэнсису Чену (Francis Cheng) из команды разработки технической документации корпорации Adobe. Джу Ли и Фрэнсис координировали мои усилия внутри корпорации Adobe и отвечали на бесконечный, как мне казалось, поток вопросов.

Множество сотрудников корпорации Adobe предоставляли мне информацию и проводили инструктаж в процессе подготовки материала для будущей книги. Я крайне признателен каждому из них и хочу особо поблагодарить следующих людей.

- Фрэнсис Чен был постоянным источником информации по базовому языку и оказал неоценимую помощь при рецензировании рукописи книги «Essential ActionScript 3.0». Фрэнсис входит в состав комитета, занимающегося разработкой спецификации языка ECMAScript 4, и является одним из авторов спецификации языка ActionScript 3.0.
- Джефф Дайер (Jeff Dyer) постоянно выкраивал время в своем графике для помощи в уточнении концепций базового языка и поиске ошибок. Джефф является одним из основных разработчиков компилятора языка ActionScript 3.0, основным автором спецификации языка ActionScript 3.0 и главным членом комитета, занимающегося разработкой спецификации языка ECMAScript 4.
- Дене Мекета (Deneb Meketa) терпеливо выдерживал мое неправильное понимание системы безопасности клиентской среды выполнения Flash Player. Телефонные звонки и переписка по электронной почте в процессе интенсивных исследований, продолжавшихся более месяца, помогли внести ясность в гл. 19. Дене является инженером, отвечающим за реализацию системы безопасности в среде выполнения Flash Player.
- Джефф Мотт (Jeff Mott) — инженер среды выполнения Flash Player — постоянно давал исчерпывающие и практически мгновенные ответы на мои вопросы, касающиеся событийной системы языка ActionScript.

- Джим Корбетт (Jim Corbett), являющийся инженером среды выполнения Flash Player, помог мне понять многие тонкости дисплейного списка и процесса загрузки событий.
- Ребекка Сан (Rebecca Sun) — инженер среды разработки Flash — ответила на множество вопросов, касающихся связей между компилятором языка ActionScript 3.0 и приложением Flash CS3. Кроме того, она приветливо выслушивала предложения и терпела мои частые спонтанные просьбы через систему обмена мгновенными сообщениями.
- Ли Томасон (Lee Thomason) — системный архитектор среды Flash Player — прочитал мне персональную лекцию по механизму отображения текста в этой среде.
- Роджер Гонзалез (Roger Gonzalez) — системный архитектор компилятора Flex — регулярно отвечал на мои вопросы, касающиеся процесса загрузки классов и компилятора Flex.
- Вернер Шарп (Werner Sharp), являющийся инженером среды выполнения Flash Player, объяснил множество тонких моментов, связанных с обработкой растровых изображений в языке ActionScript.
- Пол Бетлем (Paul Betlem) — ведущий руководитель команды разработчиков среды выполнения Flash Player — помог облегчить процесс технического рецензирования и лично просмотрел несколько глав.
- Майк Чамберс (Mike Chambers) — ведущий менеджер по связям между разработчиками среды выполнения Adobe AIR — предоставлял актуальную техническую информацию и участвовал в развитии проекта «Essential ActionScript 3.0» со времени его появления.
- Гари Гроссман (Gary Grossman), который является первым создателем языка ActionScript, научил меня многому из того, что я знаю о программировании для платформы Flash. В августе 2006 года Гари объединился с изобретателями платформы Flash (Джоном Гейем (Jon Gay) и Робертом Тацуми (Robert Tatsumi)), чтобы вместе основать новую компанию, Software as Art (http://www.softwareasart.com).

Для меня было честью познакомиться и работать с другими сотрудниками корпорации Adobe — бывшими и настоящими, — которые перечислены далее: Майк Дауни (Mike Downey), Кевин Линч (Kevin Lynch), Пол Бетлем, Эдвин Смит (Edwin Smith), Кристин Ярроу (Christine Yarrow), Джефф Камерер (Jeff Kamerer), Нигель Пегг (Nigel Pegg), Матт Вобенсмит (Matt Wobensmith), Томас Рейли (Thomas Reilly), Джетро Виллегас (Jethro Villegas), Роб Диксон (Rob Dixon), Джефф Шварц (Jeff Swartz), Валид Анбар (Waleed Anbar), Крис Тилген (Chris Thilgen), Джиллс Дрю (Gilles Drieu), Нивеш Райбхандари (Nivesh Rajbhandari), Тей Ота (Tei Ota), Акио Танака (Akio Tanaka), Суми Лим (Sumi Lim), Трой Эванс (Troy Evans), Джон Доуделл (John Dowdell), Бентли Вольф (Bentley Wolfe), Тиник Уро (Tinic Uro), Майкл Вильямс (Michael Williams), Шарон Селдон (Sharon Seldon), Джонатан Гэй (Jonathan Gay), Роберт Тацуми (Robert Tatsumi), Пете Сантангели (Pete Santangeli), Марк Андерс (Mark Anders), Джон Нэк (John Nack), Матт Чотин (Matt Chotin), Алекс Харуй (Alex Harui), Гордон Смит (Gordon Smith), Шо Кувамото (Sho Kuwamoto), Крейг Гудман (Craig Goodman), Стефан Грюнведель (Stefan Gruenwedel), Дипа Субраманиам (Deepa Subramaniam), Этан Маласки (Ethan Malasky), Син Кранзберг

(Sean Kranzberg), Майкл Моррис (Michael Morris), Эрик Виттман (Eric Wittman), Джереми Кларк (Jeremy Clark) и Джанис Пирс (Janice Pearce).

Из табл. 0.1, в которой представлена статистическая информация, вы можете понять, насколько я благодарен официальным техническим рецензентам.

Таблица 0.1. Рецензенты корпорации Adobe

Рецензент	Должность	Просмотренные главы	Количество ответов на электронные письма
Дене Мекета	Специалист по компьютерным технологиям, платформа Flash	17	75
Эрика Нортон (Erica Norton)	Ведущий инженер по качеству, среда выполнения Flash Player	14, 19, 21, 22	3
Фрэнсис Чен	Ведущий технический писатель	1–11, 13, 15, 16, 18	334
Джефф Дайер	Системный архитектор компиляторов, группа разработки языка ActionScript	17	106
Джефф Мотт	Специалист по компьютерным технологиям, группа разработки среды выполнения Flash Player	12, 20-25	85
Джим Корбетт	Ведущий специалист по компьютерным технологиям, группа разработки среды выполнения Flash Player	20, 23, 24, 28	52
Ли Томасон	Системный архитектор, среда выполнения Flash Player	25, 27	33
Майк Чамберс	Ведущий менеджер по связям между разработчиками, среда выполнения Adobe AIR	1	89
Майк Ричардс (Mike Richards)	Специалист по компьютерным технологиям, мобильные телефоны и устройства	22–26	9
Пол Робертсон (Paul Robertson)	Разработчик/кодировщик языка ActionScript	1, 2, 24, 27–31	14
Пол Бетлем	Ведущий руководитель, группа разработки среды выполнения Flash Player	20, 27, 26	19
Ребекка Сан	Специалист по компьютерным технологиям, среда разработки Flash	7, 29, 31	60
Роберт Пеннер (Robert Penner)	Ведущий инженер, среда разработки Flash	18, 23–25	16
Роджер Гонзалез	Системный архитектор компилятора Flex	25, 30, 31	64
Вернер Шарп	Ведущий специалист по компьютерным технологиям, группа разработки среды выполнения Flash Player	18, 22	35

Спасибо Робину Томасу (Robyn Thomas) — редактору этой книги, — который просмотрел и «отполировал» рукопись с потрясающей скоростью и точностью. Спасибо и всем членам руководящего состава издательства O'Reilly, а также редакторского, производственного, декораторского и художественного отделов, отдела маркетинга и отдела сбыта, включая Тима О'Рейли (Tim O'Reilly), Стива Вайсса и Карена Монтгомери

(Karen Montgomery). Спасибо литературному редактору Филиппу Данглеру (Philip Dangler), который помог сделать текст связным, читабельным и безошибочным.

Помимо технического рецензирования сотрудниками корпорации Adobe, эта книга прошла проверку на ошибки и качество независимой группой читателей, куда входили Бретт Волкер (Brett Walker), Чафик Казун (Chafic Kazoun), Дерек МакКенна (Derek McKenna), Эдвин ван Рийком (Edwin van Rijkom), Грег Бурч (Greg Burch), Джим Армстронг (Jim Armstrong), Джон Вильямс (Jon Williams), Марк Джонкман (Mark Jonkman), Мэттью Кифи (Matthew Keefe), Мауро Ди Бласи (Mauro Di Blasi), Ральф Бокельберг (Ralf Bokelberg), Рик Евинг (Ric Ewing), Робин Дебрюил (Robin Debreuil) и Виктор Аллен (Victor Allen). Эти читатели оказали неоценимую помощь, обнаружив множество несоответствий и неявных ошибок в примерах кода. Хочется отдельно отметить Марка Джонкмана за его чрезвычайно тщательное изучение рукописи книги и примеров кода.

Еще хочу поблагодарить двух наставников, которые помогли мне стать программистом и писателем. Это Брюс Эпстейн (Bruce Epstein) и Дерек Клейтон (Derek Clayton). Брюс занимался редактированием всех моих предыдущих книг, и его ценные наставления до сих пор наполняют каждое слово, написанное мой. Дерек является создателем многопользовательского сервера Unity, размещенного на сайте moock.org (http://www.moock.org/unity), постоянным источником идей для программирования и просто хорошим другом.

И, конечно же, ни одна книга по языку, основанному на спецификации языка ECMAScript, не будет полной без слов благодарности в адрес Брендана Эйка (Brendan Eich) за создание языка JavaScript и за участие в продолжающейся разработке спецификации языка ECMAScript. Спасибо, Брендан!

В заключение хочу пожелать любви и согласия следующим людям за их любовь и дружескую поддержку: Джеймсу Портеру (James Porter), Граму Бартону (Graham Barton), Джо Дуону (Joe Duong), Томми Якобсу (Tommy Jacobs), Венди Шаффер (Wendy Schaffer), Эндрю Харрису (Andrew Harris), Дейву Люкстону (Dave Luxton), Дэйву Комлосу (Dave Komlos), Марко Кроулею (Marco Crawley), Эрику Липхардту (Eric Liphardt), Кену Реддику (Ken Reddick), Майку Линковичу (Mike Linkovich), Матту Верну (Matt Wearn), Майку Добеллу (Mike Dobell), Майку «Найсу» (Mike «Nice»), Хоссу Гиффорду (Hoss Gifford), Эрику Нацке (Erik Natzke), Яреду Тарбеллу (Jared Tarbell), Маркосу Вескампу (Marcos Weskamp), Дану Албриттону (Dan Albritton), Фрэнсису Бурре (Francis Bourre), Тийсу Тремстра (Thijs Triemstra), Веронике Бросье (Veronique Brossier), Сайме Хохар (Saima Khokhar), Амиту Питару (Amit Pitaru), Джеймсу Паттерсону (James Patterson), Джошуа Дэвису (Joshua Davis), Брендену Холлу (Branden Hall), Роберту Ходгину (Robert Hodgin), Шину Мацумуре (Shin Matsumura), Юго Накамуре (Yugo Nakamura), Клаусу Валерсу (Claus Whalers), Даррону Шоллу (Darron Schall), Марио Клингеману (Mario Klingeman), Фумио Нонаке (Fumio Nonaka), Роберту Рейнхардту (Robert Reinhardt), Грану Скиннеру (Grant Skinner) и семейству Муков.

Колин Мук
Март 2007 года
Торонто, Канада

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты dgurski@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

ActionScript с нуля

В части I представлено подробное описание языка ActionScript 3.0, включая принципы объектно-ориентированного программирования, классы, объекты, переменные, методы, функции, наследование, типы данных, массивы, события, исключения, области видимости, пространства имен и язык XML. В конце части рассматривается архитектура безопасности приложения Flash Player.

Прочитав часть I, вы получите глубокие знания основ языка ActionScript 3.0 и сможете применить их на практике, разрабатывая пример-приложение создания виртуального зоопарка.

- Глава 1 «Основные понятия».
- Глава 2 «Условные операторы и циклы».
- Глава 3 «Пересмотр методов экземпляра».
- Глава 4 «Статические переменные и методы».
- Глава 5 «Функции».
- Глава 6 «Наследование».
- Глава 7 «Компиляция и выполнение программы».
- Глава 8 «Типы данных и проверка типов».
- Глава 9 «Интерфейсы».
- Глава 10 «Инструкции и операторы».
- Глава 11 «Массивы».
- Глава 12 «События и обработка событий».
- Глава 13 «Обработка исключений и ошибок».
- Глава 14 «Сборка мусора».
- Глава 15 «Динамические возможности языка ActionScript».
- Глава 16 «Область видимости».
- Глава 17 «Пространства имен».
- Глава 18 «Язык XML и расширение E4X».
- Глава 19 «Ограничения безопасности Flash Player».

Основные понятия

Программа — это набор команд, выполняемых (или исполняемых) компьютером или приложением. Текст программы, записанный в виде, удобном для восприятия человеком, называется *исходным кодом*, или просто кодом, а человек, который создает программу, — *программистом*, *кодировщиком* или *разработчиком*. Для написания программ используются определенные языки программирования. Они определяют синтаксис и грамматику, которые должны использовать программисты при написании инструкций к создаваемой программе. В настоящей книге подробно описываются особенности языка программирования ActionScript 3.0. Приготовьтесь приятно провести время.

Инструменты для написания кода на языке ActionScript

Код на языке ActionScript представляет собой обычный текст, поэтому программы на его основе можно создать даже с помощью простейшего текстового редактора, например Блокнота операционной системы Windows или приложения TextEdit операционной системы Macintosh. Тем не менее в среде программистов ActionScript наиболее распространены два коммерческих приложения, являющиеся разработками корпорации Adobe Systems Incorporated: программа Flex Builder и среда разработки Flash.

Программа Flex Builder представляет собой интегрированную среду разработки, или *IDE* (Integrated Development Environment). IDE — это приложение, предназначенное для написания и управления кодом, во многом напоминающее текстовый процессор, предназначенный для создания печатных документов. Разработчики используют программу Flex Builder для создания приложений и мультимедийного содержимого с помощью языков ActionScript, MXML или их обоих. Язык *MXML* основан на языке XML и применяется для описания пользовательских интерфейсов.

В отличие от Flex Builder, среда разработки *Flash* — это инструмент, сочетающий в себе редакторы проекта, анимации и программный редактор. Она используется программистами для создания приложений и мультимедийного содержимого путем объединения кода на языке ActionScript с нарисованными от руки изображениями, анимацией и мультимедийными элементами.

Язык ActionScript 3.0 поддерживается программами Flex Builder и Flash CS3 (другое название среды разработки Flash версии 9) и их более поздними версиями.

Найти программу Flex Builder можно по адресу: http://www.adobe.com/products/flex/. Среда разработки Flash доступна по адресу: http://www.adobe.com/products/flash/.

Большая часть этой книги посвящена созданию приложений и мультимедийного содержимого с использованием исключительно кода ActionScript. В гл. 29 описывается использование языка ActionScript в среде разработки Flash. Рассмотрение языка MXML выходит за рамки данной книги. Ознакомиться с ним можно в книге «Programming Flex 2» (Kazoun and Lott, 2007) издательства O'Reilly или в документации по программе Adobe Flex Builder.

Клиентские среды выполнения Flash

Для выполнения программ, написанных на языке ActionScript, могут применяться три различных приложения, разработанные корпорацией Adobe: Flash Player, Apollo и Flash Lite.

Приложение Flash Player позволяет выполнять программы на ActionScript в браузере или в автономном режиме на Рабочем столе. Оно обладает крайне ограниченным доступом к операционной системе (например, выполняемая программа не может управлять файлами, контролировать окна или получать доступ к большинству аппаратных устройств компьютера).

Приложение Apollo позволяет выполнять программы на ActionScript на Рабочем столе и является полностью интегрированным с операционной системой (например, выполняемая программа может осуществлять любые операции, в том числе и те, доступ к которым невозможен при использовании приложения Flash Player: управлять файлами, контролировать окна и получать доступ к аппаратному обеспечению компьютера).

Приложение Flash Lite предназначено для использования на мобильных устройствах, например на сотовых телефонах. На момент издания этой книги приложение Flash Lite было способно выполнять программы, написанные только на языке ActionScript 2.0 (но не на языке ActionScript 3.0). В то же время приложения Flash Player и Apollo позволяют выполнять программы, написанные на языке ActionScript 3.0. Таким образом, методики, изложенные в этой книге, применимы только к приложениям Flash Player и Apollo (до тех пор пока в приложении Flash Lite не будет реализована поддержка языка ActionScript 3.0).

В общем смысле, приложения Flash Player, Apollo и Flash Lite называются *клиентскими средами выполнения Flash* (или сокращенно *средами выполнения Flash*), поскольку они управляют программами на ActionScript в процессе их выполнения, или «прогона». Среды выполнения Flash доступны для операционных систем Windows, Macintosh и Linux, а также для различных мобильных устройств. Поскольку программы на ActionScript выполняются средой выполнения Flash, а не конкретной операционной системы или аппаратного устройства, то любую программу на языке ActionScript можно считать переносимой, так как она может выполняться на различных аппаратных устройствах (телефонах, игровых приставках) и в операционных системах (Windows, Macintosh и Linux).

Зачастую термин *«виртуальная машина ActionScript»* используется как аналог термина *«клиентская среда выполнения Flash»*. Однако на деле между этими двумя терминами существует разница и они не являются взаимозаменяемыми. Технически виртуальная машина ActionScript (AVM — ActionScript virtual machine) представляет собой программный модуль, являющийся *частью* приложений Flash Player, Apollo и Flash Lite, который выполняет программы на ActionScript. В то же время на любую среду выполнения Flash возлагаются и другие задачи, например отображение содержимого на экране, воспроизведение видео и аудио, взаимодействие с операционной системой. Версия виртуальной машины ActionScript, позволяющая выполнять код ActionScript 3.0, получила название *AVM2*. Версия виртуальной машины ActionScript, позволяющая выполнять код ActionScript 1.0 и ActionScript 2.0 (данные версии языка ActionScript не рассматриваются в этой книге), получила название *AVM1*.

Компиляция

Прежде чем программа, написанная на языке ActionScript, будет обработана средой выполнения Flash, код ActionScript 3.0 должен быть из формы, понятной программисту, преобразован в сжатый, двоичный формат, принятый в среде выполнения Flash и называемый *байт-кодом ActionScript*, или *ABC*. Однако сам по себе байт-код ActionScript не может быть исполнен средой Flash; он должен быть помещен в бинарный файл-контейнер с расширением SWF. Для хранения в SWF-файле байт-кода и всех включенных мультимедийных элементов, необходимых ActionScript, применяется *формат файла Flash SWF*. Процесс преобразования программы на ActionScript в байт-код называется *компиляцией программы*. Процесс генерации SWF-файла называется *компиляцией SWF-файла* или иногда — *экспортированием* или *публикацией SWF-файла*.

Для компиляции программ ActionScript 3.0 и SWF-файлов используется программный модуль, называемый *компилятором*. Компилятор, применяемый для преобразования кода ActionScript, называется *компилятором ActionScript*. Компилятор, применяемый для генерации SWF-файлов, называется *компилятором SWF*. Любой компилятор SWF, реализующий полную поддержку формата файла Flash, включает компилятор ActionScript. Естественно, компилятор SWF (и как следствие, компилятор ActionScript) входит в состав приложения Flex Builder и среды разработки Flash. Приложение Flex Builder и среда разработки Flash используют один и тот же компилятор ActionScript, но при этом имеют различные компиляторы SWF, называемые *компилятором Flex* и *компилятором Flash* соответственно. Кроме того, компилятор Flex доступен в виде отдельного консольного приложения mxmlc, которое входит в состав бесплатного инструментария разработчика корпорации Adobe — *Flex 2 SDK*, и может быть загружен по адресу: http://www.adobe.com/go/flex2_sdk.

Динамическая компиляция. В процессе выполнения программы на ActionScript среда Flash читает скомпилированный байт-код ActionScript и преобразует его в машинные команды, определенные для конкретного аппаратного обеспечения

компьютера, на котором выполняется данная программа. В большинстве случаев преобразованные машинные команды сохраняются в памяти для дальнейшего использования, благодаря чему отпадает необходимость в повторном преобразовании байт-кода ActionScript.

Как и процесс преобразования кода ActionScript 3.0 в байт-код, процесс преобразования байт-кода ActionScript в машинный код и его последующее сохранение для дальнейшего выполнения называется компиляцией. Таким образом, для большинства программ на ActionScript компиляция выполняется в два этапа.

На первом этапе разработчик компилирует код из удобного для чтения формата в формат, который понимает среда выполнения Flash (байт-код ActionScript). После этого среда выполнения Flash автоматически компилирует байт-код ActionScript в понятный конкретному аппаратному обеспечению формат, на котором выполняется программа (машинный код). Такой вид компиляции (байт-код в машинный код) называется *динамической компиляцией*, или *JIT* (Just-In-Time), поскольку она происходит непосредственно перед тем моментом, когда программе потребуется определенный фрагмент скомпилированного байт-кода. Динамическая компиляция иногда называется *динамической трансляцией*. Опытным программистам, возможно, будет интересно узнать, что динамическая компиляция не применяется для кода, находящегося на верхнем уровне описания класса (поскольку этот код выполняется всего один раз).

Краткий обзор

Выше было рассмотрено множество базовых понятий. Теперь подведем промежуточные итоги.

Программа, написанная на языке ActionScript, представляет собой набор инструкций, исполняемых одной из существующих сред выполнения Flash: приложением Flash Player, Apollo или Flash Lite. Программы на языке ActionScript можно создавать в обычном текстовом редакторе, в приложении Flex Builder или в среде разработки Flash. Перед выполнением программа должна быть скомпилирована в SWF-файл с помощью компилятора SWF, в качестве которого может выступать компилятор Flash, входящий в состав среды разработки Flash, или компилятор mxmlc, входящий в состав приложения Flex Builder и инструментария разработчика Flex 2 SDK.

Не волнуйтесь, если некоторые из понятий или терминов вам совершенно незнакомы. Вы сможете ознакомиться с ними в следующих разделах.

Теперь приступим к написанию кода.

Классы и объекты

Представьте, что вы собираетесь построить самолет. Обдумайте этапы предстоящей работы. Вряд ли вы сразу направитесь в магазин за металлом, чтобы приступить

к сварке. Для начала необходимо подготовить чертеж будущего самолета. На самом же деле, принимая во внимание тот факт, что вы строите самолет с нуля, необходимо подготовить не один, а несколько чертежей — по одному для каждой части самолета (колес, крыльев, кресел, тормозов и т. д.). Каждый чертеж должен в полной мере описывать определенную часть конструкции и соответствовать реальной детали в физической реализации. Для построения самолета необходимо изготовить каждую деталь по отдельности, а затем собрать готовые составляющие в соответствии с основным чертежом. Взаимодействие собранных частей самолета будет определять его поведение.

Если приведенные примеры понятны для вас, значит, у вас есть все необходимое, чтобы стать программистом на языке ActionScript. Выполняющаяся программа, как и самолет, летящий высоко в небе и представляющий собой совокупность взаимодействующих частей, разработанных по набору чертежей, является совокупностью взаимодействующих *объектов*, построенных из набора *классов*. В программе объекты ActionScript представляют собой как материальные предметы, так и неосязаемые понятия. Например, объект может представлять число в вычислении, интерактивную кнопку пользовательского интерфейса, момент времени на календаре или эффект размытия на изображении. Объекты являются воплощениями, или *экземплярами*, классов. Иначе говоря, классы — это чертежи, по которым создаются объекты.

Первый шаг в написании новой программы заключается в определении классов. Каждый класс с помощью кода описывает характеристики и поведение определенного типа объекта. Некоторые классы программы должны быть написаны с нуля, тогда как другие классы предоставляются языком ActionScript и различными средами выполнения Flash. Классы, написанные с нуля (называемые *пользовательскими классами*), используются для представления объектов специализированного типа, например формы заказа в интернет-магазине, автомобиля в гоночном симуляторе или сообщения в программе обмена текстовыми сообщениями. В отличие от этого, классы, предоставляемые языком ActionScript и различными средами выполнения Flash (называемые *предопределенными классами*), используются для выполнения фундаментальных задач, например для представления чисел и текста, воспроизведения звука, вывода изображений, обеспечения доступа к сети и формирования ответа на запрос пользователя.

Из классов, описанных в программе, мы формируем объекты (или создаем экземпляры), а затем управляем этими объектами, давая указания к выполнению тех или иных действий. Действия, выполняемые объектами, определяют поведение программы.

Процесс создания программы с помощью классов и объектов называется объектно-ориентированным программированием (ООП).

Прежде чем приступить к написанию программы, кратко рассмотрим важную группу классов, называемых *собственными классами* и являющихся частью языка ActionScript. Собственные классы, приведенные в табл. 1.1, применяются для работы с основными типами данных, например с числами и текстом. Логично предположить, что в каждой создаваемой программе вы будете использовать экземпляры

по крайней мере одного или двух собственных классов языка ActionScript — аналогично использованию готовых деталей от стороннего производителя при построении самолета. Изучите табл. 1.1, чтобы получить общее представление об этих классах. В следующих разделах собственные классы будут рассмотрены более подробно.

Таблица 1.1. Собственные классы языка ActionScript

Класс	Описание
String	Представляет текстовые данные (то есть строку или символы)
Boolean	Определяет логические состояния true (истина) или false (ложь)
Number	Представляет числа с плавающей запятой (то есть числа с дробной частью)
Int	Определяет целые числа (то есть числа без дробной части)
Uint	Представляет положительные целые числа
Array	Определяет упорядоченный список
Error	Представляет ошибку в программе (то есть проблему в вашем коде)
Date	Представляет определенный момент времени
Math	Содержит распространенные математические величины и операции
RegExp	Определяет инструменты для поиска и замены текста
Function	Представляет многократно используемый набор инструкций, которые могут быть вызваны и исполнены повторно
Object	Определяет базовые возможности всех объектов языка ActionScript

Теперь попробуем применить классы и объекты в примере программы — простом приложении, имитирующем зоопарк с виртуальными животными.

Применение методики, называемой созданием сценариев на временной шкале, в среде разработки Flash позволяет создавать программы на ActionScript без предварительного определения класса (дополнительную информацию можно найти в гл. 29). Тем не менее, даже если в будущем вы не планируете создавать классы самостоятельно, я настоятельно рекомендую вам познакомиться с методиками, рассмотренными в этой главе. Знакомство с процессом создания классов позволит значительно углубить ваше понимание языка ActionScript в целом и поможет вам стать более профессиональным программистом.

Создание программы

Как уже было сказано, программы на языке ActionScript построены из классов, которые являются «чертежами» взаимодействующих между собой частей (объектов) программы. Обычно разработка новой программы на ActionScript начинается с фазы проектирования, в течение которой функциональные возможности программы разбиваются на логически связанные классы. Каждому классу присваивается имя, определяются его свойства и роль в создаваемой программе. Один класс назначается *основным классом*. Он содержит стартовую точку, то есть *программную точку входа*, для приложения. Для запуска новой программы в среде выполнения Flash автоматически создается экземпляр основного класса программы.

Основному классу нашего примера программы по созданию виртуального зоопарка мы присвоим имя `VirtualZoo`. Сначала создадим папку с именем virtualzoo

в файловой системе компьютера, внутри которой создадим вложенную папку src (сокращенно от слова source — «исходный код»). В ней будут храниться все файлы с расширением AS (то есть все файлы, содержащие исходный код).

Исходный код каждого основного класса программы должен размещаться в отдельном текстовом файле, имя которого состоит из имени основного класса и расширения AS. Таким образом, необходимо создать пустой текстовый файл с именем `VirtualZoo.as`. Нужно убедиться, что имя файла `VirtualZoo.as` полностью совпадает с именем класса `VirtualZoo`, поскольку в данном случае учитывается регистр символов. Далее поместим файл `VirtualZoo.as` в папку virtualzoo/src. Тогда текущая файловая структура для исходных файлов нашей программы будет иметь следующий вид:

```
virtualzoo
  |- src
    |- VirtualZoo.as
```

Теперь, когда файл `VirtualZoo.as` создан, мы можем приступать к написанию класса `VirtualZoo`. Однако сначала нужно решить возможную проблему: если выбранное имя основного класса будет конфликтовать (то есть совпадать) с именем одного из предопределенных классов языка ActionScript, то компилятор языка ActionScript не позволит создать этот класс и программа не сможет быть выполнена. Чтобы избежать подобных проблем с именами, воспользуемся пакетами.

Поскольку нам необходимо рассмотреть большой объем материала, мы не будем компилировать код нашей программы вплоть до гл. 7. Если вы все же решите самостоятельно откомпилировать примеры, представленные в гл. 1–6, то, скорее всего, столкнетесь с множеством различных предупреждений и ошибок при компиляции примеров. Внимательное изучение указанных глав позволит вам избежать многих из этих ошибок.

Пакеты

Как видно из названия, *пакет* — это общее определение блока, в который входят группы классов и, как будет сказано ниже, другие элементы программы. Каждый пакет определяет границы независимой физической области программы и присваивает этой области имя, называемое *именем пакета*. Чтобы отличать классы от пакетов, имена пакетов принято записывать со строчной буквы, а имена классов — с прописной.

Когда исходный код класса размещается внутри пакета, имя пакета автоматически становится частью имени класса подобно тому, как ребенок наследует фамилию своих родителей. Например, класс `Player`, размещенный в пакете `game`, получает имя `game.Player`. Обратите внимание, что сначала указывается имя пакета, которое отделяется от имени класса с помощью символа «точка» (.) (термин *«символ»* в среде программистов обозначает букву, цифру, знак препинания и т. д.). Имя пакета помогает отличить класс `game.Player` от других классов с именем `Player`, тем самым предотвращая конфликты имен между различными частями программы или между пользовательскими и предопределенными классами языка ActionScript.

Для создания нового пакета используется *директива описания пакета*. Рассмотрим этот термин. В языке ActionScript все инструкции программы обычно называются *директивами*. *Описания* — это один из типов директив; они создают или описывают, например, пакет или класс. В данном случае описываемым элементом является пакет, откуда и название термина — директива описания пакета.

Если описание создает какой-либо элемент в программе, то говорят, что оно определяет или объявляет этот элемент. Описания часто называют объявлениями.

В общем виде директива описания пакета записывается следующим образом:

```
package имяПакета {
}
```

Все описания пакетов начинаются с ключевого слова: `package`. *Ключевое слово* — это имя команды, зарезервированное для использования в языке ActionScript. В данном случае ключевое слово `package` сообщает компилятору ActionScript о необходимости создания пакета. Сразу после ключевого слова `package` указывается желаемое имя пакета — в предыдущем примере оно заменено выражением *имяПакета* (здесь и далее код, выделенный подобным образом, например *имяПакета*, обозначает текст, который должен быть заменен программистом). Затем с помощью фигурных скобок { и } отмечаются начало и конец содержимого пакета. Чтобы добавить класс в пакет, необходимо записать исходный код класса между фигурными скобками, как показано в следующем примере:

```
package имяПакета {
  Сюда помещается исходный код класса
}
```

С технической точки зрения фигурные скобки в описании пакета являются своего рода оператором, называемым *оператором блока*. Как и описания, операторы относятся к директивам, или, иначе говоря, к базовым инструкциям программы. Оператор блока обозначает начало и конец группы директив, которые должны рассматриваться как логическое целое. Оператор блока описания пакета называется *блоком пакета* или иногда *телом пакета*.

Полный список операторов языка ActionScript указан в гл. 10.

Принято (но вовсе не обязательно) именам пакетов присваивать следующую иерархическую структуру:

- доменное имя организации, которая занимается разработкой программы, записанное в обратном порядке;
- точка (.);
- общее описание содержимого пакета.

Например, пакету, содержащему классы для картографического приложения, разрабатываемого фирмой Acme Corp. (доменное имя acme.com), может быть присвоено имя `com.acme.map`, как показано в следующем примере:

```
package com.acme.map {
}
```

Обратите внимание, что имя домена верхнего уровня `com` предшествует имени домена нижнего уровня `acme` (то есть в имени пакета составляющие доменного имени записываются в обратном порядке).

Доменные имена гарантированно являются уникальными благодаря системе авторизованных регистраторов доменов верхнего уровня. Иными словами, использование доменного имени вашей организации в начале имени пакета позволит избежать конфликтов имен с кодом, разработанным другими организациями.

Теперь попытаемся воспользоваться пакетами в нашей программе создания виртуального зоопарка. Чтобы упростить пример, назовем пакет `zoo`, без указания доменного имени организации. Для описания пакета `zoo` добавим следующий код в файл `VirtualZoo.as`:

```
package zoo {
}
```

После того как мы добавили пакет в файл `VirtualZoo.as`, необходимо изменить расположение файла в файловой системе, чтобы оно соответствовало имени созданного пакета. Вследствие требований, налагаемых всеми компиляторами языка ActionScript корпорации Adobe, исходный файл, содержащий класс (или другое описание) внутри пакета, должен размещаться в структуре папок, соответствующей имени пакета. Например, файл, содержащий пакет с именем `com.gamecompany.zoo`, должен размещаться в папке **zoo**, вложенной в папку **gamecompany**, которая, в свою очередь, вложена в папку **com** (то есть **com/gamecompany/zoo**). Таким образом, мы создадим новую папку с именем **zoo** в файловой структуре нашей программы и перенесем файл `VirtualZoo.as` в эту папку. Файловая структура исходных файлов программы тогда будет выглядеть следующим образом:

```
virtualzoo
  |- src
    |- zoo
      |- VirtualZoo.as
```

Теперь, когда у нас есть описание пакета, добавим в него класс `VirtualZoo`.

Описание класса

Для создания нового класса используется *описание класса*, как показано в следующем обобщенном коде:

```
class Идентификатор {
}
```

Описание класса начинается с ключевого слова `class`, за которым указывается имя класса (в приведенном коде имя класса заменено выражением *Идентификатор*). Термин *«идентификатор»* употребляется в значении «имя». Идентификаторы не должны содержать пробелы или тире и не могут начинаться с цифры. Каждое новое слово

в имени класса принято записывать с прописной буквы, как, например, в именах классов `Date` или `TextField` (`TextField` — это предопределенный класс среды выполнения Flash, экземпляры которого представляют текст, отображаемый на экране).

Фигурные скобки (`{` и `}`), следующие за выражением *Идентификатор* в предыдущем описании класса, являются оператором блока, точно так же, как и в примере описания пакета. Оператор блока описания класса называется *блоком класса* или иногда *телом класса*. Блок класса содержит директивы, описывающие характеристики и поведение класса и его экземпляров.

В следующем примере приводится описание класса `VirtualZoo`, являющегося основным классом для нашей игры-симулятора. Описание класса помещено в тело пакета, который описан в файле `VirtualZoo.as`:

```
package zoo {
  class VirtualZoo {
  }
}
```

Поскольку описание класса `VirtualZoo` находится в пакете `zoo`, полным именем класса (называемым *полностью определенным* именем класса) является `zoo.VirtualZoo`. Тем не менее в тексте мы будем использовать сокращенное, или *неполное*, имя класса — `VirtualZoo`.

Теперь, когда мы описали основной класс нашей программы, создадим еще один класс — `VirtualPet`. С его помощью мы создадим объекты, представляющие зверей в зоопарке.

Как и в случае с классом `VirtualZoo`, мы поместим код класса `VirtualPet` в пакет `zoo`, сохранив его в собственном файле `VirtualPet.as` внутри папки **zoo**. Исходный код из файла `VirtualPet.as` выглядит следующим образом:

```
package zoo {
  class VirtualPet {
  }
}
```

Обратите внимание, что описание пакета может размещаться в нескольких исходных файлах. И хотя классы `VirtualZoo` и `VirtualPet` физически хранятся в разных AS-файлах, они принадлежат одному пакету `zoo`. Любой класс, описание которого принадлежит телу пакета с именем `zoo`, считается частью этого пакета независимо от имени файла размещения. В отличие же от описания пакета, описание класса не может находиться в нескольких файлах и должно полностью размещаться в одном файле.

Модификаторы управления доступом для классов. По умолчанию обращение к классу, входящему в состав определенного пакета, может осуществляться только из кода, принадлежащего тому же пакету. Чтобы класс был доступен для использования за пределами пакета, которому он принадлежит, мы должны описать этот класс с помощью атрибута `public`. Вообще говоря, *атрибуты* определяют порядок использования класса и его экземпляров в программе. Атрибуты указываются перед ключевым словом `class` в описании класса, как показано в приведенном ниже общем примере:

```
атрибут class ИдентификаторКласса {
}
```

Например, чтобы добавить атрибут `public` к классу `VirtualPet`, нужно использовать следующий код:

```
package zoo {
  public class VirtualPet {
  }
}
```

Однако применение атрибута `public` в случае с классом `VirtualPet` необязательно, поскольку класс `VirtualPet` используется только классом `VirtualZoo`, а тот, в свою очередь, может обращаться к классу `VirtualPet` (классы, принадлежащие одному пакету, могут всегда обращаться друг к другу). Таким образом, мы можем вернуться к исходному описанию класса `VirtualPet`, которое косвенным образом позволяет использовать этот класс только внутри пакета `zoo`:

```
package zoo {
  class VirtualPet {
  }
}
```

Если мы хотим явно указать, что класс `VirtualPet` может быть использован только внутри пакета `zoo`, то в описание класса необходимо добавить атрибут `internal`, как показано ниже:

```
package zoo {
  internal class VirtualPet {
  }
}
```

Класс, описанный с помощью атрибута `internal`, может быть использован только внутри пакета, которому он принадлежит. Другим словами, описание класса с помощью атрибута `internal` совершенно не отличается от описания класса без использования каких-либо модификаторов управления доступом. Атрибут `internal` просто служит для однозначного толкования замысла программиста.

Атрибуты `internal` и `public` называются *модификаторами управления доступом*, поскольку управляют порядком доступа к областям внутри программы, в которых допускается использовать данный класс (к ним разрешен доступ данного класса).

В отличие от класса `VirtualPet`, класс `VirtualZoo` должен быть определен с помощью атрибута `public`, поскольку он является основным классом приложения.

Компиляторы, разработанные корпорацией Adobe, требуют, чтобы основной класс приложения был определен с помощью атрибута public.

Следующий код представляет обновленное описание класса `VirtualZoo`, содержащее обязательный атрибут `public`:

```
package zoo {
  public class VirtualZoo {
  }
}
```

Краткий обзор приложения «Зоопарк»

В настоящий момент наша игра состоит из двух классов: `VirtualZoo` (основной класс) и `VirtualPet` (класс, представляющий зверей в виртуальном зоопарке). Оба класса принадлежат пакету `zoo` и хранятся в виде обычных текстовых файлов с именами `VirtualZoo.as` и `VirtualPet.as` соответственно.

Согласно требованию, предъявляемому компиляторами языка ActionScript корпорации Adobe, класс `VirtualZoo` описан с использованием атрибута `public`, поскольку является основным классом приложения. В отличие от класса `VirtualZoo`, класс `VirtualPet` может быть использован только внутри пакета `zoo`, поскольку описан с помощью атрибута `internal`.

В примере листинга 1.1 представлен весь имеющийся к настоящему времени код игры, а также кое-что новое — комментарии к исходному коду.

Комментарий к исходному коду — это примечание, которое предназначено только для программистов и не воспринимается компилятором в процессе компиляции кода.

Комментарии к исходному коду ActionScript бывают двух видов: однострочные, начинающиеся с двух слэшей (`//`), и многострочные, начинающиеся с последовательности `/*` и заканчивающиеся символами `*/`.

Так выглядит однострочный комментарий:

```
// Эта информация только для нас, программистов
```

А так записывается многострочный комментарий:

```
/*
Эта информация
только для нас,
программистов
*/
```

Текущий код для нашей игры выглядит следующим образом.

Листинг 1.1. Игра «Зоопарк»

```
// Содержимое файла VirtualZoo.as
package zoo {
  public class VirtualZoo {
  }
}

// Содержимое файла VirtualPet.as
package zoo {
  internal class VirtualPet {
  }
}
```

Теперь приступим к разработке нашей программы, начав с метода-конструктора основного класса приложения — `VirtualZoo`.

Методы-конструкторы

Метод-конструктор (или сокращенно *конструктор*) — это отдельный набор инструкций, применяемых для инициализации экземпляров класса. Для создания метода-конструктора внутри блока класса помещается *описание функции*, как показано в следующем обобщенном коде:

```
class НекийКласс {
  function НекийКласс ( ) {
  }
}
```

Как видно из приведенного кода, описание метода-конструктора начинается с ключевого слова `function`. Затем следует имя метода-конструктора, которое должно полностью совпадать с именем класса (в том числе и регистр символов!). За именем метода-конструктора следует пара круглых скобок, в которых находится список параметров конструктора (они будут рассмотрены позднее). Фигурные скобки (`{` и `}`), следующие за списком параметров, являются оператором блока — точно такие же операторы блока применяются в описаниях пакета и класса. Оператор блока метода-конструктора называется *телом конструктора*. Тело конструктора содержит директивы, используемые для инициализации экземпляров. Всякий раз, когда создается новый экземпляр класса *НекийКласс*, выполняются директивы, размещенные в теле конструктора (последовательно, сверху вниз). Процесс выполнения директив, размещенных в теле конструктора, называется *выполнением конструктора*.

Методы-конструкторы описываются с использованием ключевого слова function, поскольку с технической точки зрения они являются определенным видом функций. Подробно функции будут рассмотрены в гл. 5.

Если метод-конструктор класса не описан явно, то компилятор языка ActionScript автоматически создает конструктор, не выполняющий никаких действий по инициализации новых экземпляров класса. Несмотря на это удобство, следуя хорошей практике программирования, желательно всегда включать конструктор в описание класса, даже если он не содержит никаких инструкций. Наличие пустого конструктора служит формальным признаком отсутствия конструктора в дизайне класса и создает необходимость включения в описание конструктора соответствующего комментария. Например:

```
class НекийКласс {
  // Пустой конструктор. Для этого класса инициализация не требуется.
  function НекийКласс ( ) {
  }
}
```

В отличие от прав доступа классов, права доступа методов-конструкторов не могут регулироваться при помощи модификаторов управления доступом. В языке ActionScript 3.0 все методы-конструкторы косвенно считаются открытыми (тем не менее, возможно, в будущих версиях языка будет включена поддержка и «за-

крытых» методов-конструкторов). В целях обеспечения однородности стиля в этой книге при описании методов-конструкторов используется модификатор управления доступом `public`, чем подчеркивается тот факт, что все методы-конструкторы должны быть открытыми. Пример использования этого правила приведен в следующем коде:

```
class НекийКласс {
  public function НекийКласс ( ) {
  }
}
```

Причина, по которой в языке ActionScript 3.0 методы-конструкторы должны быть открытыми, обусловлена лимитом времени, выделенным на разработку спецификации языка ECMAScript 4, и непостоянством этой спецификации. Подробную информацию по этому вопросу можно найти в статье Шо Кувамото (Sho Kuwamoto) по адресу http://kuwamoto.org/2006/04/05/as3-on-the-lack-of-private-and-protected-constructors (Шо является руководителем команды разработчиков приложения Adobe Flex Builder).

Метод-конструктор основного класса приложения выполняет особую роль в программе. Он предоставляет возможность выполнения кода сразу после запуска приложения. По существу, метод-конструктор основного класса приложения считается *точкой входа программы*.

Следующий код содержит изменения, связанные с добавлением метода-конструктора в класс `VirtualZoo`:

```
package zoo {
  public class VirtualZoo {
    public function VirtualZoo ( ) {
    }
  }
}
```

Теперь у нашего приложения появилась служебная точка входа. В процессе запуска приложения среда выполнения Flash автоматически создаст экземпляр класса `VirtualZoo` и выполнит его метод-конструктор. Поскольку наше приложение создает виртуальный зоопарк, первое, что необходимо сделать в конструкторе класса `VirtualZoo`, — создать объект класса `VirtualPet` (то есть добавить животное в зоопарк). В следующем разделе мы рассмотрим процедуру создания объектов.

Создание объектов

Для создания объекта из класса (говоря техническим языком, создания экземпляра объекта) в сочетании с именем класса используется ключевое слово `new`. Ниже представлен обобщенный код, позволяющий описать данный подход:

```
new ИмяКласса
```

Например, чтобы создать объект из класса `VirtualPet`, используется следующий код:

```
new VirtualPet
```

Из одного класса можно создать несколько независимых объектов. В следующем примере описан код создания двух объектов `VirtualPet`:

```
new VirtualPet
new VirtualPet
```

Синтаксис констант

Выше было сказано, что для создания нового объекта используется обобщенный синтаксис:

```
new ИмяКласса
```

Этот синтаксис применяется как к предопределенным, так и к пользовательским классам языка ActionScript. Например, следующий код демонстрирует создание нового экземпляра предопределенного класса `Date`, представляющего определенный момент времени:

```
new Date
```

Тем не менее для некоторых собственных классов язык ActionScript также предлагает альтернативный, более удобный способ создания экземпляров, называемый *синтаксисом констант*. К примеру, чтобы создать новый экземпляр класса `Number`, представляющий число с плавающей запятой `25,4`, можно использовать удобную запись в виде константы:

```
25.4
```

Подобным же образом для создания нового экземпляра класса `String`, представляющего текст `"hello"`, можно использовать запись в виде константы:

```
"hello"
```

Наконец, чтобы создать новый экземпляр класса `Boolean`, представляющий логическое состояние `true`, можно также использовать запись в виде константы:

```
true
```

Для создания нового экземпляра класса `Boolean`, представляющего логическое состояние `false`, снова применяется запись в виде константы:

```
false
```

Синтаксис констант также доступен для классов `Object`, `Function`, `RegExp` и `XML`. Синтаксис констант для класса `Object` приведен в гл. 15, для класса `Function` — в гл. 5, а для класса `XML` — в гл. 18. Информацию по синтаксису констант для класса `RegExp` можно найти в соответствующей документации корпорации Adobe.

Пример создания объекта: добавление животного в зоопарк

Теперь, когда нам известно, как создавать объекты, мы можем добавить объект класса `VirtualPet` в нашу программу по созданию виртуального зоопарка. Следующий код делает именно это:

```
package zoo {
  public class VirtualZoo {
    public function VirtualZoo ( ) {
```

```
      new VirtualPet
    }
  }
}
```

Обратите внимание, что в этом коде обращение к классу `VirtualPet` происходит не по его уточненному имени `zoo.VirtualPet`, а по имени `VirtualPet`, являющемуся неуточненным, поскольку код из определенного пакета может обращаться к классам этого пакета по их неуточненным именам.

Однако код не может обращаться к классам в других пакетах. Чтобы получить доступ к открытому классу в другом пакете, необходимо использовать директиву `import`, которая записывается в следующем общем виде:

```
import имяПакета.ИмяКласса;
```

В приведенном коде *имяПакета* — это имя пакета, которому принадлежит класс, а *ИмяКласса* — это имя открытого класса, который должен быть использован. Если указанный класс не является открытым, то его невозможно будет импортировать, поскольку классы, не являющиеся открытыми, не могут быть использованы за пределами пакета, которому принадлежат. Как только класс будет импортирован в программу, к нему можно обращаться по его неуточненному имени. Например, чтобы создать экземпляр предопределенного класса `flash.media.Sound` (который используется для загрузки и воспроизведения звуковых файлов), используется следующий код:

```
import flash.media.Sound
new Sound
```

Если импортировать класс на уровне пакета, то он будет доступен из любого места кода, принадлежащего телу пакета. Например, в следующем коде класс `flash.media.Sound` импортируется на уровне пакета, а затем создается экземпляр класса `Sound` в методе-конструкторе `VirtualZoo`:

```
package zoo {
  import flash.media.Sound

  public class VirtualZoo {
    public function VirtualZoo ( ) {
      new Sound
    }
  }
}
```

В случае возникновения конфликта между неуточненными именами классов для них необходимо использовать уточненные имена. Например, если собственный класс `Sound` описан в пакете `zoo`, обязательным является использование следующего кода, создающего экземпляр предопределенного класса `flash.media.Sound` (обратите внимание на использование уточненного имени):

```
new flash.media.Sound
```

Для создания же экземпляра класса `Sound` из пакета `zoo` нам бы пришлось использовать следующий код:

```
new zoo.Sound
```

Использование неуточненного имени класса (то есть `Sound`) само по себе приводит к ошибке, которая не позволяет откомпилировать программу. Ошибки, которые препятствуют компиляции, называются *ошибками этапа компиляции.*

Чтобы получить доступ ко всем открытым классам в другом пакете, необходимо использовать следующий обобщенный код:

```
import имяПакета.*
```

Например, чтобы получить доступ ко всем открытым классам в пакете `flash.media`, используется такой код:

```
import flash.media.*
```

Обратите внимание, что классы, принадлежащие пакету без имени, помещаются в автоматически создаваемый пакет, который называется *безымянным.* Классы из безымянного пакета можно непосредственно использовать в любом месте кода программы, не применяя директиву `import`. Другими словами:

```
package {
  // Классы, описанные здесь, принадлежат безымянному пакету
  // и могут быть непосредственно использованы
  // в любом месте кода программы
}
```

Тем не менее следует избегать размещения описаний классов в безымянном пакете, поскольку их имена могут конфликтовать с именами других классов (и других типов описаний), описанных в языке ActionScript, других программах или даже в других частях одной программы.

Говоря техническим языком, директива import открывает общедоступное пространство имен указанного пакета для текущей и всех вложенных областей видимости. Если вы новичок в программировании на языке ActionScript, то вам не стоит беспокоиться о техническом аспекте директивы import. Вся необходимая информация будет рассмотрена в следующих главах.

Теперь вернемся к нашей первоначальной задаче — созданию объектов в программе «Зоопарк». Вспомним следующий код, в котором создается новый объект класса `VirtualPet`:

```
package zoo {
  public class VirtualZoo {
    public function VirtualZoo ( ) {
      new VirtualPet
    }
  }
}
```

В приведенном коде успешно создается новый объект класса `VirtualPet`, но при этом возникает проблема: после создания объекта программа не имеет никакой возможности обращаться к нему. В результате она не может использовать новое животное или управлять им. Чтобы предоставить ей такую возможность — обращаться к объекту класса `VirtualPet`, — используются специальные переменные.

Переменные и значения

В языке ActionScript любой объект рассматривается как отдельный, независимый фрагмент данных (или информации), называемый *значением*. Не считая объектов, единственными допустимыми значениями в языке ActionScript являются специальные значения `null` и `undefined`, представляющие понятие «пустое значение». *Переменная* — это идентификатор (то есть имя), ассоциированный со значением. Например, переменной может являться идентификатор `submitBtn`, который ассоциирован с объектом, представляющим кнопку на интерактивной странице в Интернете. Или переменной может быть идентификатор `productDescription`, ассоциированный с объектом `String`, описывающим некий продукт.

Переменные используются для отслеживания информации в программе. Они позволяют обращаться к объекту после его создания.

Существует четыре типа переменных: локальные переменные, переменные экземпляра, динамические переменные экземпляра и статические переменные. Первые два типа будут рассмотрены прямо сейчас, а остальные — далее в этой книге.

Локальные переменные

Локальные переменные применяются для временного отслеживания информации внутри метода-конструктора, метода экземпляра и статического метода или функции. Методы экземпляров и статические методы пока не рассматривались, поэтому сейчас сосредоточимся на использовании локальных переменных в методах-конструкторах.

Для создания локальной переменной внутри метода-конструктора используется описание переменной, как показано в следующем обобщенном коде. Обратите внимание, что описание начинается с ключевого слова `var` и, как и все директивы, не содержащие операторов блока, завершается точкой с запятой. Точка с запятой обозначает конец директивы так же, как точка обозначает конец предложения в обычном языке:

```
class НекийКласс {
  public function НекийКласс ( ) {
    var идентификатор = значение;
  }
}
```

В этом коде *идентификатор* представляет имя локальной переменной, а *значение* — значение, ассоциированное с этой переменной. Знак равенства и элемент *значение* называют *инициализатором переменной*, поскольку они определяют исходное значение переменной.

Процесс ассоциирования переменной со значением называется присваиванием, установкой или записью значения переменной.

Если инициализатор переменной не указан, то компилятор языка ActionScript автоматически присваивает переменной значение по умолчанию, соответствующее ее типу. Эти значения будут рассмотрены в гл. 8.

Локальная переменная может быть использована только внутри того метода или функции, в которой она описана. После завершения выполнения метода или функции срок действия локальной переменной заканчивается и она больше не может быть использована в программе.

Для обращения к объекту класса `VirtualPet`, который был создан ранее в конструкторе класса `VirtualZoo`, создадим локальную переменную. Локальной переменной присвоим имя `pet`, а для связывания объекта `VirtualPet` с этой переменной воспользуемся инициализатором. Привожу код:

```
package zoo {
  public class VirtualZoo {
    public function VirtualZoo ( ) {
      var pet = new VirtualPet;
    }
  }
}
```

Теперь, когда локальная переменная `pet` связана с объектом `VirtualPet`, она может быть использована для обращения к объекту и, следовательно, для управления им. Однако в настоящий момент объект `VirtualPet` не может выполнять никакие действия, поскольку его функциональность еще не запрограммирована. Способы устранения этого недостатка будут рассмотрены в разд. «Параметры и аргументы конструктора», в котором я также расскажу, как предоставить животным возможность иметь имена.

Переменные экземпляра

Ранее было сказано, что класс используется для описания характеристик и поведения объекта определенного типа. В объектно-ориентированном программировании под *характеристикой* понимают определенную часть информации (то есть значение), которая описывает определенный аспект объекта, например ширину, скорость или цвет. Для отслеживания характеристик объекта применяются переменные экземпляра.

Переменная экземпляра — это переменная, принадлежащая определенному объекту. Обычно каждая переменная экземпляра описывает некую характеристику объекта, к которому принадлежит. Например, переменной экземпляра может являться идентификатор `width`, связанный со значением `150`, которое определяет ширину кнопки интерфейса, или идентификатор `shippingAddress`, связанный со значением `ул. Некая, 34`, которое определяет адрес доставки объекта заказанного товара.

Как видно из следующего обобщенного кода, переменные экземпляра создаются с помощью размещаемых непосредственно в описании класса описаний переменных:

```
class НекийКласс {
  var идентификатор = значение;
}
```

Добавление описания переменной экземпляра в описание класса приводит к автоматическому присоединению этой переменной к каждому экземпляру данно-

го класса. Как и в случае с локальными переменными, инициализатор описания переменной экземпляра задает исходное значение для создаваемой переменной. Однако, поскольку в переменных отдельно взятого экземпляра класса хранятся его собственные значения, исходное значение переменной экземпляра зачастую не указывается и присваивается позднее уже при выполнении программы.

В качестве примера добавим переменную экземпляра в класс `VirtualPet`. Она позволит отслеживать имя каждого объекта `VirtualPet`. Переменную экземпляра назовем именем `petName`:

```
package zoo {
  internal class VirtualPet {
    var petName = "Unnamed Pet";
  }
}
```

В результате использования приведенного кода переменная экземпляра `petName` будет автоматически присоединена к каждому новому экземпляру класса `VirtualPet`. Исходным значением переменной `petName` для всех экземпляров класса `VirtualPet` будет являться фраза `Unnamed Pet`. Тем не менее после создания экземпляра класса `VirtualPet` переменной `petName` может быть присвоено новое, индивидуальное значение.

Для присвоения переменной экземпляра нового значения используется следующий обобщенный код:

объект.*переменнаяЭкземпляра* = *значение*

Здесь *объект* — это объект, переменной экземпляра которого присваивается значение; *переменнаяЭкземпляра* — это одна из переменных экземпляра *объект* (описанных в классе объекта); *значение* — это присваиваемое значение.

Воспользуемся описанной методикой, чтобы присвоить какое-либо имя объекту `VirtualPet`, созданному ранее в конструкторе класса `VirtualZoo`. Привожу код описания класса `VirtualZoo`, который содержит все предыдущие изменения:

```
package zoo {
  public class VirtualZoo {
    public function VirtualZoo ( ) {
      var pet = new VirtualPet;
    }
  }
}
```

В соответствии с обобщенным кодом, используемым для присваивания нового значения переменной экземпляра, сначала необходимо обратиться к объекту. В данном случае для обращения к желаемому экземпляру класса `VirtualPet` применим локальную переменную `pet`:

```
pet
```

Затем ставится точка:

```
pet.
```

После этого записывается имя переменной экземпляра, значение которой нужно изменить — в данном случае petName:

```
pet.petName
```

В конце ставится знак = и указывается значение, которое необходимо присвоить переменной экземпляра. В нашем примере применим значение Stan:

```
pet.petName = "Stan"
```

Разве это не мило? Теперь у нашего животного появилась кличка. Мы делаем успехи.

Ниже приведен измененный код описания класса VirtualZoo:

```
package zoo {
  public class VirtualZoo {
    public function VirtualZoo ( ) {
      var pet = new VirtualPet;
      pet.petName = "Stan";
    }
  }
}
```

Обратите внимание, что в предыдущем коде значение переменной экземпляра petName, описанной в классе VirtualPet, присваивается через экземпляр класса VirtualPet, принадлежащего классу VirtualZoo. Следовательно, коду в классе VirtualZoo доступна переменная экземпляра petName. Когда класс разрешает доступ из других классов к своим переменным экземпляра, он позволяет этим классам вносить изменения в характеристики своих экземпляров.

Имя животного — это характеристика, которая естественным образом пригодна для внешнего изменения. Тем не менее некоторые переменные экземпляра могут представлять характеристики, которые не должны изменяться за пределами класса, содержащего описание данных переменных. Например, далее будет создана переменная экземпляра caloriesPerSecond, определяющая скорость, с которой у того или иного животного переваривается пища. Если переменной caloriesPerSecond присвоить слишком маленькое или слишком большое значение, то животное может постоянно хотеть есть или, наоборот, никогда не проголодаться. Поэтому, чтобы внешний код не смог присвоить неприемлемое значение переменной caloriesPerSecond, необходимо ограничить доступ к этой переменной. Для этого применяются модификаторы управления доступом.

Модификаторы управления доступом для переменных экземпляра. *Модификатор управления доступом переменной экземпляра* определяет уровень доступа к этой переменной в программе. Для описаний переменных экземпляра существуют следующие модификаторы управления доступом: public, internal, protected и private.

Модификаторы public и internal для переменных экземпляра обладают таким же эффектом, что и для классов. Переменная экземпляра, объявленная с использованием модификатора public, может быть доступна как внутри, так и снаружи пакета, в котором была описана; переменная экземпляра, объявленная с исполь-

зованием модификатора `internal`, может быть доступна только внутри пакета, в котором была описана.

Модификаторы `protected` и `private` накладывают еще большие ограничения, чем модификатор `internal`. Переменная экземпляра, объявленная с использованием модификатора `protected`, может быть доступна только для кода класса, содержащего описание этой переменной, или для кода потомков этого класса (мы еще не рассматривали наследование, поэтому, если вы незнакомы с объектно-ориентированным программированием, пока не обращайте внимания на этот модификатор). Переменная экземпляра, объявленная с использованием модификатора `private`, может быть доступна только для кода класса, содержащего описание этой переменной. Если при объявлении переменной никакой модификатор не указан, то используется модификатор `internal` (доступ внутри пакета).

Модификаторы управления доступом для переменных экземпляра перечислены в табл. 1.2.

Таблица 1.2. Модификаторы управления доступом переменной экземпляра

	Атрибут			
Размещение кода	**public**	**internal**	**protected**	**private**
Код в классе, содержащем описание переменной	Доступна	Доступна	Доступна	Доступна
Код в потомке класса, содержащего описание переменной	Доступна	Доступна	Доступна	Недоступна
Код в другом классе, принадлежащем пакету с описанием переменной	Доступна	Доступна	Недоступна	Недоступна
Код не принадлежит пакету с описанием переменной	Доступна	Недоступна	Недоступна	Недоступна

Если для описания переменных экземпляра класса использовать модификатор `private`, то информация каждого экземпляра будет полностью закрыта для случайного изменения, что позволит исключить зависимость внешнего кода от внутренней структуры класса и избежать случайного присваивания некорректных значений переменным экземпляра. Как правило, для каждой переменной экземпляра явно указывают модификатор управления доступом.

Использование модификатора `public` для описания переменных экземпляра является нежелательным, если это не заложено в архитектуру класса, однако если вы не уверены в том, какой модификатор управления доступом использовать, то его применение становится наиболее предпочтительным. Впоследствии, если понадобится, уровень доступа к переменной экземпляра может быть легко изменен, тем самым она станет более доступной. Если же переменная экземпляра будет описана с использованием модификатора `public` и внешний код уже использует эту переменную, то изменить модификатор управления доступом на `private` будет достаточно сложно.

В текущей версии нашего приложения, создающего виртуальный зоопарк, к переменной экземпляра `petName` происходит обращение как из класса `VirtualPet`, так и из класса `VirtualZoo`, поэтому мы должны описать переменную `petName`

с использованием модификатора управления доступом `internal`, как показано в следующем коде:

```
package zoo {
  internal class VirtualPet {
    internal var petName = "Unnamed Pet";
  }
}
```

Обратите внимание, что описание переменной экземпляра с использованием атрибута `internal` аналогично описанию переменной без использования какого-либо модификатора управления доступом (поскольку модификатор `internal` применяется по умолчанию).

В оставшейся части книги вы найдете множество примеров использования переменных экземпляра, ознакомиться с которыми можно будет позднее. Пока же вернемся к разработке программы создания виртуального зоопарка.

В настоящий момент структура класса `VirtualPet` допускает возможность присваивания значения переменной `petName` каждого объекта `VirtualPet` по желанию. Однако если необходимо гарантировать, что имя будет присвоено каждому животному, то можно использовать параметры конструктора, которые описываются в следующем разделе.

Параметры и аргументы конструктора

Параметр конструктора — это особый тип локальной переменной, представляющий собой часть описания метода-конструктора. В отличие от обычных локальных переменных, исходное значение параметра конструктора может (или в некоторых случаях должно) задаваться из внешнего кода при создании нового экземпляра класса.

При создании параметров конструктора вместо ключевого слова `var` между круглыми скобками описания функции конструктора просто указывается желаемое имя и инициализатор переменной, как показано в следующем обобщенном коде:

```
class НекийКласс {
  function НекийКласс (идентификатор = значение) {
  }
}
```

В данном коде *идентификатор* — это имя параметра конструктора, а *значение* — исходное значение параметра.

Если возникает необходимость описать несколько параметров в методе-конструкторе, то их имена перечисляются через запятую, как показано в обобщенном коде ниже (обратите внимание на разрывы строк, которые не только допустимы, но и широко распространены):

```
class НекийКласс {
  function НекийКласс (идентификатор1 = значение1,
                       идентификатор2 = значение2,
```

```
                идентификатор3 = значение3) {
  }
}
```

По умолчанию исходное значение параметра конструктора определяется значением, указанным в описании этого параметра. Однако значение параметра конструктора можно определить и при создании объекта, используя следующий обобщенный код:

```
new НекийКласс(значение1, значение2, значение3)
```

В этом коде *значение1*, *значение2* и *значение3* — это значения, присваиваемые в указанном порядке параметрам метода-конструктора класса *НекийКласс*. Значение, присваиваемое параметру конструктора при создании объекта (как показано в предыдущем коде), называется *аргументом конструктора*. Использование аргумента конструктора в качестве значения параметра конструктора называется *передачей* этого значения конструктору.

Если описание параметра конструктора не содержит инициализатора переменной, то исходное значение этого параметра указывается через аргумент конструктора. Такой параметр называется *обязательным параметром конструктора*. Следующий обобщенный код демонстрирует создание класса с одним обязательным параметром конструктора (обратите внимание, что описание параметра не содержит инициализатора переменной):

```
class НекийКласс {
  function НекийКласс (обязательныйПараметр) {
  }
}
```

Любой код, создающий экземпляр предыдущего класса, обязательно должен указывать значение параметра *обязательныйПараметр* с помощью аргумента конструктора, как показано ниже:

```
new НекийКласс(значение)
```

Отсутствие аргумента конструктора для обязательного параметра приведет к ошибке либо на этапе компиляции программы (если для компиляции выбран строгий режим), либо на этапе выполнения программы (если программа была откомпилирована в стандартном режиме). Различия между строгим и стандартным режимами компиляции будут рассмотрены в гл. 7.

При создании нового объекта без аргументов конструктора некоторым программистам нравится оставлять пустые круглые скобки. Например, многие разработчики предпочитают записывать

```
new VirtualPet( )
```

а не

```
new VirtualPet
```

Выбор формата целиком и полностью является вопросом стиля. Язык ActionScript допускает использование обоих этих форматов. Тем не менее в среде программистов на языке ActionScript большее предпочтение отдается первому стилю (со скобками), нежели второму (без скобок). Поэтому начиная с этого момента при создании новых объектов в примерах кода из данной книги будут всегда использоваться круглые скобки, даже при отсутствии аргументов конструктора.

Используя в качестве примера предыдущий обобщенный код, описывающий метод-конструктор класса с параметром, добавим новый метод-конструктор в класс `VirtualPet` и опишем один обязательный параметр конструктора — `name`. Значение параметра `name` будет присвоено переменной экземпляра `petName` каждого объекта `VirtualPet`.

Рассмотрим основной код, содержащий описание нового метода-конструктора без каких-либо инструкций:

```
package zoo {
  internal class VirtualPet {
    internal var petName = "Unnamed Pet";

    public function VirtualPet (name) {
    }
  }
}
```

Поскольку параметр `name` является обязательным, его исходное значение должно определяться извне при создании объекта. В связи с этим мы должны обновить код, создающий объект `VirtualPet` в конструкторе `VirtualZoo`. До этого наш код выглядел так:

```
package zoo {
  public class VirtualZoo {
    public function VirtualZoo ( ) {
      var pet = new VirtualPet;
      pet.petName = "Stan";
    }
  }
}
```

Это же обновленная версия, в которой значение `Stan` передается конструктору класса `VirtualPet`, а не присваивается переменной `petName` созданного экземпляра:

```
package zoo {
  public class VirtualZoo {
    public function VirtualZoo ( ) {
      var pet = new VirtualPet("Stan");
    }
  }
}
```

В этом коде при создании экземпляра класса `VirtualPet` выполняется конструктор этого класса и аргумент конструктора `Stan` присваивается параметру `name`. Из этого следует, что внутри конструктора класса `VirtualPet` параметр `name` можно использовать для присваивания значения `Stan` переменной экземпляра `petName` нового объекта `VirtualPet`. Для этого необходимо указать значение переменной `petName`, используя выражение идентификатора.

Выражения и выражения идентификатора рассматриваются в следующем разделе.

Выражения

Представление значения в исходном коде программы на ActionScript называется *выражением*. Например:

```
new Date( )
```

Здесь `new` — выражение, представляющее новый объект (в данном случае объект `Date`).

Подобным же образом следующий код демонстрирует константное выражение, представляющее объект `Number` со значением `2.5`:

```
2.5
```

Отдельные выражения с помощью операторов могут быть объединены в составное выражение, значение которого вычисляется на этапе выполнения программы. *Оператор* — это встроенная команда, позволяющая объединять, преобразовывать значения (называемые *операндами оператора*) или манипулировать ими. Каждый оператор записывается либо с помощью символа, например +, либо с помощью ключевого слова, например `instanceof`.

К примеру, оператор умножения, используемый для нахождения произведения двух чисел, записывается с помощью символа `*`. Следующий код демонстрирует составное выражение для умножения числа 4 на число 2,5:

```
4 * 2.5
```

При выполнении этого кода вычисляется результат произведения и все составное выражение (`4 * 2.5`) заменяется одним результатом вычисления (`10`). Процесс определения значения выражения называется его *вычислением*.

С полным списком операторов языка ActionScript можно ознакомиться в гл. 10.

Чтобы представить значения, неизвестные при компиляции программы (на этапе компиляции), однако указываемые или вычисляемые при выполнении программы (на этапе выполнения), применяются *имена переменных*. После вычисления программой выражения, содержащего имя переменной, это имя заменяется соответствующим значением переменной. Процесс замены имени переменной ее значением называется *извлечением*, *получением* или *чтением* значения переменной.

Для примера рассмотрим составное выражение, в котором два значения, представленные именами переменных, перемножаются:

```
quantity * price
```

Переменные `quantity` и `price` являются своего рода контейнерами для значений, которые будут определены в процессе выполнения программы. Значение переменной `quantity`, например, может задаваться пользователем, а значение переменной `price` может быть получено из базы данных. Далее предположим, что переменной `quantity` присвоено значение `2`, а переменной `price` — значение `4.99`. При вычислении выражения `quantity * price` программа заменит имя переменной

quantity значением 2, а имя переменной price — значением 4.99. Таким образом, в процессе вычисления это выражение будет заменено следующим:

```
2 * 4.99
```

Окончательным результатом выражения является значение 9.98.

Говоря формальным языком, выражение, состоящее только из одного имени переменной, например quantity, называется выражением идентификатора.

Теперь попробуем применить выражение идентификатора в программе по созданию виртуального зоопарка.

Присваивание одной переменной значения другой переменной

В процессе написания кода программы мы остановились на создании метода-конструктора для класса VirtualPet. Метод-конструктор описывает единственный параметр name, значение которого определяется во внешнем коде, отвечающем за создание объекта в классе VirtualZoo. Ниже представлен исходный код классов VirtualPet и VirtualZoo, включающий в себя все произведенные ранее изменения:

```
// Класс VirtualPet
package zoo {
  internal class VirtualPet {
    internal var petName = "Unnamed Pet";

    public function VirtualPet (name) {
    }
  }
}

// Класс VirtualZoo
package zoo {
  public class VirtualZoo {
    public function VirtualZoo ( ) {
      var pet = new VirtualPet("Stan");
    }
  }
}
```

Теперь, после того как мы ознакомились с процессом применения переменных в выражениях, параметр name может быть использован для присвоения значения Stan переменной экземпляра petName нового объекта VirtualPet.

Как нам уже известно, чтобы присвоить новое значение переменной экземпляра, используется следующий обобщенный код:

объект.переменнаяЭкземпляра = значение

В соответствии с этим кодом, чтобы присвоить значение переменной, сначала необходимо указать объект. В данном случае таким объектом является создаваемый экземпляр класса `VirtualPet`. Для обращения к нему используется ключевое слово `this` — автоматически передаваемый параметр, значением которого является создаваемый объект.

В теле метода-конструктора создаваемый объект называется текущим. Для обращения к нему применяется ключевое слово this.

После ключевого слова `this` ставится точка, а затем указывается имя переменной экземпляра, которой необходимо присвоить значение, — в данном случае `petName`.

```
this.petName
```

После чего ставится знак равенства и указывается значение, которое должно быть присвоено переменной экземпляра:

```
this.petName = value
```

В нашем случае присваиваемым значением является значение параметра `name`. Таким образом, вместо слова `value` просто подставляется имя параметра `name`:

```
this.petName = name
```

На этапе выполнения среда Flash (в которой выполняется программа) заменит параметр `name` из предыдущего кода значением, переданным конструктору класса `VirtualPet`. Это значение впоследствии будет присвоено переменной экземпляра `petName`.

Код класса `VirtualPet` с внесенными изменениями выглядит следующим образом:

```
package zoo {
  internal class VirtualPet {
    internal var petName = "Unnamed Pet";

    public function VirtualPet (name) {
      this.petName = name;
    }
  }
}
```

Теперь, когда значение переменной экземпляра `petName` присваивается в конструкторе класса `VirtualPet`, ненужное исходное значение `Unnamed Pet`, указанное в описании переменной `petName`, может быть удалено. Описание переменной `petName` до настоящего момента выглядело следующим образом:

```
internal var petName = "Unnamed Pet";
```

После изменений описание этой переменной примет следующий вид (обратите внимание на отсутствие инициализатора переменной):

```
package zoo {
  internal class VirtualPet {
    internal var petName;
```

```
    public function VirtualPet (name) {
      this.petName = name;
    }
  }
}
```

Выражение, которое используется для присваивания значения переменной, например this.petName = name, называется выражением присваивания. В качестве знака равенства в таких выражениях применяется оператор, называемый оператором присваивания.

Копии и ссылки. В предыдущем разделе был описан процесс присваивания значения одной переменной другой. В частности, переменной экземпляра `petName` было присвоено значение параметра `name`. Вот этот код:

```
this.petName = name;
```

Результат присваивания значения одной переменной другой зависит от типа присваиваемого значения.

В случаях, когда значением переменной-источника в выражении присваивания является экземпляр класса `String`, `Boolean`, `Number`, `int` или `uint`, среда выполнения создает копию этого значения и присваивает созданную копию целевой переменной. После окончания процедуры присваивания в системной памяти образуются две независимые версии исходного значения — само исходное значение и его копия. Переменная-источник указывает, или ссылается, на первоначальное значение в памяти. Целевая переменная ссылается на новое значение в памяти.

В других случаях, когда значением переменной-источника в выражении присваивания выступает экземпляр пользовательского класса или экземпляр предопределенного класса языка ActionScript, за исключением классов `String`, `Boolean`, `Number`, `int` или `uint`, программа связывает вторую переменную непосредственно со значением первой. После присваивания в памяти будет существовать только одна копия значения, на которую будут ссылаться обе переменные. В подобной ситуации говорят, что переменные совместно используют ссылку на один объект в памяти. Очевидно, что изменения, вносимые в объект через первую переменную, будут доступны и для второй переменной. Например, рассмотрим код, в котором создаются две локальные переменные — `a` и `b`, после чего значение переменной `a` присваивается переменной `b`.

```
var a = new VirtualPet("Stan");
var b = a;
```

В процессе выполнения первой строки приведенного выше кода средой Flash будет создан новый объект класса `VirtualPet`, после этого он будет записан в память и связан с локальной переменной `a`. В результате выполнения второй строки кода объект `VirtualPet`, на который уже ссылается переменная `a`, будет связан с локальной переменной `b`. Изменения, вносимые в объект `VirtualPet` через переменную `a`, будут естественным образом отражаться на переменной `b`, и наоборот. Например, если переменной экземпляра `petName` присвоить новое значение, используя код `b.petName = "Tom"`, а затем обратиться к этой переменной с помощью конструкции `a.petName`, то будет получено то же значение — `Tom`.

Или, если переменной экземпляра petName присвоить значение, используя код a.petName = "Ken", а затем обратиться к данной переменной с помощью конструкции b.petName, будет получено то же значение — Ken.

Переменная, связанная с объектом, не содержит этот объект, а лишь ссылается на него. Сам объект хранится в системной памяти, а его сохранением занимается среда выполнения.

Переменная экземпляра для нашего животного

В одном из предыдущих разделов было сказано, что в тот момент, когда метод или функция, в которой описана локальная переменная, перестает выполняться, данная переменная прекращает свое существование. Чтобы обеспечить доступность созданного экземпляра VirtualPet в классе после выполнения конструктора VirtualZoo, внесем изменения в этот класс. Вместо того чтобы присваивать объект VirtualPet локальной переменной, присвоим этот объект переменной экземпляра pet. Она будет закрытой, поэтому обращаться к ней можно только из кода класса VirtualZoo. Ниже представлен код, отражающий описанные изменения:

```
package zoo {
  public class VirtualZoo {
    private var pet;

    public function VirtualZoo ( ) {
      this.pet = new VirtualPet("Stan");
    }
  }
}
```

На протяжении нескольких предыдущих разделов мы рассматривали вопросы использования переменных экземпляра для наделения объектов характеристиками класса. Теперь сосредоточим свое внимание на использовании методов экземпляра для переноса поведения класса на его объекты.

Методы экземпляра

Метод экземпляра — это отдельный набор инструкций, выполняющих определенную задачу, связанную с данным объектом. По сути, методы экземпляра описывают действия, которые может выполнять тот или иной объект. Например, в предопределенном классе Sound (экземпляры которого представляют звуки в программе) описан метод экземпляра с именем play, который позволяет начать воспроизведение звука. Аналогичным образом в предопределенном классе TextField (экземпляры которого представляют текст на экране) описан метод с именем setSelection, позволяющий изменять количество выделенных символов в текстовом поле.

Для создания метода экземпляра используется описание функции внутри блока класса, как показано в следующем обобщенном коде:

```
class НекийКласс {
  function идентификатор ( ) {
  }
}
```

В данном коде ключевое слово `function` обозначает начало описания метода экземпляра. Затем указывается имя метода экземпляра, которое может являться любым допустимым идентификатором (как уже упоминалось, имена идентификаторов не могут содержать пробелы или тире, а также не должны начинаться с цифры). За именем метода следует пара круглых скобок, содержащих список параметров метода, которые будут рассмотрены позднее. Фигурные скобки { и }, следующие за списком параметров, являются оператором блока. Оператор блока метода экземпляра называется *телом метода*. Оно содержит директивы, используемые для выполнения определенной задачи.

Поскольку методы с технической точки зрения являются определенным видом функций, при описании методов экземпляра используется ключевое слово function. Подробно функции рассматриваются в гл. 5.

Для выполнения кода, описанного в теле определенного метода, используется *выражение вызова*, как показано в следующем обобщенном коде. Обратите внимание на обязательное использование скобок ( ), следующих за именем метода.

```
объект.имяМетода( )
```

В этом коде *имяМетода* — это имя метода, код которого должен быть выполнен, а *объект* — ссылка на определенный экземпляр, который будет использоваться для выполнения задачи, представленной указанным методом. Использование выражения вызова для выполнения кода, описанного в теле метода экземпляра, называется *вызовом метода объекта* (или *вызовом метода через объект*). Кроме того, применяется термин *«активизация»*, обозначающий *вызов*.

При упоминании имени определенного метода в большинстве документации используется оператор круглых скобок ( ). Например, в обычной документации чаще пишут setSelection( ), а не просто setSelection. Использование оператора скобок помогает различать в тексте имена методов и переменных.

Теперь реализуем представленные концепции в нашей программе создания виртуального зоопарка.

Чтобы наделить наших животных способностью питаться, добавим новую переменную экземпляра и новый метод экземпляра в класс `VirtualPet`. Новая переменная экземпляра — `currentCalories` — будет отслеживать количество пищи, съеденной каждым животным, в виде числового значения. В новом методе экземпляра `eat( )` будет реализована концепция принятия пищи путем добавления 100 калорий к текущему значению переменной экземпляра `currentCalories`. В конечном счете метод `eat( )` будет вызываться в ответ на действие пользователя — кормление животного.

Следующий код демонстрирует описание переменной `currentCalories`. Для исключения возможности влияния внешнего кода на количество калорий, которым обладает каждый экземпляр `VirtualPet`, переменная `currentCalories` описана с использованием модификатора `private`. Обратите внимание, что каждому новому экземпляру `VirtualPet` изначально присваивается 1000 калорий.

```
package zoo {
  internal class VirtualPet {
    internal var petName;
    private var currentCalories = 1000;

    public function VirtualPet (name) {
      this.petName = name;
    }
  }
}
```

Следующий код демонстрирует описание метода экземпляра `eat ( )`, тело которого пока не содержит никаких инструкций. Обратите внимание, что описания методов экземпляра размещаются после описания метода-конструктора класса, а описания переменных экземпляра — до описания метода-конструктора класса.

```
package zoo {
  internal class VirtualPet {
    internal var petName;
    private var currentCalories = 1000;

    public function VirtualPet (name) {
      this.petName = name;
    }

    function eat ( ) {
    }
  }
}
```

Хотя тело метода `eat ( )` пока не содержит никаких инструкций, подобного описания вполне достаточно, чтобы вызвать метод `eat ( )` для объекта `VirtualPet`, как показано в следующей обновленной версии класса `VirtualZoo`:

```
package zoo {
  public class VirtualZoo {
    private var pet;

    public function VirtualZoo ( ) {
      this.pet = new VirtualPet("Stan");
      // Вызов метода eat( ) для объекта VirtualPet, ссылка на который
      // хранится в переменной pet
      this.pet.eat( );
    }
  }
```

Предположим, что в теле метода `eat( )` к значению переменной `currentCalories` того объекта, через который был вызван метод `eat( )`, необходимо добавить 100. Для обращения к этому объекту используется ключевое слово `this`.

В теле метода экземпляра объект, через который был вызван данный метод, называется текущим объектом. Для обращения к текущему объекту используется ключевое слово this. Обратите внимание, что понятие «текущий объект» может применяться как к объекту, создаваемому в методе-конструкторе, так и к объекту, через который был вызван метод экземпляра.

Добавление числового значения (например, 100) к значению существующей переменной (например, `currentCalories`) включает в себя два этапа. Сначала вычисляется результат сложения значения переменной и числового значения, а затем полученная сумма присваивается переменной. Обобщенный код выглядит следующим образом:

некаяПеременная = некаяПеременная + числовоеЗначение

В случае с методом `eat( )` к значению переменной `currentCalories` текущего объекта (`this`) мы собираемся добавить `100`. В результате получаем следующий код:

```
this.currentCalories = this.currentCalories + 100;
```

В качестве удобной альтернативы предыдущему коду язык ActionScript предлагает *оператор сложения с присваиванием* +=, который, если используется с числами, прибавляет значение, находящееся справа от оператора, к переменной, находящейся слева от него, как показано в следующем коде:

```
this.currentCalories += 100;
```

Код класса `VirtualPet` теперь выглядит следующим образом:

```
package zoo {
  internal class VirtualPet {
    internal var petName;
    private var currentCalories = 1000;

    public function VirtualPet (name) {
      this.petName = name;
    }

    function eat ( ) {
      this.currentCalories += 100;
    }
  }
}
```

Начиная с текущего момента, при каждом вызове метода `eat( )` экземпляра `VirtualPet` значение переменной `currentCalories` данного экземпляра будет увеличиваться на 100. Например, следующий код, взятый из конструктора класса `VirtualZoo`, увеличивает значение переменной `currentCalories` экземпляра `VirtualPet`, ссылка на который хранится в переменной `pet`, до 1100 (поскольку

всем экземплярам VirtualPet изначально присваивается значение 1000 калорий).

```
this.pet = new VirtualPet("Stan");
this.pet.eat( );
```

Обратите внимание, что, несмотря на закрытость характеристики currentCalories класса VirtualPet, значение этой переменной может быть изменено в результате выполнения действия (принятия пищи), инициированного внешним кодом — экземпляром VirtualPet. Однако в некоторых случаях даже методы экземпляра должны быть закрытыми. Как и в случае с переменными экземпляра, чтобы контролировать уровень доступа методов экземпляра в программе, используются модификаторы управления доступом.

Модификаторы управления доступом для методов экземпляра

Для описаний методов экземпляра применяются те же модификаторы управления доступом, что и для переменных экземпляра: public, internal, protected и private. Доступ к методу экземпляра, объявленному с использованием модификатора public, может быть осуществлен как внешними командами по отношению к пакету, в котором метод создан, так и внутренними. Метод экземпляра, объявленный с использованием модификатора internal, доступен только для внутренних команд пакета, в котором он описан. Метод экземпляра, объявленный с использованием модификатора protected, может быть доступен только для кода класса, содержащего описание этого метода, или для кода потомков этого класса (мы еще не рассматривали наследование, поэтому, если вы незнакомы с объектно-ориентированным программированием, не обращайте внимания на этот модификатор). Метод экземпляра, объявленный с использованием модификатора private, может быть доступен только для кода класса, содержащего описание этого метода. В ситуации, когда при описании метода ни один из модификаторов не был указан, применяется модификатор internal (доступ внутри пакета).

Использование модификаторов управления доступом при описании методов класса позволяет реализовать на практике принцип «черного ящика». В объектно-ориентированном программировании каждый объект рассматривается как черный ящик, управляемый с помощью набора внешних смоделированных кнопок. Человек, использующий эти кнопки, ничего не знает (и, впрочем, для него это неважно) о действиях, происходящих внутри объекта, — его интересует только то, чтобы объект выполнил желаемое действие. Открытые методы экземпляра класса являются теми самыми кнопками, с помощью которых программист может заставить объект выполнить определенную операцию. Закрытые методы экземпляра класса используются для выполнения других внутренних операций. Таким образом, чтобы заставить экземпляры данного класса выполнять определенные действия, в описании класса должны быть открыты только те методы, которые требуются внешнему коду. Методы, предназначенные для выполнения внутренних операций, должны быть описаны с использованием модификаторов private, protected или internal.

В качестве аналогии представьте себе, что объект — это автомобиль, водителем которого является программист, использующий объект, а производитель — это программист, создавший класс объекта. Чтобы управлять автомобилем, водителю совершенно не обязательно знать, как устроен двигатель. Он просто использует педаль газа, чтобы набирать скорость, и рулевое колесо, чтобы поворачивать. Задача ускорения автомобиля в ответ на нажатие педали газа решается производителем автомобиля, но никак не водителем.

При разработке собственных классов вопросам удобства их использования должно уделяться не меньше внимания, чем вопросам их внутренней реализации. Не забывайте регулярно ставить себя на место «водителя». В идеальном случае всякий раз, когда вносятся изменения во внутреннюю реализацию класса, открытые методы класса, используемые внешним кодом, должны изменяться незначительно или не изменяться совсем. Если на автомобиль устанавливается новый двигатель, то у водителя по-прежнему должна сохраняться возможность пользоваться педалью газа. По мере возможности изменяемые части классов необходимо держать «за кулисами», в закрытых методах.

В терминах объектно-ориентированного программирования открытые методы с открытыми переменными экземпляра класса иногда называются внешним интерфейсом класса, или API класса (Application Programming Interface — программный интерфейс приложения). Термин API также используется для обозначения общих функций, предоставляемых целой группой классов. Например, предопределенные классы среды выполнения Flash, отвечающие за отображение содержимого на экране, называются экранным API. Подобным образом набор пользовательских классов, используемых для визуализации трехмерных объектов, может называться 3D API. Помимо классов в состав программных интерфейсов могут входить и другие программные описания (например, переменные и функции).

В языке ActionScript термин «интерфейс» имеет дополнительное техническое значение, которое мы рассмотрим в гл. 9. Во избежание путаницы для описания открытых методов и переменных экземпляра класса термин «интерфейс» в этой книге не применяется.

Теперь, возвращаясь к программе по созданию виртуального зоопарка, добавим модификатор управления доступом в описание метода `eat( )` класса `VirtualPet`. Поскольку метод `eat( )` является одним из служебных средств, с помощью которых внешний код может управлять объектами `VirtualPet`, он будет реализован как открытый. Привожу измененный код:

```
package zoo {
  internal class VirtualPet {
    internal var petName;
    private var currentCalories = 1000;

    public function VirtualPet (name) {
      this.petName = name;
    }

    public function eat ( ) {
      this.currentCalories += 100;
    }
  }
}
```

В таком виде метод `eat( )` класса `VirtualPet` является жестким, поскольку всякий раз при вызове этого метода к значению переменной `currentCalories` прибавляется одно и то же количество калорий. В конечном счете необходимо добиться, чтобы количество добавляемых калорий изменялось динамически в зависимости от типа пищи, которую предлагает пользователь. Чтобы предоставить внешнему коду возможность указывать количество добавляемых в процессе кормления калорий при вызове метода `eat( )`, придется воспользоваться параметрами метода.

Параметры и аргументы метода

Как и в случае с параметрами конструктора, *параметр метода* — это особый тип локальной переменной, представляющий собой часть описания метода, но, в отличие от локальных переменных, исходное значение параметра метода может (или в некоторых случаях должно) задаваться из внешнего кода.

Для описания параметра метода применяется следующий обобщенный код (обратите внимание, что описание параметров метода имеет такую же структуру, как и при описании параметров конструктора):

```
function имяМетода (идентификатор1 = значение1,
                    идентификатор2 = значение2,
                    ...
                    идентификаторn = значениеn) {
}
```

В приведенном выше коде *идентификатор1 = значение1, идентификатор2 = значение2, ... идентификаторn = значениеn* — список имен параметров метода и их соответствующих исходных значений. По умолчанию исходным значением параметра метода является значение, указанное в описании этого параметра. Тем не менее значение параметра метода может быть дополнительно указано в выражении вызова, как показано в следующем обобщенном коде:

```
имяМетода(значение1, значение2... значениеn)
```

В данном коде *имяМетода* — это имя вызываемого метода, а *значение1, значение2... valuen* — это список значений, которые по порядку присваиваются параметрам метода *имяМетода*. Значение параметра метода, указанное через выражение вызова (как показано в предыдущем коде), называется *аргументом метода*. Использование аргумента метода для задания значения параметра метода называется *передачей* этого значения в метод.

Как и в случае с параметрами конструктора, исходное значение параметра метода, если его описание не содержит инициализатора переменной, должно быть указано через аргумент метода этого параметра. Такой параметр называется *обязательным параметром метода*.

Следующий обобщенный код демонстрирует описание метода с одним обязательным параметром (обратите внимание, что описание параметра не содержит инициализатора переменной):

```
function имяМетода (обязательныйПараметр) {
}
```

Любой код, вызывающий предыдущий метод, обязательно должен указывать значение параметра *обязательныйПараметр* с помощью аргумента метода, как показано в следующем обобщенном коде:

имяМетода(значение)

Отсутствие аргумента метода для обязательного параметра неизбежно приведет к ошибке либо на этапе компиляции программы (если для компиляции программы выбран строгий режим), либо на этапе ее выполнения (если программа была откомпилирована в стандартном режиме).

Теперь обновим описание метода `eat( )` класса `VirtualPet`, включив обязательный параметр `numberOfCalories`. Вызов метода `eat( )` всякий раз будет обеспечивать увеличение значения переменной `currentCalories` текущего объекта на значение параметра `numberOfCalories`. Привожу обновленный код метода `eat( )`:

```
package zoo {
  internal class VirtualPet {
    internal var petName;
    private var currentCalories = 1000;

    public function VirtualPet (name) {
      this.petName = name;
    }

    public function eat (numberOfCalories) {
      this.currentCalories += numberOfCalories;
    }
  }
}
```

Поскольку параметр `numberOfCalories` является обязательным, его исходное значение должно указываться во внешнем коде при вызове метода `eat( )`. Попробуем реализовать это требование для объекта `VirtualPet`, создаваемого в конструкторе `VirtualZoo`. До этого момента код конструктора `VirtualZoo` выглядел следующим образом:

```
package zoo {
  public class VirtualZoo {
    private var pet;

    public function VirtualZoo ( ) {
      this.pet = new VirtualPet("Stan");
      this.pet.eat( );
    }
  }
}
```

Обновленная версия кода, где в метод `eat( )` передается значение `50`, будет выглядеть так:

```
package zoo {
  public class VirtualZoo {
    private var pet;
```

```
    public function VirtualZoo ( ) {
      this.pet = new VirtualPet("Stan");
      this.pet.eat(50);
    }
  }
}
```

Поскольку выражение вызова из данного кода присваивает значение `50` параметру `numberOfCalories` метода `eat ( )`, его выполнение увеличивает значение переменной `currentCalories` экземпляра `VirtualPet`, на который ссылается переменная `pet`, на `50`. Это значит, что после выполнения кода конструктора значение переменной `currentCalories` экземпляра, на который ссылается переменная `pet`, будет равно `1050`.

Возвращаемые значения метода

Подобно тому, как методы могут принимать значения в виде аргументов, они также могут генерировать возвращаемые значения. Для возврата значения из метода используется *оператор возврата*, как показано в следующем обобщенном коде:

```
function имяМетода ( ) {
  return значение;
}
```

Значение, возвращаемое методом, называется возвращаемым значением, или результатом, метода.

После выполнения метода его возвращаемое значение становится значением выражения вызова, с помощью которого был вызван этот метод.

Чтобы продемонстрировать использование возвращаемых значений метода, добавим новый метод в класс `VirtualPet`, который позволит определить возраст животного и вернуть получившийся результат. Для определения возраста животного нам понадобятся базовые знания о классе `Date`, экземпляры которого представляют определенные моменты времени. Для создания нового экземпляра класса `Date` используется следующий код:

```
new Date( )
```

Для внутреннего представления времени в экземплярах класса `Date` используется «количество миллисекунд до или после полуночи 1 января 1970 года». Например, время «одна секунда после полуночи 1 января 1970 года» выражается числом 1000. Подобным образом, время «полночь 2 января 1970 года» выражается числом 86 400 000 (один день — это 1000 мс × 60 с × 60 мин × 24 ч). По умолчанию новый объект `Date` представляет текущее время на локальной системе.

Для обращения к числовому значению «миллисекунд с 1970 года» определенного объекта `Date` используется переменная экземпляра `time`. Например, в следующем коде создается новый экземпляр класса `Date` и затем возвращается значение его переменной `time`:

```
new Date( ).time;
```

В результате выполнения этого кода 24 января 2007 года в 17:20 было получено значение 1 169 677 183 875, представляющее точное количество миллисекунд между полночью 1 января 1970 года и временем выполнения кода (то есть 17:20 24 января 2007 года).

Теперь вернемся к классу VirtualPet. Чтобы иметь возможность определять возраст объектов VirtualPet, необходимо сохранить точное время создания каждого такого объекта. Для сохранения времени создания каждого объекта создадим экземпляр предопределенного класса Date в конструкторе класса VirtualPet и присвоим этот экземпляр переменной creationTime экземпляра класса VirtualPet. Привожу этот код:

```
package zoo {
  internal class VirtualPet {
    internal var petName;
    private var currentCalories = 1000;
    private var creationTime;

    public function VirtualPet (name) {
      this.creationTime = new Date( );
      this.petName = name;
    }

    public function eat (numberOfCalories) {
      this.currentCalories += numberOfCalories;
    }
  }
}
```

Использование переменной creationTime позволяет определить возраст любого объекта VirtualPet путем вычитания времени создания объекта из текущего времени. Это вычисление производится с помощью нового метода getAge( ):

```
public function getAge ( ) {
  var currentTime = new Date( );
  var age = currentTime.time - this.creationTime.time;
}
```

Чтобы возвратить вычисленный возраст объекта, воспользуемся оператором возврата:

```
public function getAge ( ) {
  var currentTime = new Date( );
  var age = currentTime.time - this.creationTime.time;

  return age;
}
```

Следующий код демонстрирует описание метода getAge( ) в контексте описания класса VirtualPet:

```
package zoo {
  internal class VirtualPet {
    internal var petName;
```

```
    private var currentCalories = 1000;
    private var creationTime;

    public function VirtualPet (name) {
      this.creationTime = new Date( );
      this.petName = name;
    }

    public function eat (numberOfCalories) {
      this.currentCalories += numberOfCalories;
    }

    public function getAge ( ) {
      var currentTime = new Date( );
      var age = currentTime.time - this.creationTime.time;
      return age;
    }
  }
}
```

Теперь воспользуемся возвращенным значением метода `getAge ( )` в конструкторе класса `VirtualZoo`. Выражение вызова метода `getAge ( )` рассмотрим в следующей обновленной версии конструктора `VirtualZoo`:

```
package zoo {
  public class VirtualZoo {
    private var pet;

    public function VirtualZoo ( ) {
      this.pet = new VirtualPet("Stan");
      this.pet.getAge( );
    }
  }
}
```

В приведенном коде выражение `pet.getAge ( )` возвращает числовое значение, представляющее количество миллисекунд, прошедших с момента создания объекта `VirtualPet`, на который ссылается переменная `pet`. Чтобы впоследствии можно было обращаться к этому значению в программе, его необходимо присвоить переменной, как показано в следующем коде:

```
package zoo {
  public class VirtualZoo {
    private var pet;

    public function VirtualZoo ( ) {
      this.pet = new VirtualPet("Stan");
      var age = this.pet.getAge( );
    }
  }
}
```

Позже, в более полной версии программы по созданию виртуального зоопарка, возраст будет выводиться на экран.

Возвращаемые значения методов являются широко используемой составляющей объектно-ориентированного программирования. В книге будут постоянно описываться возвращаемые значения. Точно так же вы постоянно будете применять их при разработке собственных классов.

Обратите внимание, что выражение вызова можно объединять с остальными выражениями с помощью операторов. Например, в следующем коде для вычисления половины возраста животного используется оператор деления:

```
pet.getAge( ) / 2
```

Подобным же образом в следующем коде создаются два объекта класса `VirtualPet`, выполняется сложение возрастов созданных объектов, и затем полученная сумма присваивается локальной переменной `totalAge`.

```
package zoo {
  public class VirtualZoo {
    private var pet1:
    private var pet2:

    public function VirtualZoo ( ) {
      this.pet1 = new VirtualPet("Sarah");
      this.pet2 = new VirtualPet("Lois");
      var totalAge = this.pet1.getAge( ) + this.pet2.getAge( );
    }
  }
}
```

Обратите внимание, что, когда оператор возврата не возвращает никакого значения, он просто завершает выполнение текущего метода. Например:

```
public function некийМетод ( ) {
  // Код, размещенный здесь (перед оператором возврата), будет выполнен

  return;

  // Код, размещенный здесь (после оператора возврата), выполнен не будет
}
```

Значением выражения вызова, используемого для вызова метода, который не имеет возвращаемого значения (или вообще не имеет оператора возврата), является специальное значение `undefined`. Операторы возврата, не возвращающие никакого значения, обычно используются для завершения методов на основании некоторого условия.

Подписи методов

Иногда в документации и при обсуждении вопросов, связанных с объектно-ориентированным программированием, встречается понятие *«подпись метода»*, обозначающее совокупность имени метода и списка его параметров. В языке ActionScript подпись метода также включает в себя тип данных каждого параметра и тип возвращаемого методом значения. Типы данных параметров и типы возвращаемых значений рассматриваются в гл. 8.

Например, подпись для метода `eat( )` является:

```
eat(numberOfCalories)
```

А подпись метода `getAge( )` представляет собой просто:

```
getAge( )
```

На этом закончим рассмотрение основных вопросов, связанных с методами экземпляров. Прежде чем завершить эту главу, рассмотрим последний вопрос, относящийся к терминам языка ActionScript.

Члены и свойства

В терминологии языка ActionScript 3.0 совокупность переменных и методов объекта называется *свойствами* объекта, где *свойство* означает «имя, связанное со значением или методом».

В некоторых справочниках по языку ActionScript (в основном это касается справочников от корпорации Adobe) понятие *«свойство»* также используется для обозначения переменной экземпляра. Для того чтобы избежать путаницы, вызванной данным противоречием, в настоящей книге понятие «свойство» употребляться не будет.

В случае необходимости будет использоваться традиционное понятие объектно-ориентированного программирования — *члены экземпляра* (или просто *члены*), — обозначающее совокупность методов и переменных экземпляра. К примеру, можно сказать, что «`radius` не является членом класса `Box`», подразумевая, что в классе `Box` не описаны методы или переменные с именем `radius`.

Обзор программы по созданию виртуального зоопарка

В этой главе было введено большое количество новых концепций и понятий. Теперь попрактикуемся в их использовании, проанализировав программу «Зоопарк» последний раз в этой главе.

Приложение, имитирующее зоопарк, состоит из двух классов: `VirtualZoo` (основной класс) и `VirtualPet` (класс, представляющий животных в зоопарке).

Сразу же после запуска нашего приложения экземпляр класса `VirtualZoo` автоматически создается средой выполнения Flash (поскольку класс `VirtualZoo` является основным классом приложения). В результате создания экземпляра класса `VirtualZoo` выполняется метод-конструктор `VirtualZoo`. Метод-конструктор `VirtualZoo` создает экземпляр класса `VirtualPet`, передавая в качестве единственного аргумента конструктора значение `Stan`.

В рассматриваемом классе `VirtualPet` описаны три переменных экземпляра: `petName`, `currentCalories` и `creationTime`. Эти переменные экземпляра

определяют кличку, количество пищи в желудке и дату рождения каждого животного.

С помощью константного выражения каждому новому объекту `VirtualPet` в качестве исходного значения переменной `currentCalories` присваивается `1000`. Исходным значением переменной `creationTime` является объект класса `Date`, указывающий время создания объекта `VirtualPet`. При создании объекта `VirtualPet` переменной `petName` присваивается значение обязательного параметра конструктора `name`. Параметр конструктора `name` получает свое значение через аргумент конструктора, который указывается в выражении `new`, используемом для создания объекта `VirtualPet`.

В классе `VirtualPet` описаны два метода экземпляра: `eat( )` и `getAge( )`. Метод `eat( )` увеличивает значение переменной `currentCalories` на указанную величину. Метод `getAge( )` вычисляет и возвращает возраст животного в миллисекундах.

Текущая версия программы по созданию виртуального зоопарка представлена в листинге 1.2.

Листинг 1.2. Программа «Зоопарк»

```
// Класс VirtualPet
package zoo {
  internal class VirtualPet {
    internal var petName;
    private var currentCalories = 1000;
    private var creationTime;

    public function VirtualPet (name) {
      this.creationTime = new Date( );
      this.petName = name;
    }

    public function eat (numberOfCalories) {
      this.currentCalories += numberOfCalories;
    }

    public function getAge ( ) {
      var currentTime = new Date( );
      var age = currentTime.time - this.creationTime.time;
      return age;
    }
  }
}

// Класс VirtualZoo
package zoo {
  public class VirtualZoo {
    private var pet;
```

```
    public function VirtualZoo ( ) {
      this.pet = new VirtualPet("Stan");
    }
  }
}
```

В этой главе мы достигли больших успехов. Тем не менее еще многое предстоит узнать. Когда будете готовы приступить к дальнейшему изучению основ языка ActionScript 3.0, переходите к следующей главе.

Условные операторы и циклы

В этой главе мы отвлечемся от общих тем, касающихся классов и объектов. Вместо этого мы сосредоточимся на двух важнейших типах инструкций: *условных операторах* и *циклах*. Условные операторы используются для добавления логики в программу, а циклы применяются для выполнения повторяющихся задач. И условные операторы и циклы являются чрезвычайно распространенными, и их можно увидеть практически в каждой программе, написанной на языке ActionScript. Как только мы рассмотрим условные операторы и циклы, мы вернемся к изучению классов и объектов и продолжим разработку нашей программы по созданию виртуального зоопарка.

Все примеры кода, представленные в этой главе, не принадлежат какому-либо функционирующему классу или программе. Тем не менее в реальной программе условные операторы и циклы могут использоваться внутри методов экземпляра, методов-конструкторов, статических методов, функций, непосредственно в теле класса или пакета и даже за пределами тела пакета.

Условные операторы

Условный оператор — это такой тип оператора, который исполняется только при выполнении определенного условия. Условные операторы позволяют программе выбирать один из нескольких возможных путей дальнейшего исполнения в зависимости от ее текущего состояния.

В языке ActionScript существует два различных условных оператора: оператор `if` и оператор `switch`. Кроме того, в языке ActionScript есть и простой условный оператор `?:`, который кратко рассматривается в гл. 10. Подробную информацию об операторе `?:` можно найти в справочнике по языку ActionScript корпорации Adobe.

Оператор if

Оператор `if` напоминает развилку на дороге. Он содержит два блока кода и выражение (называемое *условным выражением*), которое определяет блок кода для дальнейшего выполнения. Для создания оператора `if` применяется следующий обобщенный код:

```
if (условноеВыражение) {
  блокКода1
} else {
  блокКода2
}
```

Когда при выполнении программы на языке ActionScript встречается оператор `if`, среда выполнения Flash выполняет либо инструкции *блокКода1*, либо инструкции *блокКода2* в зависимости от значения выражения *условноеВыражение*. Если результатом выражения *условноеВыражение* является значение `true` типа `Boolean`, то выполняется первый блок кода. Если результатом выражения *условноеВыражение* является значение `false` типа `Boolean`, то — второй блок кода. Если результатом выражения *условноеВыражение* является значение другого типа, отличного от `Boolean`, то среда выполнения Flash автоматически преобразует результат выражения *условноеВыражение* в объект типа `Boolean` и использует результат преобразования при выборе блока кода для исполнения.

Правила преобразования значений различных типов в объект типа Boolean описаны в табл. 8.5.

Например, в следующем операторе `if` результатом указанного условного выражения является значение `true` типа `Boolean`, поэтому переменной `greeting` присваивается значение `Hello`, а не `Bonjour`.

```
var greeting;

// Результатом условного выражения является значение true, поэтому...
if (true) {
  // ...выполняется этот код
  greeting = "Hello";
} else {
  // Этот код не выполняется
  greeting = "Bonjour";
}
```

Конечно, условное выражение, использованное в предыдущем примере, в реальной программе применялось бы крайне редко, если вообще применялось бы, поскольку его результатом всегда является одно и то же значение. В подавляющем большинстве классов результат условного выражения определяется динамически в процессе выполнения на основании информации, вычисляемой программой или вводимой пользователем.

Например, предположим, что мы создаем общедоступный сайт, один из разделов которого посвящен азартным играм. Играть в них могут пользователи, возраст которых не менее 18 лет. Во время регистрации на сайте статус каждого пользователя загружается из базы данных. Загруженный статус присваивается переменной `gamblingAuthorized`. Если значением этой переменной является `true`, то возраст пользователя составляет 18 лет или более; значение `false` означает, что пользователю менее 18 лет.

Когда пользователь пытается войти в раздел с азартными играми, приложение использует следующий условный оператор, чтобы определить, можно ли предоставить доступ к этому разделу:

```
if (gamblingAuthorized) {
  // Расположенный здесь код отображает интерфейс раздела с азартными играми
} else {
```

```
  // Представленный здесь код отображает сообщение
  // "Доступ запрещен"
}
```

Зачастую условным выражением оператора `if` является либо *выражение равенства*, либо *выражение отношения*. Для сравнения двух значений и представления результата этого сравнения в виде значения типа `Boolean` (то есть либо `true`, либо `false`) в выражениях равенства и выражениях отношения используются *операторы равенства* и *операторы отношения*. Например, в следующем выражении равенства используется *оператор равенства* (==) для сравнения выражения `Mike` с выражением `Margaret`:

```
"Mike" == "Margaret"
```

Результатом предыдущего выражения является значение `false` типа `Boolean`, поскольку выражение `Mike` не равно выражению `Margaret`.

Подобным образом для сравнения значения `6` со значением `7` в следующем выражении отношения применяется *оператор «меньше чем»* (<):

```
6 < 7
```

Результатом этого выражения является значение `true` типа `Boolean`, поскольку 6 меньше 7.

Как видно из предыдущих примеров, экземпляры класса `String` сравниваются по отдельным символам, а при сравнении экземпляров классов `Number`, `int` и `uint` сравниваются математические величины, хранящиеся в этих экземплярах. Обратите внимание, что при сравнении строк учитывается регистр, например выражение `a` не равно выражению `A`. Правила, используемые при сравнении значений (в каких случаях два значения равны между собой или одно значение больше или меньше другого), можно найти в описании операторов ==, ===, < и > в справочнике по языку ActionScript корпорации Adobe.

Теперь рассмотрим пример оператора `if`, в качестве условного выражения которого используется знак равенства. Предположим, что мы создаем программу для интернет-магазина с виртуальной корзиной для покупок. В программе создана переменная экземпляра `numItems`, отражающая текущее количество товаров в корзине пользователя. Если корзина пуста, то программа выдает сообщение **Ваша корзина пуста**. В ином случае программа выдает сообщение **Количество товаров в вашей корзине: n** (где *n* обозначает количество товаров в корзине).

В следующем примере кода показано, как в программе может быть создано сообщение о текущем статусе корзины пользователя. Присваиваемое значение переменной `basketStatus` зависит от значения переменной `numItems`.

```
var basketStatus;

if (numItems == 0) {
  basketStatus = "Ваша корзина пуста";
} else {
  basketStatus = "Количество товаров в вашей корзине: " + numItems;
}
```

Если значение переменной numItems в предыдущем примере кода равно нулю, то программа присваивает переменной basketStatus следующее выражение:

```
"Ваша корзина пуста"
```

В противном случае программа присваивает переменной basketStatus следующее выражение:

```
"Количество товаров в вашей корзине: " + numItems
```

Обратите внимание на использование *оператора конкатенации* (+) в предыдущем выражении. Он преобразует числовое значение, хранящееся в переменной numItems, в строку и объединяет ее со строкой "Количество товаров в вашей корзине:". Результирующим значением станет объединение двух выражений. Например, если значение переменной numItems равно 2, то результатом операции конкатенации будет следующая строка:

```
"Количество товаров в вашей корзине: 2"
```

Оператор if без условия else

Когда в условии else оператора if нет необходимости, то его можно просто опустить. Предположим, что в нашем приложении для интернет-магазина необходимо реализовать следующую возможность: пользователь, заказавший более десяти товаров, получает скидку 10 %. При подсчете общей стоимости покупки в процессе оформления заказа мы можем использовать код, аналогичный следующему:

```
if (numItems > 10) {
  totalPrice = totalPrice * .9;
}
```

Если значение переменной numItems будет меньше 11, то значение переменной totalPrice останется неизмененным.

Цепочка операторов if

Когда необходимо выбрать один из более чем двух возможных путей выполнения программы, следует объединить вместе несколько операторов if, как показано в следующем обобщенном коде для условия с тремя возможными результатами:

```
if (условноеВыражение1) {
  блокКода1
} else if (условноеВыражение2) {
  блокКода2
} else {
  блокКода3
}
```

Предположим, что нужно разработать многоязычное приложение, которое выводит приветствие для своих пользователей на одном из четырех языков: английском, японском, французском или немецком. При запуске программы мы просим пользователя выбрать язык и присваиваем соответствующей переменной language одно из следующих строковых значений: "english", "japanese", "french" или "german" (обратите внимание, что названия языков начинаются со строчных букв; обычно при сравнении строк все буквы записываются либо в нижнем, либо

в ВЕРХНЕМ регистре). Чтобы создать приветствие на выбранном языке, используем следующий код:

```
var greeting;

if (language == "english") {
  greeting = "Hello";
} else if (language == "japanese") {
  greeting = "Konnichiwa";
} else if (language == "french") {
  greeting = "Bonjour";
} else if (language == "german") {
  greeting = "Guten tag";
} else {
  // Расположенный здесь код может быть использован
  // для отображения сообщения об ошибке,
  // вызванной неправильно выбранным языком
}
```

Если при выполнении предыдущего кода значением переменной `language` является `"english"`, то переменной `greeting` присваивается значение `"Hello"`. Если значением переменной `language` является `"japanese"`, `"french"` или `"german"`, то переменной `greeting` присваивается значение `"Konnichiwa"`, `"Bonjour"` или `"Guten tag"` соответственно. Если переменная `language` не имеет ни одного из перечисленных значений (возможно, из-за возникшей ошибки в программе) — `"english"`, `"japanese"`, `"french"` или `"german"`, — то выполняется код, относящийся к последнему оператору `else`.

Теперь, когда мы познакомились с оператором `if`, рассмотрим оператор `switch`, предлагаемый языком ActionScript в качестве удобного способа создания условия с несколькими возможными результатами.

Поведение оператора switch можно реализовать и с помощью операторов if, однако, когда речь идет об условиях с несколькими возможными результатами, оператор switch считается более понятным, чем if.

Оператор switch

Оператор `switch` позволяет выполнять один из нескольких возможных блоков кода в зависимости от результата одного условного выражения. Оператор `switch` можно представить в следующем обобщенном виде:

```
switch (условноеВыражение) {
  case выражение1:
    блокКода1
    break;
  case выражение2:
    блокКода2
    break;
  default:
    блокКода3
}
```

В предыдущем коде *условноеВыражение* — это выражение, которое среда выполнения Flash попытается последовательно сопоставить со всеми указанными выражениями `case` сверху вниз. Выражения `case` записываются с помощью оператора-метки `case` и завершаются двоеточием. Если результат выражения *условноеВыражение* совпадает со значением выражения `case`, то выполняются все инструкции, расположенные за данной меткой `case`, включая инструкции во всех последующих блоках этого оператора! Чтобы предотвратить выполнение последующих блоков, необходимо использовать оператор `break` в конце каждого блока. Если же мы хотим, чтобы несколько условий инициировали выполнение одного и того же блока кода, то оператор `break` можно опустить. Например, в следующем примере кода `блокКода1` выполняется в тех случаях, когда результат выражения *условноеВыражение* совпадает со значением либо выражения *выражение1*, либо выражения *выражение2*:

```
switch (условноеВыражение) {
  case выражение1:
  case выражение2:
    блокКода1
    break;
  case выражение3:
    блокКода2
    break;
  default:
    блокКода3
}
```

Если результат выражения *условноеВыражение* не совпадает ни с одним из значений выражений `case`, то выполняются все инструкции, расположенные за меткой `default`.

Метка `default` обычно указывается после всех выражений `case`, однако с технической точки зрения ее можно поместить в любом месте оператора `switch`. Более того, эта метка не является обязательным атрибутом рассматриваемого оператора. Если `default` не указана и результат выражения *условноеВыражение* не совпадает ни с одним из значений выражений `case`, то выполнение программы продолжается с инструкции, расположенной сразу за оператором `switch` (то есть код, размещенный внутри оператора `switch`, просто не выполняется).

Следующий пример кода демонстрирует реализацию условия для создания приветствия на нескольких языках, которое было рассмотрено в предыдущем разделе, с использованием оператора `switch` вместо цепочки операторов `if`. Оба подхода работают одинаково, однако можно утверждать, что код с использованием оператора `switch` легче для чтения и зрительного восприятия.

```
var greeting;

switch (language) {
  case "english":
    greeting = "Hello";
    break;
```

```
  case "japanese":
    greeting = "Konnichiwa";
    break;
  case "french":
    greeting = "Bonjour";
    break;

  case "german":
    greeting = "Guten tag";
    break;

  default:
    // Расположенный здесь код может быть использован для отображения
    // сообщения об ошибке, вызванной неправильно выбранным языком
}
```

В операторе switch при сравнении результата выражения условноеВыражение со значениями выражений case неявно используется оператор строгого равенства (===), а не оператор равенства (==). Описание различий между этими операторами можно найти в справочнике по языку ActionScript корпорации Adobe.

Циклы

В предыдущем разделе мы узнали, что условный оператор позволяет единожды выполнить блок кода, если результатом его условного выражения является значение `true`. *Цикл*, в свою очередь, позволяет многократно выполнять блок до тех пор, пока результатом его условного выражения является значение `true`.

В языке ActionScript существует пять различных типов циклов: `while`, `do-while`, `for`, `for-in` и `for-each-in`. Первые три типа обладают схожей функциональностью, однако их синтаксис отличается. Оставшиеся два типа используются для доступа к динамическим переменным экземпляра объекта. Мы еще не рассматривали динамические переменные экземпляра, поэтому пока остановимся на первых трех типах циклов. Информацию по циклам `for-in` и `for-each-in` можно найти в гл. 15.

Оператор while

Структурно оператор `while` во многом напоминает `if`: основной оператор содержит блок кода, который выполняется только в том случае, если результатом заданного условного выражения является значение `true`:

```
while (условноеВыражение) {
  блокКода
}
```

Если результатом выражения *условноеВыражение* является `true`, то выполняются инструкции из блока *блокКода* (называемого *телом цикла*). Однако, в отличие от оператора `if`, когда выполнение блока *блокКода* завершается, управление снова передается на начало оператора `while` (то есть среда выполнения Flash «возвращается» к его началу). Второе выполнение оператора `while` ничем не отличается от первого: вычисляется

результат выражения *условноеВыражение* и, если его значением является `true`, снова выполняется блок *блокКода*. Этот процесс будет продолжаться до тех пор, пока результатом выражения *условноеВыражение* не станет `false`, после чего выполнение программы будет продолжено с инструкции, расположенной сразу за оператором `while`.

Если выражение *условноеВыражение* никогда не возвращает значение `false`, то цикл будет выполняться бесконечно и среда выполнения Flash будет вынуждена сгенерировать ошибку, которая остановит цикл (а вместе с ним и весь исполняемый в данный момент код). Чтобы избежать подобной ситуации, в блок *блокКода* цикла `while` обычно включается инструкция, которая модифицирует выражение *условноеВыражение*, заставляя его вернуть значение `false` при выполнении определенного условия.

Например, рассмотрим следующий цикл, определяющий результат возведения числа 2 в третью степень (то есть 2 умножается на 2 и умножается на 2), — тело цикла выполняется два раза:

```
var total = 2;
var counter = 0;

while (counter < 2) {
  total = total * 2;
  counter = counter + 1;
}
```

При выполнении предыдущего цикла `while` среда выполнения Flash сначала вычисляет результат условного выражения:

```
counter < 2
```

Поскольку значение переменной `counter` равно `0` и, соответственно, меньше 2, значением условного выражения является `true`; таким образом, Flash выполняет тело цикла:

```
total = total * 2;
counter = counter + 1;
```

В теле цикла переменной `total` присваивается ее текущее значение, умноженное на 2, а к значению переменной `counter` прибавляется 1. С этого момента значением переменной `total` является `4`, а переменной `counter` — `1`. После выполнения тела цикла наступает время повторить его.

При выполнении цикла во второй раз среда Flash снова проверяет значение условного выражения. На этот раз значением переменной `counter` является `1`, что по-прежнему меньше 2, поэтому значением условного выражения является `true`. Следовательно, Flash выполняет тело цикла во второй раз. Как и в предыдущий раз, переменной `total` присваивается ее текущее значение, умноженное на 2, а к значению переменной `counter` прибавляется 1. С этого момента значением переменной `total` является `8`, а переменной `counter` — `2`. После выполнения тела цикла снова наступает время повторить его.

При выполнении цикла в третий раз среда Flash снова проверяет значение условного выражения. На этот раз значением переменной `counter` является `2`, что уже *не* меньше 2, поэтому значением условного выражения является `false` и выполнение цикла прекращается. В процессе вычислений значение переменной `total`, которое

изначально равнялось 2, было дважды умножено на само себя, поэтому результатом является 8.

В реальном коде для выполнения экспоненциальных вычислений следует использовать функцию Math.pow( ), а не оператор цикла. Например, для возведения 2 в третью степень используется конструкция Math.pow(2, 3).

Хотя предыдущий цикл не вызывает особого восторга, он обладает потрясающей гибкостью. Например, если бы мы хотели возвести, скажем, число 2 в степень 16, мы могли бы просто обновить значение в условном выражении, чтобы тело цикла выполнялось 15 раз, как показано в следующем примере:

```
var total = 2;
var counter = 0;
while (counter < 15) {
  total = total * 2;
  counter = counter + 1;
}
// Здесь значение переменной total равно 65 536
```

Одно выполнение тела цикла называется *итерацией*. Соответственно, переменная, которая контролирует выполненное количество итераций данного цикла — в нашем случае это `counter`, — называется *итератором* или, реже, *индексом* цикла. Традиционно, для именования итераторов циклов используется буква `i`, как показано в следующем примере кода:

```
var total = 2;
var i = 0;
while (i < 15) {
  total = total * 2;
  i = i + 1;
}
```

Последняя строка в теле цикла из предыдущего примера называется *корректором цикла*, поскольку она до известной степени корректирует значение итератора, что в конечном счете приводит к завершению цикла. В данном случае корректор цикла прибавляет 1 к значению итератора. Эта операция является настолько распространенной, что для нее был создан собственный оператор: оператор инкремента, записываемый как ++. Оператор инкремента прибавляет 1 к значению своего операнда. Например, в следующем примере к значению переменной `n` прибавляется 1:

```
var n = 0;
n++;          // значение переменной n теперь равно 1
```

В следующем примере кода наш цикл реализован уже с использованием оператора инкремента:

```
var total = 2;
var i = 0;
while (i < 15) {
  total = total * 2;
  i++;
}
```

Противоположностью оператора инкремента является оператор декремента, записываемый как --. Он вычитает 1 из значения своего операнда. Например, в следующем примере из значения переменной n вычитается 1:

```
var n = 4;
n--;          // значение переменной n теперь равно 3
```

Оператор декремента зачастую используется в циклах, где значение итератора цикла уменьшается от указанной величины, а не увеличивается (как это происходило в предыдущих примерах). На протяжении этой книги мы будем использовать как оператор инкремента, так и оператор декремента. Однако вообще первый используется гораздо чаще, чем второй.

Обработка списков с помощью циклов

Циклы обычно используются для обработки списков элементов.

Предположим, что мы создаем регистрационную форму, в которой пользователь должен указать адрес электронной почты. Перед отправкой формы на сервер мы хотим проверить, содержит ли указанный адрес электронной почты символ @. Если этого символа нет, мы предупредим пользователя, что введенный адрес электронной почты является неверным.

Обратите внимание, что в этом примере наша концепция «правильного» адреса является крайне упрощенной. Например, в нашем коде адреса, начинающиеся или заканчивающиеся символом @ либо содержащие несколько символов @, считаются корректными. Тем не менее пример демонстрирует первый скромный шаг в создании алгоритма проверки адресов электронной почты.

Чтобы проверить наличие символа @ в адресе электронной почты, мы используем цикл, трактующий адрес как список из отдельных символов. Перед выполнением цикла мы создадим переменную `isValidAddress` и присвоим ей значение `false`. Тело цикла будет выполняться один раз для каждого символа в адресе электронной почты. При выполнении тела цикла в первый раз проверяется, является ли *первый* символ адреса электронной почты символом @. Если является, то в теле цикла переменной `isValidAddress` присваивается значение `true`, что указывает на корректность адреса электронной почты. При выполнении тела цикла во второй раз проверяется, является ли *второй* символ адреса электронной почты символом @. И снова, если этот символ найден, в теле цикла переменной `isValidAddress` присваивается значение `true`, что указывает на корректность адреса электронной почты.

Тело цикла продолжает проверять каждый символ в адресе электронной почты до тех пор, пока не останется ни одного символа. Если после выполнения цикла значение переменной `isValidAddress` по-прежнему равно `false`, значит, символ @ не был найден и, следовательно, адрес электронной почты считается некорректным. С другой стороны, если значение переменной `isValidAddress` равно `true`, значит, символ @ *был* найден и, следовательно, адрес электронной почты считается корректным.

Теперь взглянем на реальный код проверки. В настоящем приложении мы бы начали работу с получения адреса электронной почты, указанного пользователем.

Тем не менее, чтобы упростить этот пример, мы укажем адрес электронной почты вручную, как показано ниже:

```
var address = "me@moock.org";
```

После этого мы создадим переменную `isValidAddress` и присвоим ей значение `false`:

```
var isValidAddress = false;
```

Затем создадим итератор цикла:

```
var i = 0;
```

Далее определим оператор `while` для нашего цикла. Мы хотим, чтобы тело цикла выполнялось один раз для каждого символа строки, хранящейся в переменной `address`. Для получения количества символов в строке используется переменная экземпляра `length` класса `String`. Например, значением выражения `"abc".length` является 3. Это значит, что строка `abc` состоит из трех символов. Таким образом, общая структура нашего цикла выглядит следующим образом:

```
while (i < address.length) {
  i++;
}
```

Всякий раз при выполнении тела цикла мы должны получить один из символов строки, хранящейся в переменной `address`, и сравнить его со строкой `"@"`. Если полученный символ совпадает с ней, то присвоим переменной `isValidAddress` значение `true`. Чтобы получить определенный символ строки, воспользуемся методом экземпляра `charAt( )` собственного класса `String`. Название метода является сокращением фразы character at («символ в позиции»). Методу `charAt( )` необходимо передать один аргумент — число, указывающее позицию, или *индекс*, получаемого символа. Нумерация символов начинается с нуля. Например, результатом следующего выражения вызова является значение `"m"`, поскольку именно этот символ находится в позиции 0:

```
address.charAt(0);
```

Подобным образом результатом следующего выражения вызова является значение `"@"`, поскольку в позиции 2 находится символ `@`:

```
address.charAt(2);
```

В теле цикла индекс получаемого символа задается динамически через итератор `i`, как показано в следующем примере кода:

```
while (i < address.length) {
  if (address.charAt(i) == "@") {
    isValidAddress = true;
  }
  i++;
}
```

Код проверки адреса электронной почты целиком:

```
var address = "me@moock.org";
var isValidAddress = false;
var i = 0;
```

```
while (i < address.length) {
  if (address.charAt(i) == "@") {
    isValidAddress = true;
  }
  i++;
}
```

В качестве упражнения рассмотрим, как среда выполнения Flash будет выполнять предыдущий оператор `while`.

Сначала среда Flash вычисляет значение условного выражения:

```
i < address.length
```

Здесь значение переменной `i` равно `0`, а значение выражения `address.length` равно `12`. Число 0 меньше 12, поэтому результатом условного выражения является `true`, среда выполнения Flash выполняет тело цикла:

```
if (address.charAt(i) == "@") {
  isValidAddress = true;
}
i++;
```

В теле цикла среда Flash сначала должна определить, нужно ли выполнять код в условном операторе:

```
if (address.charAt(i) == "@") {
  isValidAddress = true;
}
```

Чтобы определить, нужно ли выполнять код в предыдущем условном операторе, Flash проверяет, совпадает ли результат выражения вызова `address.charAt(i)` со строкой `"@"`. При выполнении тела цикла в первый раз значение переменной `i` равно `0`, поэтому выражение вызова `address.charAt(i)` преобразуется в выражение `address.charAt(0)`, которое, как мы видели раньше, возвращает символ `"m"` (первый символ в адресе электронной почты). Символ `"m"` *не* равен символу `"@"`, поэтому среда Flash не выполняет код в условном операторе.

После этого Flash выполняет корректор цикла, увеличивая значение переменной `i` на 1:

```
i++;
```

После выполнения тела цикла наступает время повторить его.

При выполнении цикла во второй раз среда Flash снова проверяет значение условного выражения. На этот раз значение переменной `i` равно `1`, а значение выражения `address.length` по-прежнему равно `12`. Число 1 меньше 12, поэтому результатом условного выражения является значение `true` и Flash выполняет тело цикла во второй раз. Как и раньше, в теле цикла определяется, нужно ли выполнять код в условном операторе:

```
if (address.charAt(i) == "@") {
  isValidAddress = true;
}
```

Значение переменной `i` равно `1`, поэтому выражение вызова `address.charAt(i)` преобразуется в выражение `address.charAt(1)`, которое возвращает символ `"e"`

(второй символ в адресе электронной почты). Символ "e" вновь *не* равен символу "@", поэтому среда Flash не выполняет код в условном операторе.

После этого Flash выполняет корректор цикла, увеличивая значение переменной i до 2. И снова наступает время повторить цикл.

При выполнении цикла в третий раз Flash проверяет значение условного выражения. На этот раз значение переменной i равно 2, а значение выражения address.length по-прежнему равно 12. Число 2 меньше 12, поэтому результатом условного выражения является значение true и среда Flash выполняет тело цикла в третий раз. Как и раньше, в теле цикла определяется, нужно ли выполнять код в условном операторе:

```
if (address.charAt(i) == "@") {
  isValidAddress = true;
}
```

Значение переменной i равно 2, поэтому выражение вызова address.charAt(i) преобразуется в выражение address.charAt(2), которое возвращает символ "@". Символ "@" *равен* символу "@", поэтому среда Flash выполняет код в условном операторе, присваивая переменной isValidAddress значение true. После этого Flash выполняет корректор цикла, увеличивая значение переменной i до 3.

Тело цикла будет выполнено еще девять раз. После завершения оператора цикла значением переменной isValidAddress будет являться true. Это сообщает программе, что адрес электронной почты можно с уверенностью отправлять на сервер для дальнейшей обработки.

Завершение цикла с помощью оператора break

Цикл, описанный в предыдущем разделе, был работоспособным, но неэффективным. В соответствии с упрощенной логикой гипотетического алгоритма проверки адреса, если адрес электронной почты содержит символ @, он считается корректным. Чтобы определить наличие символа @, в цикле из предыдущего раздела проверялся каждый отдельный символ в указанном адресе электронной почты. В случае с адресом me@mock.org тело цикла выполнялось целых 12 раз, хотя уже после проверки третьего символа было ясно, что адрес является корректным. Следовательно, девять раз тело цикла выполнялось без надобности.

Чтобы сделать цикл из предыдущего раздела более эффективным, можно воспользоваться оператором break, который сразу завершает выполнение цикла:

```
var address = "me@moock.org";
var isValidAddress = false;
var i = 0;

while (i < address.length) {
  if (address.charAt(i) == "@") {
    isValidAddress = true;
    break;
  }
  i++;
}
```

В предыдущем примере кода, как только символ @ будет найден в строке, хранящейся в переменной address, переменной isValidAddress будет присвоено значение true, после чего оператор break завершит выполнение цикла.

Если создаваемый вами цикл используется для поиска определенного элемента в списке, то всегда применяйте оператор break для завершения цикла сразу после нахождения искомого элемента.

Упражнение: попробуйте модифицировать предыдущий цикл, чтобы адреса, начинающиеся или заканчивающиеся символом @ либо содержащие несколько символов @, считались некорректными. Кроме того, вы можете попробовать изменить цикл, чтобы некорректными считались и адреса, не содержащие ни одного символа . (точка).

Оператор do-while

Как уже отмечалось ранее, оператор while говорит среде выполнения Flash многократно выполнять блок кода до тех пор, пока результатом указанного условия является значение true. Из-за особенностей структуры цикла while его тело будет полностью пропущено, если при проверке условного выражения в первый раз его результатом не является значение true. Оператор do-while гарантирует, что тело цикла будет выполнено по крайней мере один раз с минимальными затратами. Синтаксис оператора do-while отчасти напоминает перевернутый синтаксис оператора while:

```
do {
  блокКода
} while (условноеВыражение);
```

Описание цикла начинается с ключевого слова do, за которым следует блок *блокКода* тела цикла. При первом прохождении цикла do-while блок *блокКода* выполняется даже до проверки результата выражения *условноеВыражение*. Если после завершения блока *блокКода* результатом выражения *условноеВыражение* является значение true, то цикл начинается заново и блок *блокКода* выполняется снова. Цикл выполняется до тех пор, пока выражение *условноеВыражение* не примет значение false.

Оператор for

Цикл for, по существу, является синонимом цикла while, однако для его записи применяется более компактный синтаксис. Выражения инициализации и корректирования цикла размещаются вместе с условным выражением в верхней части цикла for. Вот его синтаксис:

```
for (инициализация; условноеВыражение; корректирование) {
  блокКода
}
```

Перед первой итерацией цикла for выполняется выражение *инициализация* (один, и только один раз). Обычно это выражение используется для присваивания исходного

значения одной или нескольким переменным итераторов. Как и в случае с другими циклами, если результатом выражения *условноеВыражение* является значение `true`, то блок *блокКода* выполняется. В противном случае цикл завершается. И хотя выражение *корректирование* размещается в заголовке цикла, оно выполняется в *конце* каждой итерации, перед очередной проверкой результата выражения *условноеВыражение* на допустимость продолжения цикла.

Вот пример цикла `for`, используемого для возведения числа 2 в степень 3:

```
var total = 2;

for (var i = 0; i < 2; i++) {
  total = total * 2;
}
```

Для сравнения приведем эквивалентный цикл `while`:

```
var total = 2;
var i = 0;

while (i < 2) {
  total = total * 2;
  i++;
}
```

Следующий цикл `for` используется для определения наличия символа `@` в строке. С функциональной точки зрения этот цикл идентичен нашему предыдущему циклу `while`, который выполняет ту же задачу:

```
var address = "me@moock.org";
var isValidAddress = false;

for (var i = 0; i < address.length; i++) {
  if (address.charAt(i) == "@") {
    isValidAddress = true;
    break;
  }
}
```

Однажды применив на практике синтаксис цикла `for`, вы увидите, что он позволяет экономить место и в нем существует четкая грань между телом цикла и управляющими элементами.

Булева логика

Ранее в этой главе мы увидели, как принимаются логические решения с использованием условных выражений, которые возвращают логические значения. Решения принимались на основании одного фактора, например, «если значением переменной `language` является `"english"`, отображать сообщение `"Hello"`». Но не вся программная логика настолько проста. В программах зачастую приходится рассматривать сразу несколько факторов в *логике ветвлений* (то есть принимать решение). Для объединения нескольких факторов в одном условном

выражении применяются логические операторы: || (логическое ИЛИ) и && (логическое И).

Логическое ИЛИ

Оператор *логического* ИЛИ чаще всего применяется для инициирования определенного действия, когда выполняется по крайней мере одно из двух условий. Например, «если я голоден *или* испытываю жажду, я пойду на кухню». Оператор логического ИЛИ обозначается символами ||. Обычно символ | можно ввести, нажав одновременно клавишу Shift и клавишу с изображением обратного слэша (\), расположенную в правом верхнем углу большинства западных клавиатур. Оператор логического ИЛИ имеет следующий обобщенный вид:

выражение1 || *выражение2*

Если оба выражения (*выражение1* и *выражение2*) или результаты вычисления этих выражений являются логическими значениями, то оператор логического ИЛИ возвращает значение `true`, когда результатом одного из выражений является `true`, а значение `false` только в том случае, когда результатом обоих выражений является `false`.

```
true  || false  // true, поскольку первый операнд равен true
false || true   // true, поскольку второй операнд равен true
true  || true   // true (достаточно, чтобы любой операнд был равен true)
false || false  // false, поскольку оба операнда равны false
```

Когда результат выражения *выражение1* не является логическим значением, среда Flash сначала преобразует результат в логическое значение. Если результатом преобразования окажется значение `true`, то оператор логического ИЛИ вернет значение выражения *выражение1*. В противном случае оператор логического ИЛИ вернет значение выражения *выражение2*. Описанное правило продемонстрировано в следующих примерах:

```
0 || "hi there!"      // результат выражения выражение1 не преобразуется
                      // в значение true, поэтому оператор возвращает
                      // значение выражения выражение2: "hi there!"

"hey" || "dude"       // выражение выражение1 представляет непустую строку,
                      // поэтому результат этого выражения преобразуется
                      // в значение true и оператор возвращает значение
                      // выражения выражение1: "hey"

false || 5 + 5        // результат выражения выражение1 не преобразуется
                      // в значение true, поэтому оператор возвращает
                      // значение выражения выражение2 (то есть 10)
```

Результаты преобразования различных типов данных к логическим значениям перечислены в разд. «Преобразование в примитивные типы» гл. 8.

Возвращаемые оператором логического ИЛИ значения, которые не являются логическими, на практике используются редко. Вместо этого результат оператора ИЛИ обычно применяется в условных операторах для принятия логических решений. Рассмотрим следующий код:

```
var x = 10;
var y = 15;
if (x || y) {
  // Этот блок кода выполняется в том случае, когда значение
  // одной из переменных x или y не равно 0
}
```

В третьей строке вместо логического значения условного выражения оператора `if` мы видим оператор логического ИЛИ (`x || y`). Первый шаг в вычислении значения выражения `x || y` заключается в преобразовании числа 10 (которое является значением первого операнда — `x`) в логическое значение. Любое ненулевое конечное число преобразуется в логическое значение `true`. Оператор логического ИЛИ возвращает значение переменной `x`, равное `10`. Таким образом, для среды выполнения Flash оператор `if` выглядит следующим образом:

```
if (10) {
  // Этот блок кода выполняется в том случае, когда значение
  // одной из переменных x или y не равно 0
}
```

Однако 10 является числом, а не логическим значением. Что же происходит дальше? Оператор `if` преобразует результат выполнения оператора логического ИЛИ к логическому значению. В данном случае `10` преобразуется в логическое значение `true` и среда Flash представляет наш код следующим образом:

```
if (true) {
  // Этот блок кода выполняется в том случае, когда значение
  // одной из переменных x или y не равно 0
}
```

Вот и ответ. Результат условного выражения равен `true`, поэтому код, заключенный в фигурные скобки, выполняется.

Обратите внимание, что, если значение результата первого выражения оператора логического ИЛИ равно `true`, вычисление результата второго выражения является ненужным и, как следствие, неэффективным действием. По этой причине среда выполнения Flash вычисляет результат второго выражения только тогда, когда значение результата первого выражения равно `false`. Эту особенность полезно использовать в тех случаях, когда вы не хотите вычислять второе выражение до тех пор, пока результатом первого выражения не окажется `false`. В следующем примере выполняется проверка, входит ли указанное число в заданный диапазон. Если число слишком маленькое, нет необходимости в выполнении второй проверки, которая определяет, является ли число слишком большим.

```
if (xPosition < 0 || xPosition > 100) {
  // Этот блок кода выполняется, если значение переменной
  // xPosition находится в диапазоне от 1 до 100 включительно
}
```

Заметьте, что переменная `xPosition` должна быть включена в каждое сравнение. Следующий код демонстрирует распространенную ошибку, когда пытаются проверить значение переменной `xPosition` дважды:

```
// Ой! Забыли включить переменную xPosition в сравнение со значением 100
if (xPosition < 0 || > 100) {
  // Этот блок кода выполняется, если значение переменной
  // xPosition находится в диапазоне от 1 до 100 включительно
}
```

Логическое И

Как и оператор ИЛИ, оператор логического И в основном используется для условного исполнения блока кода — в данном случае, когда обязательно выполняются оба условия. Оператор логического И имеет следующий обобщенный вид:

выражение1 && *выражение2*

Выражения *выражение1* и *выражение2* могут быть любыми допустимыми. В простейшем случае, когда результатами обоих выражений являются логические значения, оператор логического И возвращает `false` в тех случаях, когда результатом одного из выражений является значение `false`, а `true` — только в том случае, когда результатом обоих выражений является значение `true`.

```
true  && false  // false, поскольку результат второго выражения равен false
false && true   // false, поскольку результат первого выражения равен false
true  && true   // true, поскольку результаты обоих выражений равны true
false && false  // false, поскольку результаты обоих выражений равны false
                // (достаточно, чтобы результат одного из выражений был
                // равен false)
```

Рассмотрим использование оператора логического И в двух примерах. В первом примере некоторый блок кода выполняется только в том случае, когда значения обеих переменных больше 50:

```
x = 100;
y = 51;
if (x>50 && y>50) {
  // Этот блок кода выполняется только в том случае,
  // когда значения переменных x и y больше 50
}
```

Теперь представим сайт с форумом, посвященным Новому году. Доступ к форуму пользователи могут получить только при вводе правильного пароля *и* только 1 января. Следующий код демонстрирует использование оператора логического И для проверки выполнения обоих условий (правильным паролем является слово fun):

```
var now = new Date( );          // Создает новый объект Date
var day = now.getDate( );       // Возвращает целое число в диапазоне
                                // от 1 до 31
var month = now.getMonth( );    // Возвращает целое число в диапазоне
                                // от 0 до 11

if ( password=="fun" && (month + day)==1 ) {
  // Позволить пользователю войти...
}
```

С технической точки зрения поведение оператора логического И очень похоже на поведение оператора логического ИЛИ. Сначала результат выражения *выражение1* преобразуется в логическое значение. Если результатом этого преобразования является `false`, то возвращается результат выражения *выражение1*. Если результатом преобразования является `true`, то возвращается результат выражения *выражение2*.

Как и в случае с оператором логического ИЛИ, если значение результата первого выражения оператора логического И равно `false`, нахождение результата второго выражения является ненужным и, как следствие, неэффективным действием. По этой причине среда выполнения Flash вычисляет результат второго выражения только тогда, когда значение результата первого выражения равно `true`. Эту особенность полезно использовать в тех случаях, когда вы не хотите вычислять второе выражение до тех пор, пока результатом первого выражения не окажется значение `true`. В следующем примере операция деления выполняется только в том случае, если делитель не равен нулю:

```
if ( numItems!=0 && totalCost/numItems>3 ) {
  // Этот блок кода выполняется только в том случае, когда количество
  // элементов не равно нулю и общая стоимость каждого элемента больше 3
}
```

Логическое НЕ

Оператор логического НЕ (!) возвращает логическое значение, противоположное значению его единственного операнда. Этот оператор записывается в следующем обобщенном виде:

!выражение

Если результатом выражения *выражение* является значение `true`, то оператор логического НЕ возвращает `false`. Если результатом выражения *выражение* является значение `false`, то оператор логического НЕ возвращает `true`. Если результат выражения *выражение* не является логическим значением, то для упрощения вычислений полученный результат преобразуется в логическое значение, после чего возвращается его противоположное значение.

Как и оператор неравенства (!=), оператор логического НЕ удобен для проверки того, чем *не является* тот или иной объект, а не того, чем он *является*. Например, тело следующего условного оператора выполняется только в том случае, если текущей датой *не* является 1 января. Обратите внимание на дополнительные скобки, с помощью которых задается требуемый порядок выполнения операций (приоритет), рассматриваемый в гл. 10.

```
var now = new Date( );        // Создает новый объект Date
var day = now.getDate( );     // Возвращает целое число в диапазоне
                              // от 1 до 31
var month = now.getMonth( ); // Возвращает целое число в диапазоне
                              // от 0 до 11

if ( !( (month + day)==1) ) {
  // Выполнение "непервоянварского" кода
}
```

Оператор логического НЕ иногда также используется для *переключения* значения переменной с `true` на `false` и наоборот. Например, предположим, что у нас есть одна кнопка, включающая и выключающая звук приложения. Когда кнопка нажата, программа может использовать следующий код для включения или выключения воспроизведения аудио:

```
soundEnabled = !soundEnabled // Переключение текущего состояния звука

if (soundEnabled) {
  // Убедиться, что звуки слышны
} else {
  // Выключить все звуки
}
```

Обратите внимание, что символ `!` также используется в операторе неравенства (`!=`). В программировании этот символ обычно обозначает «*не*» или «*противоположность*». Он не имеет никакого отношения к символу !, обозначающему факториал в обычной системе математических обозначений.

Возвращение к классам и объектам

Наше знакомство с условными операторами и циклами подошло к концу, однако мы не рассмотрели примеры применения абсолютно всех возможностей на практике. На протяжении этой книги мы еще встретим множество примеров использования условных операторов и циклов в реальных ситуациях.

В следующей главе мы вернемся к общим темам, касающимся классов и объектов. Если вы соскучились по нашим виртуальным животным, продолжайте изучение материала.

Пересмотр методов экземпляра

Из гл. 1 мы узнали, как создавать методы экземпляра. В этой главе мы расширим полученные базовые знания, рассмотрев следующие дополнительные темы, касающиеся методов экземпляра:

- исключение ключевого слова `this`;
- связанные методы;
- методы получения и изменения состояния;
- get- и set-методы;
- дополнительные аргументы.

В процессе изучения нового материала мы продолжим разрабатывать программу, создающую виртуальный зоопарк, начатую в гл. 1. Однако перед началом работы уделите несколько минут повторению уже пройденного материала. В листинге 3.1 продемонстрирована самая последняя версия кода на момент завершения гл. 1.

Листинг 3.1. Программа «Зоопарк»

```
// класс VirtualPet
package zoo {
  internal class VirtualPet {
    internal var petName:
    private var currentCalories = 1000:
    private var creationTime:

    public function VirtualPet (name) {
      this.creationTime = new Date( ):
      this.petName = name:
    }

    public function eat (numberOfCalories) {
      this.currentCalories += numberOfCalories:
    }

    public function getAge ( ) {
      var currentTime = new Date( ):
      var age = currentTime.time - this.creationTime.time:
      return age:
    }
  }
}

// класс VirtualZoo
package zoo {
```

```
  public class VirtualZoo {
    private var pet:

    public function VirtualZoo ( ) {
      this.pet = new VirtualPet("Stan");
    }
  }
}
```

Исключение ключевого слова this

Как известно из гл. 1, ключевое слово `this` используется для обращения к текущему объекту внутри методов-конструкторов или методов экземпляра. Например, в следующем коде выражение `this.petName = name` говорит среде выполнения присвоить значение переменной экземпляра `petName` созданного объекта:

```
public function VirtualPet (name) {
  this.petName = name;
}
```

В следующем коде выражение `this.currentCalories += numberOfCalories` говорит среде выполнения присвоить значение переменной `currentCalories` того объекта, метод `eat ( )` которого был вызван:

```
public function eat (numberOfCalories) {
  this.currentCalories += numberOfCalories;
}
```

Использование ключевого слова `this` в коде, в котором происходит частое обращение к переменным и методам текущего объекта, может оказаться утомительным, а также приведет к загруженности кода. Для упрощения и улучшения читабельности кода язык ActionScript позволяет обращаться к переменным и методам экземпляра текущего объекта вообще без использования ключевого слова `this`.

Вот как это работает: когда среда выполнения Flash встречает идентификатор в выражении внутри метода-конструктора или метода экземпляра, она выполняет поиск локальной переменной, параметра или вложенной функции, чье имя совпадает с именем данного идентификатора (вложенные функции рассматриваются в гл. 5). Если ни одно из имен локальных переменных, параметров или вложенных функций не совпадает с именем идентификатора, среда Flash автоматически выполняет поиск переменной или метода экземпляра, чье имя совпадает с именем идентификатора. Если совпадение найдено, то в выражении будут использованы соответствующие переменная или метод экземпляра.

Например, рассмотрим, что произойдет, если мы уберем ключевое слово `this` из метода `eat ( )`, как показано в следующем коде:

```
public function eat (numberOfCalories) {
  currentCalories += numberOfCalories;
}
```

При выполнении приведенного выше кода среда Flash встречает идентификатор `numberOfCalories` и пытается найти локальную переменную, параметр или вложенную функцию по данному имени. У метода *есть* параметр с таким именем, поэтому в выражении используется значение этого параметра (вместо идентификатора `numberOfCalories`).

После этого среда выполнения Flash встречает идентификатор `currentCalories` и пытается найти локальную переменную, параметр или вложенную функцию по данному имени. С именем `currentCalories` нет ни одной переменной, параметра или вложенной функции, поэтому среда Flash пытается найти переменную или метод экземпляра по данному имени. На этот раз поиск оказывается успешным: класс `VirtualPet` *содержит* переменную экземпляра с именем `currentCalories`, поэтому среда выполнения Flash использует эту переменную в выражении. В результате значение параметра `numberOfCalories` прибавляется к значению переменной экземпляра `currentCalories`.

Следовательно, внутри метода `eat( )` выражения `this.currentCalories` и `currentCalories` являются идентичными.

Для улучшения читабельности кода многие разработчики (это относится и к данной книге) избегают частого использования ключевого слова `this`. С этого момента при обращении к переменным и методам экземпляра мы будем опускать ключевое слово `this`. Тем не менее некоторые программисты предпочитают использовать его всегда лишь для того, чтобы отличать переменные и методы экземпляра от локальных переменных.

Обратите внимание, что использование ключевого слова `this` допустимо только внутри методов экземпляра, методов-конструкторов, функций и кода в глобальной области видимости (рассматривается в гл. 16). Применение ключевого слова `this` в любом другом месте программы приведет к ошибке на этапе компиляции.

Процесс поиска идентификаторов средой выполнения Flash называется разрешением идентификаторов. Как будет рассмотрено в гл. 16, разрешение идентификаторов выполняется с учетом области (или области видимости) программы, в которой они встречаются.

Разрешение конфликтов имен переменных/параметров. Когда переменная экземпляра и параметр метода имеют одинаковые имена, для обращения к переменной можно использовать ключевое слово `this` (это называется *разрешением неоднозначности* между переменной и параметром). Например, в следующей модифицированной версии класса `VirtualPet` представлен метод `eat( )` с параметром `calories`, имя которого совпадает (то есть конфликтует) с именем переменной экземпляра `calories`:

```
package zoo {
  internal class VirtualPet {
    // Переменная экземпляра 'calories'
    private var calories = 1000;

    // Метод с параметром 'calories'
    public function eat (calories) {
```

```
      this.calories += calories;
    }
  }
}
```

Внутри тела метода `eat( )` выражение `calories` (без ключевого слова `this`) ссылается на параметр метода, а выражение `this.calories` (с ключевым словом `this`) — на переменную экземпляра. В этом случае говорят, что параметр `calories` *затеняет* переменную экземпляра `calories`, поскольку сам по себе идентификатор `calories` ссылается на параметр, а не на переменную экземпляра. Обратиться к переменной экземпляра можно только с помощью ключевого слова `this`.

Обратите внимание, что, как и параметры, локальные переменные могут затенять переменные и методы экземпляра, чьи имена совпадают с именами локальных переменных. Локальная переменная также затеняет параметр метода с таким же именем, фактически переопределяя данный параметр и не оставляя программе никакого шанса обратиться к нему.

Многие программисты специально используют одинаковые имена для параметров и переменных экземпляра, полагаясь на ключевое слово `this` при разрешении неоднозначностей. Тем не менее, чтобы ваш код легко читался, вы можете просто избегать использования имен параметров, совпадающих с именами переменных, методов экземпляра или локальных переменных.

Теперь перейдем к рассмотрению следующего вопроса, касающегося методов экземпляра, — связанных методов.

Связанные методы

В языке ActionScript сам метод может рассматриваться как значение. Другими словами, метод может быть присвоен переменной, передан в функцию или другой метод, а также возвращен из функции или другого метода.

Например, в следующем коде создается новый объект класса `VirtualPet`, после чего локальной переменной `consume` присваивается метод `eat( )` созданного объекта. Обратите внимание, что в операторе присваивания после имени метода не ставятся круглые скобки `( )` вызова метода. В результате переменной `consume` присваивается сам метод, а не его возвращаемое значение.

```
package zoo {
  public class VirtualZoo {
    private var pet;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      // Присваивание переменной метода eat( )
      var consume = pet.eat;
    }
  }
```

Метод, присвоенный переменной, может быть вызван через эту переменную стандартным оператором круглых скобок — ( ). Например, в следующем коде вызывается метод, на который ссылается переменная consume:

```
package zoo {
  public class VirtualZoo {
    private var pet:

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      // Присваивание связанного метода переменной consume
      var consume = pet.eat;
      // Вызов метода, на который ссылается переменная consume
      consume(300);
    }
  }
}
```

При выполнении предыдущего кода, выделенного полужирным шрифтом, вызывается метод eat ( ), в качестве аргумента которого передается значение 300. Вопрос заключается в том, какое животное принимает пищу? Или, говоря техническим языком, над каким объектом выполняется этот метод?

Когда метод присваивается переменной и затем вызывается через нее, код выполняется над тем объектом, который изначально использовался для обращения к методу. Например, в предыдущем коде, когда метод eat ( ) присваивается переменной consume, обращение к методу происходит через объект класса VirtualPet с именем "Stan". Таким образом, при вызове метода eat ( ) через переменную consume код будет выполняться над объектом класса VirtualPet с именем "Stan".

Метод, который присваивается переменной, передается в функцию или другой метод либо возвращается из функции или другого метода, называется *связанным методом*. Свое название связанные методы получили потому, что каждый такой метод навсегда связывается с объектом, через который изначально происходит обращение к методу. Связанные методы являются экземплярами собственного класса Function.

При вызове связанного метода не нужно указывать объект, над которым должен выполняться данный метод. Вместо этого связанный метод будет автоматически выполнен над объектом, который изначально использовался при создании связи.

Внутри тела связанного метода ключевое слово this ссылается на объект, с которым связан данный метод. Например, внутри тела связанного метода, который был присвоен переменной consume, ключевое слово this ссылается на объект класса VirtualPet с именем "Stan".

Связанные методы используются в тех случаях, когда одна часть программы желает, чтобы другая часть выполнила определенный метод над заданным объектом. Примеры такого сценария можно найти в гл. 12, где рассматривается система обработки событий. Связанные методы нашли широкое применение в системе обработки событий языка ActionScript.

Продолжая тему разговора о методах экземпляра, в следующем разделе рассмотрим использование методов экземпляра для изменения состояния объекта.

Использование методов для получения и изменения состояния объекта

Ранее в этой книге мы узнали, что хорошей практикой объектно-ориентированного программирования является объявление переменных экземпляра с использованием модификатора `private`. Это значит, что их значения нельзя прочитать или изменить в коде за пределами класса, в котором они объявлены.

Хорошая практика объектно-ориентированного программирования также предписывает следующее: вместо того чтобы предоставлять возможность внешнему коду непосредственно изменять значения переменных экземпляра, мы должны определять методы экземпляра для получения или изменения состояния объекта.

Ранее мы определили переменную экземпляра с именем `currentCalories` в классе `VirtualPet`. Концептуально эта переменная описывает степень голода каждого животного. Чтобы внешний код мог изменять степень голода животного, мы *можем* сделать переменную `currentCalories` открытой. В этом случае внешний код будет присваивать переменной, описывающей степень голода, любое произвольное значение, как показано в следующем коде:

```
somePet.currentCalories = 5000;
```

Тем не менее предыдущий подход обладает серьезным недостатком. Если внешний код сможет непосредственно изменять значение переменной `currentCalories`, то у класса `VirtualPet` не будет никакой возможности убедиться, что значение, присваиваемое переменной, является допустимым, или осмысленным. Например, внешний код может присвоить переменной `currentCalories` значение `1 000 000`, позволив животному жить сотни лет, не испытывая голода. Или, если внешний код присвоит переменной `currentCalories` отрицательное значение, может произойти сбой программы.

Для того чтобы избежать подобных проблем, мы должны объявить переменную `currentCalories` закрытой (как мы сделали это раньше в классе `VirtualPet`). Вместо того чтобы позволять внешнему коду непосредственно изменять значение переменной `currentCalories`, мы добавим один или несколько открытых методов экземпляра, которые могут быть использованы для изменения степени голода каждого животного допустимым способом. Наш класс `VirtualPet` уже обладает методом `eat( )` для утоления голода животного. Тем не менее этот метод позволяет добавлять любое количество калорий к значению переменной `currentCalories`. Модифицируем метод `eat( )` класса `VirtualPet` таким образом, чтобы значение переменной `currentCalories` не могло превышать `2000`:

```
public function eat (numberOfCalories) {
  currentCalories += numberOfCalories;
}
```

Чтобы ограничить максимальное значение переменной `currentCalories` числом `2000`, мы просто добавим оператор `if` в метод `eat( )`, как показано в следующем коде:

```
public function eat (numberOfCalories) {
  // Рассчитать новое предложенное количество калорий
  // для данного животного
  var newCurrentCalories = currentCalories + numberOfCalories;

  // Если новое предложенное количество калорий для данного животного
  // больше максимально допустимого значения (то есть 2000)...
  if (newCurrentCalories > 2000) {
    // ...присвоить переменной currentCalories максимально
    // допустимое значение (2000)
    currentCalories = 2000;
  } else {
    // ...в противном случае увеличить значение переменной currentCalories
    // на указанное количество калорий
    currentCalories = newCurrentCalories;
  }
}
```

Метод `eat( )` класса `VirtualPet` предоставляет внешнему коду безопасный способ изменения степени голода выбранного объекта `VirtualPet`. Однако до сих пор класс `VirtualPet` не предоставлял внешнему коду никакой возможности для определения степени голода. Чтобы предоставить внешнему коду доступ к этой информации, определим метод `getHunger( )`, возвращающий оставшееся количество калорий у объекта `VirtualPet`, выраженное в процентах. Код нового метода выглядит следующим образом:

```
public function getHunger ( ) {
  return currentCalories / 2000;
}
```

Теперь у нас есть методы для получения и изменения текущей степени голода объекта `VirtualPet` (`getHunger( )` и `eat( )`). В традиционной терминологии объектно-ориентированного программирования метод, который *получает* состояние объекта, называется *методом-аксессором*, или, более неофициально, *методом-читателем*. С другой стороны, метод, который изменяет состояние объекта, называется *методом-мутатором*, или, более неофициально, *методом-писателем*. Тем не менее в языке ActionScript 3.0 термин «метод-аксессор» относится к особой разновидности методов, которые оформляются с использованием синтаксиса чтения и записи значения переменной и рассматриваются далее, в разд. «Get- и set-методы». Как отмечалось ранее, чтобы избежать путаницы в этой книге, мы не будем употреблять традиционные термины «аксессор», «мутатор», «читатель» и «писатель». Вместо этого мы воспользуемся неофициальными терминами *метод-получатель* и *метод-модификатор* при обсуждении методов-аксессоров и методов-мутаторов. Более

того, мы будем применять термины «get-метод» и «set-метод» только в отношении специальных автоматических методов языка ActionScript.

Чтобы немного попрактиковаться в использовании методов-получателей и методов-модификаторов, снова изменим класс `VirtualPet`. Раньше для получения и присвоения имени объекту `VirtualPet` мы обращались непосредственно к переменной `petName`, как показано в следующем коде:

```
somePet.petName = "Erik";
```

Предыдущий подход тем не менее в дальнейшем может стать источником проблем в нашей программе. Он позволяет присваивать переменной `petName` очень длинные значения, которые могут не вместиться на экране при отображении имени животного. Кроме того, переменной `petName` может быть присвоена пустая строка (`""`), которая вообще не отобразится на экране. Чтобы избежать описанных проблем, объявим переменную `petName` закрытой и определим метод-модификатор для задания имени животного. Наш метод-модификатор `setName( )` ограничивает максимальную длину имени 20 символами и предотвращает попытки присвоить переменной `petName` пустую строку (`""`):

```
public function setName (newName) {
  // Если длина заданного нового имени больше 20 символов...
  if (newName.length > 20) {
    // ...обрезать имя, используя собственный метод String.substr( ),
    // возвращающий указанную часть строки, над которой
    // выполняется данный метод
    newName = newName.substr(0, 20);
  } else if (newName == "") {
    // ...в противном случае, если заданное новое имя является
    // пустой строкой, завершить выполнение метода, не изменяя
    // значения переменной petName
    return;
  }

  // Присвоить новое проверенное имя переменной petName
  petName = newName;
}
```

Теперь, когда мы объявили переменную `petName` закрытой, необходимо определить метод-получатель, с помощью которого внешний код сможет получить доступ к имени объекта `VirtualPet`. Мы присвоим нашему методу-получателю имя `getName( )`. Пока этот метод будет просто возвращать значение переменной `petName` (зачастую возвращение значения переменной экземпляра является единственной задачей метода-получателя). Рассмотрим код метода:

```
public function getName ( ) {
  return petName;
}
```

В настоящее время метод `getName( )` очень прост, однако он добавляет гибкость в нашу программу. Например, в будущем может понадобиться, чтобы имена животных формировались с учетом пола. Для этого мы просто обновим метод, как

показано далее (следующая гипотетическая версия метода `getName( )` предполагает, что в классе `VirtualPet` определена переменная экземпляра `gender`, хранящая пол каждого животного):

```
public function getName ( ) {
  if (gender == "male") {
    return "Mr. " + petName;
  } else {
    return "Mrs. " + petName;
  }
}
```

В листинге 3.2 продемонстрирован новый код класса `VirtualPet`, в который были добавлены определения методов `getName( )` и `setName( )`. Для лучшей читабельности метод экземпляра `getAge( )` и переменная экземпляра `creationTime` были удалены из описания класса `VirtualPet`.

Листинг 3.2. Класс VirtualPet

```
package zoo {
  internal class VirtualPet {
    private var petName;
    private var currentCalories = 1000;

    public function VirtualPet (name) {
      petName = name;
    }

    public function eat (numberOfCalories) {
      var newCurrentCalories = currentCalories + numberOfCalories;
      if (newCurrentCalories > 2000) {
        currentCalories = 2000;
      } else {
        currentCalories = newCurrentCalories;
      }
    }

    public function getHunger ( ) {
      return currentCalories / 2000;
    }

    public function setName (newName) {
      // Если длина заданного нового имени больше 20 символов...
      if (newName.length > 20) {
        // ...обрезать имя
        newName = newName.substr(0, 20);
      } else if (newName == "") {
        // ...в противном случае, если заданное новое имя является
        // пустой строкой, завершить выполнение метода, не изменяя
        // значения переменной petName
        return;
      }
```

```
      // Присвоить новое проверенное имя переменной petName
      petName = newName;
    }

    public function getName ( ) {
      return petName;
    }
  }
}
```

Теперь рассмотрим пример использования наших новых методов `getName( )` и `setName( )`:

```
package zoo {
  public class VirtualZoo {
    private var pet;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      // Присвоить старое имя животного локальной переменной oldName
      var oldName = pet.getName( );
      // Дать животному новое имя
      pet.setName("Marcos");
    }
  }
}
```

Используя метод-модификатор для промежуточного присваивания значения переменной, мы можем разрабатывать приложения, способные адекватно реагировать на ошибки времени выполнения путем определения и обработки недопустимых или неподходящих значений. Значит ли это, что доступ ко всем переменным экземпляра должен осуществляться через методы? Например, рассмотрим метод-конструктор нашего класса `VirtualPet`:

```
public function VirtualPet (name) {
  petName = name;
}
```

Теперь, когда у нас появился метод для изменения значения переменной `petName`, должны ли мы модифицировать метод-конструктор класса `VirtualPet` следующим образом?

```
public function VirtualPet (name) {
  setName(name);
}
```

Ответ зависит от имеющихся условий. Вообще говоря, непосредственное обращение к закрытым переменным, описанным в данном классе, является вполне приемлемым. Тем не менее, если существует вероятность, что имя или роль переменной в будущем будут изменены, или если метод-модификатор или метод-получатель выполняют определенные действия при обращении к переменной (например, проверку на наличие ошибок), имеет смысл применять методы везде, даже внутри класса, в котором описана данная переменная. Например, в предыдущем

модифицированном методе-конструкторе класса `VirtualPet` разумно присваивать значение переменной `petName` именно через метод `setName( )`, поскольку это гарантирует, что указанное имя не окажется слишком длинным или коротким. И все-таки, в тех случаях, когда решающим фактором является быстродействие, благоразумнее использовать непосредственный доступ к переменной (его получить всегда быстрее, чем доступ через метод).

Программисты, предпочитающие использовать стиль непосредственного доступа к переменным, но при этом не желающие отказываться от преимуществ методов-получателей и методов-модификаторов, обычно применяют автоматические get- и set-методы языка ActionScript, рассматриваемые в следующем разделе.

Get- и set-методы

В предыдущем разделе мы познакомились с методами-получателями и методами-модификаторами, которые представляют собой открытые методы для получения и изменения состояния объекта. Некоторые разработчики считают подобные методы громоздкими. Они утверждают, что конструкция:

```
pet.setName("Jeff");
```

более неудобна в использовании, чем конструкция:

```
pet.name = "Jeff";
```

Чуть раньше мы убедились, что непосредственное присваивание значения переменной, например `pet.name = "Jeff"`, не является идеальной практикой объектно-ориентированного программирования и переменной в конечном счете может быть присвоено некорректное значение. Чтобы устранить несогласованность между удобством использования синтаксиса непосредственного присваивания значения переменной и безопасностью методов-получателей и методов-модификаторов, язык ActionScript поддерживает get- и set-методы. Вызвать эти методы можно с помощью синтаксиса получения или присваивания значения переменной.

Для описания get-метода используется следующий обобщенный синтаксис:

```
function get имяМетода ( ) {
  операторы
}
```

Здесь ключевое слово `get` указывает на то, что метод является get-методом, *имяМетода* представляет имя метода, а *операторы* — это ноль или более операторов, выполняемых при вызове метода (ожидается, что один из операторов возвращает значение, связанное с методом *имяМетода*).

Для описания set-метода используется следующий обобщенный синтаксис:

```
function set имяМетода (новоеЗначение) {
  операторы
}
```

Здесь ключевое слово `set` указывает на то, что метод является set-методом, *имяМетода* представляет имя метода, параметр *новоеЗначение* содержит значение, присваиваемое

внутренней переменной экземпляра, а *операторы* — это ноль или более операторов, выполняемых при вызове метода. Ожидается, что блок операторов *операторы* определит и внутренне сохранит значение, связанное с методом *имяМетода*. Обратите внимание, что в теле set-метода оператор `return` не должен применяться для возврата значения (однако сам по себе он может быть использован для завершения метода). Set-методы автоматически возвращают значение, что рассматривается далее.

Для вызова get- и set-методов применяется уникальный стиль, не требующий использования оператора вызова функции ( ). Get-метод `x( )` объекта `obj` вызывается следующим образом:

```
obj.x;
```

Но не так:

```
obj.x( );
```

Set-метод `y( )` объекта `obj` вызывается следующим образом:

```
obj.y = value;
```

Но не так:

```
obj.y(value);
```

Здесь `value` является первым (и единственным) аргументом, передаваемым в метод `y( )`.

Следовательно, get- и set-методы неким магическим образом позволяют преобразовать синтаксис обращения к переменным экземпляра в вызовы методов. В качестве примера (временно) добавим get-метод с именем `name( )` в наш класс `VirtualPet`:

```
public function get name ( ) {
  return petName;
}
```

Теперь, когда в классе определен get-метод `name( )`, все попытки получить значение переменной экземпляра `name` на самом деле приведут к вызову этого get-метода. Возвращаемое значение get-метода выглядит так, будто на самом деле было получено значение переменной `name`. Например, следующий код вызывает get-метод `name( )` и присваивает его возвращаемое значение переменной `oldName`:

```
var oldName = pet.name;
```

Сейчас (временно) добавим set-метод с именем `name( )` в наш класс `VirtualPet`:

```
public function set name (newName) {
  // Если длина заданного нового имени больше 20 символов...
  if (newName.length > 20) {
    // ...обрезать имя
    newName = newName.substr(0, 20);
  } else if (newName == "") {
    // ...в противном случае, если заданное новое имя является
    // пустой строкой, завершить выполнение метода, не изменяя
    // значения переменной petName
    return;
  }
```

```
  // Присвоить новое проверенное имя переменной petName
  petName = newName;
}
```

Теперь, когда в классе определен set-метод `name( )`, попытки присвоить значение переменной экземпляра `name` приведут к вызову данного set-метода. Значение, используемое в операторе присваивания переменной `name`, передается в set-метод, который внутренне сохраняет это значение в закрытой переменной `petName`. Например, следующий код вызывает set-метод `name( )`, который внутренне сохраняет значение `"Andreas"` в переменной `petName`:

```
pet.name = "Andreas";
```

После определения get- и set-метода с именем `name( )` переменная `name` становится всего лишь внешним фасадом. В действительности она не определена в классе, однако обращаться к ней можно так же, как и к любой другой существующей переменной. Таким образом, вы можете считать переменные экземпляра, сопровождаемые get- и set-методами (например, `name`), *псевдопеременными*.

Нельзя создавать реальную переменную с именем, совпадающим с названием get- или set-метода. Подобные попытки приведут к ошибке на этапе компиляции.

При вызове set-метода всегда вызывается соответствующий get-метод, результат которого возвращается из данного set-метода. Это позволяет программе использовать новое значение сразу после операции присваивания. Например, следующий код демонстрирует фрагмент приложения музыкального проигрывателя. Для выбора первой воспроизводимой песни используется set-метод. Благодаря вызову метода `start( )` над возвращаемым значением оператора присваивания переменной `firstSong` сразу начинается воспроизведение выбранной песни.

```
// Вызов метода start( ) над объектом new Song("dancehit.mp3") –
// возвращаемым значением set-метода firstSong( )
(musicPlayer.firstSong = new Song("dancehit.mp3")).start( );
```

Хотя возможность возвращения значений из set-методов в некоторых случаях оказывается удобной, она накладывает ограничения на get-методы: в частности, get-методы не должны выполнять задачи, которые не требуются для получения значения соответствующей внутренней переменной. Например, с помощью get-метода нельзя реализовать глобальный счетчик, отслеживающий количество обращений к переменной. Автоматический вызов get-метода из set-метода приведет к лишнему увеличению значения счетчика.

Псевдопеременную, обращение к которой происходит с помощью get- и set-метода, можно сделать доступной только для чтения — для этого нужно объявить get-метод и опустить объявление set-метода.

Выбор между использованием методов-получателей/-модификаторов и get-/set-методов — дело вкуса. В этой книге, например, get- и set-методы не используются, однако вы можете встретить этот подход в коде других программистов или в другой документации.

Чтобы завершить изучение методов экземпляра, рассмотрим, как обрабатывать неизвестное количество параметров. Для изучения материала следующего раздела необходимо иметь представление о массивах (упорядоченных списках значений), которые еще не рассматривались в этой книге. Если вы незнакомы с массивами, пока пропустите этот раздел и вернитесь к нему после прочтения гл. 11.

Методики, описанные в следующем разделе, применимы не только к методам экземпляра, но и к статическим методам и функциям, которые будут рассмотрены в следующих главах.

Обработка неизвестного количества параметров

Как известно из гл. 1, нельзя вызвать метод, не указав аргументы для всех обязательных параметров. Нельзя также вызывать метод, если указано *больше* аргументов, чем требуется.

Чтобы определить метод, который принимает произвольное количество аргументов, используется параметр `...(rest)`. Он описывает массив, содержащий все аргументы, передаваемые в данный метод. Этот параметр может использоваться как самостоятельно, так и в сочетании с именованными параметрами. Когда параметр `...(rest)` используется отдельно, описание метода имеет следующий обобщенный вид:

```
function имяМетода (...массивАргументов) {
}
```

В предыдущем коде *имяМетода* обозначает имя метода (или функции), а *массивАргументов* представляет имя параметра, присваиваемое автоматически создаваемому массиву, который содержит все передаваемые в данный метод аргументы. Первый аргумент (крайний левый в выражении вызова) хранится под индексом 0, и обратиться к нему можно с помощью выражения *массивАргументов*`[0]`. Последующие аргументы сохраняются по порядку слева направо. Таким образом, для обращения ко второму аргументу используется выражение *массивАргументов*`[1]`, к третьему — выражение *массивАргументов*`[2]` и т. д.

Параметр `...(rest)` позволяет создавать очень гибкие функции, оперирующие произвольным количеством значений. Например, следующий код демонстрирует метод, который определяет среднее значение любых чисел, передаваемых в качестве аргументов:

```
public function getAverage (...numbers) {
  var total = 0;

  for (var i = 0; i < numbers.length; i++) {
    total += numbers [i];
  }

  return total / numbers.length;
}
```

Обратите внимание, что представленный метод `getAverage( )` работает только с числовыми аргументами. Чтобы защитить этот метод от использования нечисловых аргументов, можно применить оператор `is`, рассматриваемый в подразд. «Восходящее и нисходящее приведения типов» разд. «Приведение типов» гл. 8.

Параметр `...(rest)` также может использоваться в сочетании с именованными параметрами. В этом случае он должен быть последним в списке параметров. Например, рассмотрим метод `initializeUser( )`, применяемый для инициализации пользователя в гипотетическом социальном сетевом приложении. В описании метода определяется один обязательный параметр `name`, за которым следует параметр `...(rest)` с именем `hobbies`:

```
public function initializeUser (name, ...hobbies) {
}
```

При вызове метода `initializeUser( )` мы обязаны указать аргумент для параметра `name` и при желании можем указать дополнительный список хобби, разделяя элементы списка запятыми. Внутри метода параметру `name` присваивается значение первого переданного аргумента, а параметру `hobbies` — массив всех оставшихся аргументов. Например, если вызвать данный метод следующим образом:

```
initializeUser("Hoss", "video games", "snowboarding");
```

то параметру `name` будет присвоено значение `"Hoss"`, а параметру `hobbies` — значение `["video games", "snowboarding"]`.

Далее: информация и поведение на уровне класса

Мы рассмотрели методы и переменные экземпляра. Как известно из гл. 1, они определяют поведение и характеристики объектов класса. Из следующей главы вы узнаете, как создавать поведение и управлять информацией, относящейся не к отдельным объектам, а ко всему классу.

Статические переменные и методы

Из гл. 1 вы узнали, как определять характеристики и поведение объекта с помощью переменных и методов экземпляра. В этой главе вы познакомитесь с тем, как организовывать информацию и создавать функциональность, относящуюся к самому классу, а не к его экземплярам.

Статические переменные

При изучении материала предыдущих глав мы немного попрактиковались в использовании переменных экземпляра, которые представляют собой переменные, связанные с определенным экземпляром класса. В отличие от них, *статические переменные* связаны с самим классом, а не с его определенным экземпляром. Статические переменные применяются для хранения информации, относящейся логически ко всему классу, в отличие от информации, которая меняется от одного экземпляра к другому. Например, в классе, представляющем окно, статическая переменная может использоваться для указания размера по умолчанию для новых экземпляров этого окна, а в классе, представляющем автомобиль в гоночной игре, с помощью статической переменной можно задать максимальную скорость для всех экземпляров этого автомобиля.

Подобно переменным экземпляра, для создания статических переменных применяются описания переменных, размещаемые внутри класса, однако описание статической переменной должно также включать атрибут `static`, как показано в следующем обобщенном коде:

```
class НекийКласс {
  static var идентификатор = значение;
}
```

Как и в случае с переменными экземпляра, для управления доступностью статических переменных в программе можно использовать модификаторы управления доступом. Такие модификаторы идентичны модификаторам, применяемым в описаниях переменных экземпляра, — `public`, `internal`, `protected` и `private`. Если в описании переменной никакой модификатор не задан, то используется модификатор `internal` (доступ внутри пакета). Указываемый модификатор обычно размещается перед атрибутом `static`, как показано в следующем коде:

```
class НекийКласс {
  private static var идентификатор = значение;
}
```

Для доступа к статической переменной указывается имя класса, содержащего определение данной переменной, за которым следует точка (.) и имя переменной, как показано в следующем обобщенном коде:

НекийКласс.идентификатор = значение;

Внутри класса, в котором объявлена данная переменная, имя *идентификатор* может использоваться и самостоятельно (без лидирующего имени класса и точки). Например, в классе A, в котором определена статическая переменная v, выражение A.v идентично выражению v. Тем не менее, чтобы различать статические переменные и переменные экземпляра, многие разработчики (это относится и к примерам данной книги) включают лидирующее имя класса даже в тех случаях, когда его использование не является обязательным.

Статические переменные и переменные экземпляра с одинаковыми именами могут сосуществовать внутри одного класса. Если в классе A определена переменная экземпляра v и статическая переменная с таким же именем, то при вызове в виде v идентификатор будет ссылаться на переменную экземпляра, а не на статическую переменную. Обратиться к статической переменной можно только путем указания лидирующего имени класса: A.v. В этом случае говорят, что переменная экземпляра *затеняет* статическую переменную.

Теперь добавим несколько статических переменных в наш класс VirtualPet. Как известно, статические переменные используются для хранения информации, которая логически относится ко всему классу и не меняется от одного экземпляра к другому. В нашем классе VirtualPet уже есть два примера подобной информации: максимальная длина имени животного и максимальное количество калорий, которое может принять данное животное. Для хранения этой информации добавим две новые статические переменные: maxNameLength и maxCalories. Мы не будем обращаться к нашим переменным за пределами класса VirtualPet, поэтому объявим их с использованием модификатора доступа private. Следующий код демонстрирует объявления переменных maxNameLength и maxCalories, при этом оставшаяся часть кода класса VirtualPet опущена ради краткости:

```
package zoo {
  internal class VirtualPet {
    private static var maxNameLength = 20;
    private static var maxCalories = 2000;

    // Оставшаяся часть класса не показана...
  }
}
```

Теперь, когда у нас есть переменные maxNameLength и maxCalories, мы можем модифицировать методы getHunger( ), eat( ) и setName( ), чтобы воспользоваться этими переменными. В листинге 4.1 продемонстрирована последняя версия класса VirtualPet, в который были добавлены статические переменные. Изменения, внесенные в предыдущую версию кода, выделены полужирным шрифтом. Обратите внимание, что по соглашению статические переменные класса перечислены перед переменными экземпляра.

Листинг 4.1. Класс VirtualPet

```
package zoo {
  internal class VirtualPet {
    private static var maxNameLength = 20;
    private static var maxCalories = 2000;

    private var petName;
    // Изначально каждому животному дается 50 % от максимально
    // возможного количества калорий.
    private var currentCalories = VirtualPet.maxCalories/2;

    public function VirtualPet (name) {
      setName(name);
    }

    public function eat (numberOfCalories) {
      var newCurrentCalories = currentCalories + numberOfCalories;
      if (newCurrentCalories > VirtualPet.maxCalories) {
        currentCalories = VirtualPet.maxCalories;
      } else {
        currentCalories = newCurrentCalories;
      }
    }

    public function getHunger ( ) {
      return currentCalories / VirtualPet.maxCalories;
    }

    public function setName (newName) {
      // Если длина заданного нового имени больше maxNameLength символов...
      if (newName.length > VirtualPet.maxNameLength) {
        // ...обрезать имя
        newName = newName.substr(0, VirtualPet.maxNameLength);
      } else if (newName == "") {
        // ...в противном случае, если заданное новое имя является
        // пустой строкой, завершить выполнение метода, не изменяя
        // значения переменной petName
        return;
      }

      // Присвоить новое проверенное имя переменной petName
      petName = newName;
    }

    public function getName ( ) {
      return petName;
    }
  }
}
```

Если проанализировать листинг 4.1, то можно заметить, что наличие переменных `maxNameLength` и `maxCalories` помогает централизовать код. Например,

ранее, чтобы изменить максимально допустимое количество символов в имени, нам пришлось бы заменить число 20 в двух местах тела метода `setName` — этот процесс не только отнимает много времени, но и может привести к ошибке. Теперь, чтобы изменить максимально допустимое количество символов, нужно просто присвоить другое значение переменной `maxNameLength`, и весь класс обновится автоматически.

Необъяснимые константные значения, как, например, число 20 в предыдущей версии метода setName( ), называются «магическими значениями», поскольку они играют некую важную роль, однако их назначение неочевидно. Избегайте использования магических значений в своем коде. В большинстве случаев статические переменные могут применяться для хранения значений, которые в противном случае будут «магическими».

Статические переменные зачастую используются для хранения установочных параметров, значения которых не должны изменяться после запуска программы. Чтобы исключить возможность изменения значения переменной, ее нужно объявить константой, о чем и пойдет речь в следующем разделе.

Константы

Константа — это переменная экземпляра, статическая или локальная переменная, значение которой после инициализации остается постоянным вплоть до завершения программы. Для создания константы применяется стандартный синтаксис описания переменной, однако вместо ключевого слова `var` используется ключевое слово `const`. По соглашению имена констант полностью состоят из прописных букв. Чтобы создать константную статическую переменную, прямо в теле класса можно использовать следующий обобщенный код:

```
static const ИДЕНТИФИКАТОР = значение
```

Для создания константной переменной экземпляра прямо в теле класса можно указать следующий обобщенный код:

```
const ИДЕНТИФИКАТОР = значение
```

Чтобы создать константную локальную переменную, в теле метода или функции можно использовать следующий обобщенный код:

```
const ИДЕНТИФИКАТОР = значение
```

В трех предыдущих примерах кода *ИДЕНТИФИКАТОР* обозначает имя константы, а *значение* — начальное значение переменной. В случае с константными статическими и константными локальными переменными, после того как значение *значение*, указанное в инициализаторе переменной, будет присвоено переменной, изменить ее будет невозможно.

При работе с константными переменными экземпляра, если программа откомпилирована в строгом режиме, после того как значение *значение*, указанное в инициализаторе переменной, будет присвоено переменной, изменить ее будет невозможно. Если программа откомпилирована в стандартном режиме, то после того,

как значение *значение*, указанное в инициализаторе переменной, будет присвоено переменной, изменить ее можно будет внутри функции конструктора того класса, в котором она объявлена, — после этого изменить значение переменной будет невозможно (различия между строгим и стандартным режимами компиляции будут рассмотрены в гл. 7).

Константы обычно используются для создания статических переменных, чьи фиксированные значения описывают варианты определенной настройки программы. Предположим, мы создаем программу будильника для ежедневной подачи сигнала. Подача сигнала может происходить в трех режимах: визуальном (моргающий значок), звуковом (зуммер) или визуальном и звуковом одновременно. Будильник представлен классом с именем `AlarmClock`. Для представления трех режимов подачи сигнала в классе `AlarmClock` определены три константные статические переменные: `MODE_VISUAL`, `MODE_AUDIO` и `MODE_BOTH`. Каждой константе присвоено числовое значение, определяющее соответствующий режим. Режим 1 считается визуальным, режим 2 — звуковым, а режим 3 — визуальным и звуковым одновременно. В следующем примере кода продемонстрированы описания констант режимов:

```
public class AlarmClock {
  public static const MODE_VISUAL = 1;
  public static const MODE_AUDIO  = 2;
  public static const MODE_BOTH   = 3;
}
```

В классе `AlarmClock` определена переменная экземпляра `mode`, которая позволяет хранить информацию о выбранном режиме для каждого экземпляра этого класса. Чтобы задать режим для объекта `AlarmClock`, необходимо присвоить одно из константных значений режимов (`1`, `2` или `3`) переменной экземпляра `mode`. Следующий код устанавливает звуковой режим (режим 2) в качестве режима по умолчанию для новых объектов `AlarmClock`:

```
public class AlarmClock {
  public static const MODE_VISUAL = 1;
  public static const MODE_AUDIO  = 2;
  public static const MODE_BOTH   = 3;

  private var mode = AlarmClock.MODE_AUDIO;
}
```

Когда наступает время подачи сигнала, объект `AlarmClock` выполняет соответствующее действие в зависимости от текущего режима. Следующий код демонстрирует один из способов того, как объект `AlarmClock` мог бы использовать константы режимов для выбора подходящего действия:

```
public class AlarmClock {
  public static const MODE_VISUAL = 1;
  public static const MODE_AUDIO  = 2;
  public static const MODE_BOTH   = 3;

  private var mode = AlarmClock.MODE_AUDIO;

  private function signalAlarm ( ) {
    if (mode == MODE_VISUAL) {
```

```
      // Отобразить значок
    } else if (mode == MODE_AUDIO) {
      // Воспроизвести звук
    } else if (mode == MODE_BOTH) {
      // Отобразить значок и воспроизвести звук
    }
  }
}
```

Обратите внимание, что в предыдущем коде использование констант режимов с технической точки зрения не является обязательным. Собственно говоря, мы могли бы решить эту задачу с помощью константных числовых значений (магических значений). Тем не менее при использовании констант назначение числовых значений становится гораздо более понятным. Для сравнения, следующий код демонстрирует класс `AlarmClock`, реализованный без констант. Обратите внимание, что без комментариев в коде определить назначение каждого из трех значений режимов было бы достаточно сложно:

```
public class AlarmClock {
  private var mode = 2;

  private function signalAlarm ( ) {
    if (mode == 1) {
      // Отобразить значок
    } else if (mode == 2) {
      // Воспроизвести звук
    } else if (mode == 3) {
      // Отобразить значок и воспроизвести звук
    }
  }
}
```

Теперь познакомимся с партнерами статических переменных — статическими методами.

Статические методы

В предыдущем разделе мы узнали, что статические переменные используются для хранения информации, которая относится ко всему классу. Подобным образом *статические методы* описывают функциональность, относящуюся ко всему классу, а не к отдельному экземпляру этого класса. Например, в состав API среды выполнения Flash входит класс `Point`, представляющий точку с координатами по оси X и по оси Y в декартовой системе координат. В классе `Point` определен статический метод `polar( )`, который позволяет получить объект `Point` по заданной точке в полярной системе координат (то есть по расстоянию и углу). Преобразование точки в полярной системе координат в точку в декартовой системе координат является общей операцией, относящейся к точкам в декартовом пространстве вообще, а не к отдельному объекту `Point`. Именно поэтому данный метод определен как статический.

Как и методы экземпляра, статические методы создаются описанием функций внутри описаний классов, однако описания статических методов должны также включать атрибут `static`, как показано в следующем обобщенном коде:

```
class НекийКласс {
  static function имяМетода (идентификатор1 = значение1,
                             идентификатор2 = значение2,
                             ...
                             идентификаторn = значениеn) {
  }
}
```

Как и в случае с методами экземпляра, управлять доступностью статических методов в программе можно с помощью модификаторов управления доступом. В описаниях статических методов применяются те же модификаторы управления доступом, что и в описаниях методов экземпляра: `public`, `internal`, `protected` и `private`. Если при описании метода не указан никакой модификатор доступа, то используется модификатор `internal` (доступ внутри пакета). При описании статического метода модификатор доступа обычно размещается перед атрибутом `static`, как показано в следующем обобщенном коде:

```
class НекийКласс {
  public static function имяМетода (идентификатор1 = значение1,
                                    идентификатор2 = значение2
                                    ...
                                    идентификаторn = значениеn) {
  }
}
```

Для вызова статического метода используется следующий обобщенный код:

```
НекийКласс.имяМетода(значение1, значение2...значениеn)
```

В предыдущем коде *НекийКласс* обозначает класс, в котором определен статический метод, *имяМетода* — это имя статического метода, а *значение1, значение2... значениеn* — список, состоящий из нуля или более аргументов метода. Внутри класса, в котором определен данный метод, имя *имяМетода* может быть использовано самостоятельно (без лидирующего имени класса и точки). Например, в классе `A`, в котором определен статический метод `m`, выражение `A.m( )` идентично выражению `m( )`. Тем не менее, чтобы различать статические методы и методы экземпляра, многие разработчики (это относится и к примерам данной книги) включают лидирующее имя класса даже в тех случаях, когда его использование не является обязательным.

Некоторые классы существуют только ради определения статических методов, объединяя связанную функциональность, однако экземпляры таких классов никогда не создаются. Например, собственный класс `Mouse` существует только ради определения статических методов `show( )` и `hide( )` (используются для отображения или скрытия указателя мыши). Обращение к этим статическим методам происходит непосредственно через класс `Mouse` (как, например, `Mouse.hide( )`), а не через экземпляр данного класса. Объекты класса `Mouse` никогда не создаются.

У статических методов есть два ограничения, которые отсутствуют у методов экземпляра. Во-первых, в методе класса нельзя использовать ключевое слово `this`.

Во-вторых, статический метод не может обращаться к переменным и методам экземпляра класса, в котором он определен (в отличие от методов экземпляра, которые, помимо переменных и других методов экземпляра, могут также обращаться к статическим переменным и статическим методам).

В целом, статические методы по сравнению со статическими переменными используются не часто. В нашей программе по созданию виртуального зоопарка статические методы не применяются вообще. Чтобы продемонстрировать применение статических методов на практике, вернемся к функции проверки адресов электронной почты, описанной в гл. 2. В этой функции мы создали цикл, который позволяет определить наличие или отсутствие символа @ в указанном адресе электронной почты. Теперь представим, что в результате активного развития нашего приложения было решено создать служебный класс для работы со строками. Назовем его `StringUtils`. Класс `StringUtils` не будет использоваться для создания объектов; он просто представляет коллекцию статических методов. В качестве примера мы определим один статический метод `contains( )`, возвращающий значение типа `Boolean` — это значение определяет, содержит ли указанная строка выбранный символ. Рассмотрим код:

```
public class StringUtils {
  public function contains (string, character) {
    for (var i:int = 0; i <= string.length; i++) {
      if (string.charAt(i) == character) {
        return true;
      }
    }
    return false;
  }
}
```

В следующем примере кода показано, как наше приложение могло бы использовать метод `contains( )`, чтобы проверить, содержит ли указанный адрес электронной почты символ @:

```
StringUtils.contains("me@moock.org", "@");
```

Конечно, в реальном приложении адрес электронной почты вводил бы сам пользователь и метод `contains( )` определял бы допустимость отправки формы на сервер. Следующий код демонстрирует более реалистичную ситуацию:

```
if (StringUtils.contains(userEmail, "@")) {
  // Этот код отправлял бы форму на сервер
} else {
  // Этот код отображал бы сообщение "Неправильные данные" для пользователя
}
```

Помимо статических методов, которые создаются вручную, среда выполнения Flash автоматически создает для каждого класса один статический метод, называемый инициализатором класса. Познакомимся с ним поближе.

Инициализатор класса. После определения класса на этапе выполнения программы среда Flash автоматически создает и исполняет метод, называемый *инициализатором класса*. В нем среда Flash размещает все инициализаторы статических

переменных данного класса и весь код уровня класса, не относящийся к описаниям переменных или методов.

Инициализатор класса предоставляет возможность выполнять однократные задачи настройки сразу после определения класса, вызывая методы или обращаясь к переменным, являющимся внешними по отношению к текущему классу. Предположим, что мы создаем приложение для чтения электронной почты и хотим, чтобы внешний вид этого приложения соответствовал графическому стилю операционной системы. Чтобы определить, какая графическая тема должна быть использована в почтовом клиенте, в инициализаторе основного класса приложения `MailReader` проверяется текущая операционная система и присваивается соответствующее значение статической переменной `theme`. Переменная `theme` хранит информацию о графической теме, используемой в данном приложении. В следующем коде продемонстрирован инициализатор для класса `MailReader`. Для проверки операционной системы в классе `MailReader` используется статическая переменная `os`, определенная в собственном классе `flash.system.Capabilities`.

```
package {
  import flash.system.*;

  public class MailReader {
    static var theme;
    if (Capabilities.os == "MacOS") {
      theme = "MAC";
    } else if (Capabilities.os == "Linux") {
      theme = "LINUX";
    } else {
      theme = "WINDOWS";
    }
  }
}
```

Код, размещаемый в инициализаторе класса, выполняется в режиме интерпретации, и JIT-компилятор не компилирует его. Динамически откомпилированный код выполняется гораздо быстрее интерпретируемого кода, поэтому в тех случаях, когда на первом месте стоит вопрос производительности, код, интенсивно использующий ресурсы процессора, следует выносить за пределы инициализатора класса.

Объекты Class

Как мы уже знаем, обращение к каждому статическому методу и к каждой статической переменной осуществляется через класс, в котором они определены. Например, для обращения к статической переменной `maxCalories`, заданной в классе `VirtualPet`, мы применяем следующий код:

```
VirtualPet.maxCalories
```

Использование имени класса `VirtualPet` в предыдущем коде является не просто особенностью синтаксиса; на самом деле имя класса `VirtualPet` ссылается

на объект, в котором определена переменная `maxCalories`. Объект, на который ссылается имя класса `VirtualPet`, является автоматически создаваемым экземпляром собственного класса `Class`.

Во время выполнения программы каждый класс в языке ActionScript представляется экземпляром класса `Class`. С точки зрения программиста, объекты `Class` используются в основном для доступа к статическим переменным и методам класса. Тем не менее, как и любые другие объекты, они являются значениями которые могут быть присвоены переменным, переданы или возвращены из методов и функций. Например, в следующей модифицированной версии нашего класса `VirtualPet` переменной `vp` присваивается объект `Class`, представляющий класс `VirtualPet`, после чего эта переменная используется для создания объекта `VirtualPet`:

```
package zoo {
  public class VirtualZoo {
    private var pet;

    public function VirtualZoo ( ) {
      var vp = VirtualPet;
      pet = new vp("Stan");
    }
  }
}
```

Описанная методика применяется в тех случаях, когда один SWF-файл желает обратиться к классам другого SWF-файла или когда мультимедийные элементы (например, изображения или шрифты) размещаются в другом SWF-файле. Обе описанные ситуации будут рассмотрены в части II.

Наше знакомство со статическими переменными и методами подошло к концу. Но перед тем, как перейти к следующей главе, сравним рассмотренные понятия с понятиями, применяемыми в языках C++ и Java.

Сравнение с терминологиями языков C++ и Java

Концепции переменных экземпляра, методов экземпляра, статических переменных и статических методов присутствуют в большинстве объектно-ориентированных языков программирования. Для сравнения в табл. 4.1 перечислены эквивалентные понятия, применяемые в языках Java и C++.

Таблица 4.1. Сравнение терминологий

ActionScript	Java	C++
Переменная экземпляра	Поле или переменная экземпляра	Член данных
Метод экземпляра	Метод	Функция-член
Статическая переменная	Переменная класса	Статический член данных
Статический метод	Метод класса	Статическая функция-член

К функциям

Как мы уже знаем, методы экземпляра определяют поведение, которое относится к отдельному объекту, а статические методы — поведение, относящееся к отдельному классу. В следующей главе мы познакомимся с *функциями*, которые определяют независимые поведения, не относящиеся ни к объектам, ни к классам.

Функции

Функция, или *замыкание функции*, — это дискретный набор инструкций, выполняющих определенную задачу независимо от других классов или объектов. Для описания и использования замыканий функций применяется такой же базовый синтаксис, как и для методов экземпляра и статических методов. Функции описываются с помощью ключевого слова `function` и вызываются с помощью оператора круглых скобок, при необходимости функции могут возвращать значение, а внутри тела могут определяться локальные переменные. Тем не менее, в отличие от методов экземпляра (которые всегда связаны с объектом) и статических методов (которые всегда связаны с классом), замыкания функций создаются и используются самостоятельно либо в виде подзадачи в методе, либо в виде полезной процедуры, доступной в пакете или в любом месте программы.

Говоря на строгом профессиональном жаргоне спецификации языка ActionScript 3.0, и замыкания функций, и методы являются разновидностями функций, при этом термин «функция» вообще относится к вызываемому объекту, представляющему собой набор инструкций. Таким образом, замыкание функции является функцией, не связанной с объектом или классом, а метод — функцией, связанной с объектом (в случае с методами экземпляра) или классом (в случае со статическими методами). Тем не менее в лексиконе программистов и в подавляющем большинстве документации вместо термина «замыкание функции» используется его сокращенный вариант — «функция». Если вы не читаете спецификацию языка ActionScript 3.0 или текст, в котором указано обратное, можете смело предполагать, что функция означает замыкание функции. В оставшейся части этой книги термин «функция» обозначает замыкание функции, если не указано другое.

Для создания функции используется следующий обобщенный код, размещаемый в одном из перечисленных мест: внутри метода, непосредственно внутри описания пакета, непосредственно за пределами описания пакета или внутри другой функции. Обратите внимание, что используемый для описания функции код идентичен коду, применяемому для описания обычного метода экземпляра. На самом деле, если следующий код размещается непосредственно внутри тела класса, создается метод экземпляра, а не функция.

```
function идентификатор (параметр1, параметр2... параметрn) {
}
```

В этом коде *идентификатор* обозначает имя функции, а *параметр1*, *параметр2*... *параметрn* — необязательный список параметров функции, которые используются точно так же, как и параметры метода, рассмотренные в гл. 1. Фигурные скобки, следующие за списком параметров, определяют начало и конец тела функции, содержащего инструкции, выполняемые при ее вызове.

Для вызова функции применяется следующий обобщенный код:

```
функция (значение1, значение2... значениеn)
```

В данном коде *функция* обозначает имя вызываемой функции, а *значение1, значение2... значениеn* — список аргументов, которые связаны по порядку с параметрами функции *функция*.

Функции уровня пакета

Чтобы функция была доступна в пакете или в любой точке программы, ее описание должно размещаться непосредственно внутри тела пакета. Чтобы ограничить доступ к функции только тем пакетом, в котором она описана, перед описанием функции нужно указать модификатор управления доступом `internal`, как показано в следующем коде:

```
package имяПакета {
  internal function идентификатор ( ) {
  }
}
```

Чтобы функция была доступна в любой точке программы, перед описанием функции нужно указать модификатор управления доступом `public`, как показано в следующем коде:

```
package имяПакета {
  public function идентификатор ( ) {
  }
}
```

Если никакой модификатор управления доступом не указан, то компилятор языка ActionScript автоматически использует модификатор `internal`.

Компиляторы, разработанные корпорацией Adobe, налагают два требования, затрагивающие функции уровня пакета, на исходные файлы (AS-файлы) программ, написанных на языке ActionScript.

Каждый исходный файл (AS-файл) программы должен содержать только одно описание, видимое извне. Это может быть описание класса, переменной, функции, интерфейса или пространства имен, определенное внутри тела пакета либо с помощью модификатора internal, либо с помощью модификатора public.

Имя каждого исходного файла программы должно совпадать с именем единственного видимого извне описания, которое содержится в этом файле.

Таким образом, если теоретически язык ActionScript и не налагает никаких ограничений на функции уровня пакета, то на практике компиляторы, разработанные корпорацией Adobe, требуют, чтобы каждая функция уровня пакета была определена либо с помощью модификатора internal, либо с помощью модификатора public и размещена в отдельном AS-файле, имя которого должно совпадать с именем функции. Дополнительную информацию об ограничениях, налагаемых компиляторами, можно найти в разд. «Ограничения компиляторов» гл. 7.

Следующий код создает функцию уровня пакета `isMac( )`, возвращающую значение типа `Boolean`, которое указывает, является ли Macintosh текущей операционной

системой. Поскольку в описании функции `isMac( )` указан модификатор управления доступом `internal`, эта функция будет доступна только внутри пакета `utilities`. Как было отмечено ранее, если для компиляции используется один из компиляторов, разработанных корпорацией Adobe, следующий код необходимо поместить в отдельный AS-файл с именем `isMac.as`.

```
package utilities {
  import flash.system.*;

  internal function isMac ( ) {
    return Capabilities.os == "MacOS";
  }
}
```

Чтобы функция `isMac( )` была доступна за пределами пакета `utilities`, необходимо заменить модификатор `internal` модификатором `public`, как показано в следующем коде:

```
package utilities {
  import flash.system.*;

  public function isMac ( ) {
    return Capabilities.os == "MacOS";
  }
}
```

Тем не менее, чтобы иметь возможность использовать функцию `isMac( )` за пределами пакета `utilities`, ее сначала необходимо импортировать. Предположим, что функция `isMac( )` является частью большой программы с классом `Welcome`, входящим в пакет `setup`. Чтобы воспользоваться этой функцией в классе `Welcome`, в исходный файл этого класса должна быть импортирована функция `utilities.isMac( )`, как показано в следующем коде:

```
package setup {
  // Импортировать функцию isMac( ), чтобы ее можно было использовать
  // внутри тела этого пакета
  import utilities.isMac;

  public class Welcome {
    public function Welcome ( ) {
      // Воспользоваться функцией isMac( )
      if (isMac( )) {
        // Выполнить специфические для Macintosh действия
      }
    }
  }
}
```

Глобальные функции. Функции, определенные на уровне пакета и размещаемые внутри пакета без имени, называются *глобальными*, поскольку обращаться к ним можно глобально, из любой точки программы без необходимости использования оператора `import`. Например, следующий код определяет глобальную функцию `isLinux( )`. Поскольку функция `isLinux( )` находится внутри пакета без имени, к ней можно обращаться из любого места кода данной программы.

```
package {
  import flash.system.*;

  public function isLinux ( ) {
    return Capabilities.os == "Linux";
  }
}
```

Следующий код демонстрирует модифицированную версию класса `Welcome` из предыдущего раздела, в котором вместо функции `isMac( )` используется функция `isLinux( )`. Обратите внимание, что перед применением функцию импортировать не нужно.

```
package setup {
  public class Welcome {
    public function Welcome ( ) {
      // Воспользоваться функцией isLinux( )
      if (isLinux( )) {
        // Выполнить специфические для Linux действия
      }
    }
  }
}
```

Многие функции уровня пакета и глобальные функции являются собственными для каждой отдельно взятой среды выполнения Flash. Список доступных функций можно найти в документации корпорации Adobe по интересуемой среде выполнения Flash.

Пожалуй, наиболее используемой собственной глобальной функцией является функция `trace( )`, имеющая следующий обобщенный вид:

```
trace (аргумент1, аргумент2... аргументn)
```

Функция `trace( )` представляет собой простейший инструмент для поиска ошибок в программе (то есть для отладки). Она позволяет выводить указанные аргументы либо в окно среды разработки, либо в файл журнала. Например, при выполнении программы в тестовом режиме в среде разработки Flash с помощью команды **Control ▸ Test Movie** (Управление ▸ Проверка фильма) результаты всех вызовов функции `trace( )` появятся в окне **Output** (Вывод). Подобным образом при выполнении программы в тестовом режиме в приложении Flex Builder с помощью команды **Run ▸ Debug** (Выполнить ▸ Отладка) результаты всех вызовов функции `trace( )` появятся в окне **Console** (Консоль). Информацию по конфигурированию отладочной версии приложения Flash Player для вывода аргументов функции `trace( )` в текстовый файл можно найти по адресу **http://livedocs.macromedia.com/flex/2/docs/00001531.html**.

Вложенные функции

Когда описание функции размещается внутри метода или другой функции, создается *вложенная функция*, которая доступна для использования только внутри содержащего ее метода или функции. По существу, вложенная функция описывает многократно используемую подзадачу, полностью принадлежащую тому методу

или функции, в которых она определена. Следующий код демонстрирует базовый пример вложенной функции b( ), описанной внутри метода экземпляра a( ). Вложенная функция b( ) может быть использована только внутри метода a( ); за пределами метода a( ) функция b( ) недоступна.

```
// Описание метода a( )
public function a ( ) {
  // Вызов вложенной функции b( )
  b( );

  // Описание вложенной функции b( )
  function b ( ) {
    // Здесь должно размещаться тело функции
  }
}
```

В предыдущем коде стоит обратить внимание на то, что вложенная функция может вызываться в любом месте содержащего ее метода, даже *до* описания этой функции. Обращение к переменной или функции до того, как эта переменная или функция будут описаны, называется *опережающим обращением*. Помимо этого стоит отметить, что для вложенных функций невозможно использовать модификаторы управления доступом (public, internal и т. д.).

Следующий код демонстрирует более реальный пример метода, содержащего вложенную функцию. Метод getRandomPoint( ) возвращает объект типа Point, который представляет произвольную точку в заданном прямоугольнике. Чтобы получить произвольную точку, этот метод использует вложенную функцию getRandomInteger( ) для вычисления случайных координат по осям X и Y. Обратите внимание, что в функции getRandomInteger( ) применяются собственные статические методы Math.random( ) и Math.floor( ). Первый метод возвращает случайное число с плавающей запятой, большее либо равное 0, но меньшее 1. Второй метод устраняет дробную часть числа с плавающей запятой. Дополнительную информацию по статическим методам класса Math можно найти в справочнике по языку ActionScript корпорации Adobe.

```
public function getRandomPoint (rectangle) {
  var randomX = getRandomInteger(rectangle.left, rectangle.right);
  var randomY = getRandomInteger(rectangle.top, rectangle.bottom);

  return new Point(randomX, randomY);

  function getRandomInteger (min, max) {
    return min + Math.floor(Math.random( )*(max+1 - min));
  }
}
```

Функции уровня исходного файла

Если описание функции размещается на верхнем уровне исходного файла за пределами тела пакета, то будет создана функция, доступная только внутри данно-

го исходного файла. В следующем примере представлено содержимое исходного файла `A.as`, включающее описание пакета, описание класса и описание функции уровня исходного файла. Поскольку функция определена за пределами оператора блока пакета, она может быть использована в любом месте кода внутри файла `A.as`, однако вне этого файла данная функция будет недоступна.

```
package {
  // Функцию f( ) можно использовать здесь
  class A {
    // Функцию f( ) можно использовать здесь
    public function A ( ) {
      // Функцию f( ) можно использовать здесь
    }
  }
}

// Функцию f( ) можно использовать здесь

function f ( ) {
}
```

В предыдущем коде обратите внимание на то, что описание функции `f( )` не содержит и не должно содержать никаких модификаторов управления доступом (`public`, `internal` и т. д.).

Модификаторы управления доступом не должны применяться при описании функций уровня исходного файла.

Функции уровня исходного файла иногда используются для определения дополнительных модулей, относящихся к одному классу (как, например, к классу `A` в предыдущем коде). Тем не менее, поскольку дополнительные модули для класса можно определять и с помощью закрытых статических методов, функции уровня исходного файла редко используются в реальных программах на языке ActionScript.

Доступ к описаниям из функции

Место размещения функции в программе влияет на возможность обращения к описаниям этой программы из данной функции (то есть к классам, переменным, методам, пространствам имен, интерфейсам и другим функциям). Подробное описание того, к чему можно и к чему нельзя обращаться из кода функций, можно найти в разд. «Область видимости функций» гл. 16.

Обратите внимание, однако, что внутри замыкания функции ключевое слово `this` всегда ссылается на глобальный объект, независимо от места определения этой функции. Чтобы обратиться к текущему объекту внутри вложенной функции в методе экземпляра, присвойте ключевое слово `this` переменной, как показано в следующем коде:

```
public function m ( ) {
  var currentObject = this:

  function f ( ) {
    // Здесь можно обращаться к переменной currentObject
    trace(currentObject): // Отображает объект, через который был
                          // вызван метод m( )
  }
}
```

Функции в качестве значений

В языке ActionScript любая функция представляется экземпляром класса `Function`. По существу, функция может быть присвоена переменной, передана в функцию или возвращена из нее точно так же, как и любое другое значение. Например, в следующем коде описывается функция `a ( )`, после чего она присваивается переменной `b`. Обратите внимание, что оператор круглых скобок `( )` опущен; в противном случае переменной `b` было бы просто присвоено возвращаемое значение функции `a ( )`.

```
function a ( ) {
}
var b = a;
```

Как только функция будет присвоена переменной, ее можно вызывать через эту переменную с помощью стандартного оператора круглых скобок `( )`. Например, в следующем коде функция `a ( )` вызывается через переменную `b`:

```
b( );
```

Функции-значения обычно используются при создании динамических классов и объектов, которые рассматриваются в разд. «Динамическое добавление нового поведения в экземпляр» и «Использование объектов-прототипов для дополнения классов» гл. 15.

Синтаксис литералов функций

Экземпляры класса `Function`, как и многих предопределенных классов языка ActionScript, можно создавать с помощью синтаксиса литералов. Он практически ничем не отличается от синтаксиса стандартного объявления функций, за исключением отсутствующего имени функции. Вот его общий вид:

```
function (параметр1, параметр2... параметрn) {
}
```

Здесь *параметр1, параметр2... параметрn* — это необязательный список параметров.

Чтобы воспользоваться функцией, описанной с помощью литерала функции, за пределами выражения, в котором встречается данный литерал, мы можем присвоить эту функцию переменной, как показано в следующем коде:

```
var некаяПеременная = function (параметр1, параметр2... параметрn) {
}
```

После этого вызывать функцию можно через данную переменную, как показано ниже:

```
некаяПеременная (аргумент1, аргумент2... аргументn)
```

Например, в следующем коде для создания функции, которая возводит число в квадрат, используется литерал функции, а созданная функция присваивается переменной `square`:

```
var square = function (n) {
  return n * n;
}
```

Для вызова функции из предыдущего примера применяется следующий код:

```
// Возводит в квадрат число 5 и возвращает результат
square(5)
```

Литералы функций иногда применяются совместно с собственной функцией `flash.utils.setInterval( )`, которая имеет следующий вид:

```
setInterval(ФункцияИлиМетод, задержка)
```

Функция `setInterval( )` создает *интервал*, используемый для автоматического вызова указанной функции или метода (*ФункцияИлиМетод*) каждые *задержка* миллисекунд. Каждому создаваемому интервалу присваивается число, возвращаемое функцией `setInterval( )` и называемое *идентификатором интервала*. Идентификатор интервала может быть присвоен переменной, что в дальнейшем позволит удалить соответствующий интервал вызовом функции `clearInterval( )`, как показано в следующем примере кода:

```
// Создать интервал, выполняющий функцию doSomething( ) каждые
// 50 миллисекунд. Присвоить возвращаемый идентификатор интервала
// переменной intervalID.
var intervalID = setInterval(doSomething, 50);

// ...Далее в программе прекратить автоматический вызов функции
// doSomething( )
clearInterval(intervalID);
```

Класс Timer, рассматриваемый в разд. «Пользовательские события» гл. 12 и разд. «Создание анимации с использованием события TimerEvent.TIMER» гл. 24, предоставляет гораздо более широкие возможности управления периодическим выполнением функций или методов.

В следующем коде продемонстрирован простейший класс `Clock`, который выводит отладочное сообщение `"Tick!"` один раз в секунду. Обратите внимание на использование литерала функции и собственных функций `setInterval( )` и `trace( )`.

```
package {
  import flash.utils.setInterval;

  public class Clock {
    public function Clock ( ) {
      // Выполнять литерал функции один раз в секунду
```

```
      setInterval(function ( ) {
        trace("Tick!");
      }, 1000);
    }
  }
}
```

Стоит отметить, что литералы функций используются исключительно ради удобства. Предыдущий код можно легко переписать с помощью вложенной функции, как показано далее:

```
package {
  import flash.utils.setInterval;

  public class Clock {
    public function Clock ( ) {
      // Выполнять функцию tick( ) один раз в секунду
      setInterval(tick, 1000);

      function tick ( ):void {
        trace("Tick!");
      }
    }
  }
}
```

Можно утверждать, что версия класса `Clock`, реализованная с помощью вложенной функции, легче для чтения. Литералы широко используются при связывании функций с динамическими переменными экземпляра — этот вопрос рассматривается в разд. «Динамическое добавление нового поведения в экземпляр» гл. 15.

Рекурсивные функции

Рекурсивная функция — это функция, вызывающая саму себя. Следующий код демонстрирует простейший пример рекурсии. Всякий раз при выполнении функции `trouble( )` происходит ее повторный вызов:

```
function trouble ( ) {
  trouble( );
}
```

Если рекурсивная функция безусловно вызывает саму себя, как функция `trouble( )` в нашем случае, возникает *бесконечная рекурсия* (то есть такое состояние, когда функция не прекратит вызывать саму себя никогда). Без проверки условия бесконечная рекурсия теоретически привела бы к тому, что программа оказалась бы в бесконечном циклическом процессе выполнения функции. На практике, чтобы избежать подобной ситуации, рекурсивные функции вызывают сами себя только при выполнении заданного условия. Одним из классических примеров применения рекурсии является вычисление *факториала*, который представляет собой произведение всех целых положительных чисел, меньших либо равных данному числу. Например, факториал числа 3 (на математическом языке это записывается как 3!) равен $3 \times 2 \times 1$, то есть 6. Факто-

риал числа 5 равен $5 \times 4 \times 3 \times 2 \times 1$, то есть 120. В листинге 5.1 продемонстрирована функция для вычисления факториала числа, реализованная с помощью рекурсии.

Листинг 5.1. Вычисление факториалов с помощью рекурсии

```
function factorial (n) {
  if (n < 0) {
    return;  // Неправильное число, завершаем работу
  } else if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n-1);
  }
}

// Использование в программе:
factorial(3); // Возвращает: 6
factorial(5); // Возвращает: 120
```

Вычислить факториал можно и с помощью цикла, полностью заменяющего рекурсию, как показано в листинге 5.2.

Листинг 5.2. Вычисление факториала без использования рекурсии

```
function factorial (n) {
  if (n < 0) {
    return;  // Неправильное число, завершаем работу
  } else {
    var result = 1;
    for (var i = 1; i <= n; i++) {
      result = result * i;
    }
    return result;
  }
}
```

В листингах 5.1 и 5.2 представлены два различных способа решения одной задачи. Рекурсивный подход гласит: «Факториал числа 6 равен числу 6, умноженному на факториал числа 5. Факториал числа 5 равен числу 5, умноженному на факториал числа 4...» и т. д. Нерекурсивный подход предполагает выполнение цикла для чисел от 1 до *n*, где происходит перемножение этих чисел, в результате чего получается одно большое число.

Использование рекурсивных функций считается элегантным подходом, поскольку оно обеспечивает простое решение сложных задач — циклический вызов одной и той же функции. Тем не менее повторяющиеся вызовы функции являются менее эффективными, чем итерации цикла. Нерекурсивный подход, применяемый для вычисления факториалов, во много раз эффективнее рекурсивного. Кроме того, нерекурсивный подход исключает достижение максимальной глубины рекурсии, равной по умолчанию 1000, которая, однако, может быть изменена аргументом компилятора `default-script-limits`.

В гл. 18 вы узнаете, что рекурсия иногда используется для обработки содержимого XML-документов с иерархической структурой.

Использование функций в программе «Зоопарк»

Применим наши новые знания функций к программе по созданию виртуального зоопарка (чтобы освежить в памяти существующий код программы, обратитесь к листингу 4.1, содержащему код класса `VirtualPet`).

Если помните, в последней версии нашей программы по созданию виртуального зоопарка животные могли только употреблять пищу (то есть накапливать калории), но не переваривать ее (то есть терять калории). Чтобы наши животные могли переваривать пищу, мы добавим новый метод `digest( )` в класс `VirtualPet`. Метод `digest( )` будет вычитать калории из того объекта `VirtualPet`, над которым осуществляется вызов данного метода. Чтобы имитировать переваривание пищи в течение времени, мы создадим интервал, используемый для вызова метода `digest( )` один раз в секунду. Количество калорий, потребляемых при каждом вызове метода `digest( )`, будет определяться новой статической переменной `caloriesPerSecond`. Присвоим переменной `caloriesPerSecond` значение `100`, позволяя животному прожить максимум 20 секунд на «полный желудок».

Следующий код демонстрирует описание переменной `caloriesPerSecond`:

```
private static var caloriesPerSecond = 100;
```

Далее представлено описание метода `digest( )`. Обратите внимание, что, поскольку переваривание пищи является внутренним процессом, метод `digest( )` объявлен с использованием модификатора управления доступом `private`.

```
private function digest ( ) {
  currentCalories -= VirtualPet.caloriesPerSecond;
}
```

Чтобы создать интервал, вызывающий метод `digest( )` один раз в секунду, воспользуемся собственной функцией `setInterval( )`. Каждое животное должно приступать к перевариванию пищи сразу после его создания, поэтому поместим вызов функции `setInterval( )` в метод-конструктор класса `VirtualPet`. Кроме того, сохраним идентификатор интервала, возвращаемый функцией `setInterval( )`, в новой переменной экземпляра `digestIntervalID`, чтобы в дальнейшем при необходимости можно было удалить созданный интервал.

Следующий код демонстрирует описание переменной `digestIntervalID`:

```
private var digestIntervalID;
```

Измененный конструктор класса `VirtualPet` выглядит так:

```
public function VirtualPet (name) {
  setName(name);

  // Вызывать метод digest( ) один раз в секунду
  digestIntervalID = setInterval(digest, 1000);
}
```

Теперь, когда объекты класса `VirtualPet` могут переваривать пищу, воспользуемся глобальной функцией `trace( )`, чтобы сообщать о текущем состоянии каждого животного в процессе отладки. Будет выдаваться сообщение о состоянии всякий

раз при выполнении методов `digest( )` или `eat( )`. Обновленная версия метода `digest( )` выглядит следующим образом:

```
private function digest ( ) {
  currentCalories -= VirtualPet.caloriesPerSecond;

  trace(getName( ) + " digested some food. It now has " + currentCalories
        + " calories remaining.");
}
```

Обновленная версия метода `eat( )`:

```
public function eat (numberOfCalories) {
  var newCurrentCalories = currentCalories + numberOfCalories;
  if (newCurrentCalories > VirtualPet.maxCalories) {
    currentCalories = VirtualPet.maxCalories;
  } else {
    currentCalories = newCurrentCalories;
  }

  trace(getName( ) + " ate some food. It now has " + currentCalories
        + " calories remaining.");
}
```

Если бы мы запустили нашу программу «Зоопарк» прямо сейчас, то увидели бы следующие сообщения в окне **Output** (Вывод) (среда разработки Flash) или **Console** (Консоль) (Flex Builder):

```
Stan digested some food. It now has 900 calories remaining.
Stan digested some food. It now has 800 calories remaining.
Stan digested some food. It now has 700 calories remaining.
Stan digested some food. It now has 600 calories remaining.
Stan digested some food. It now has 500 calories remaining.
Stan digested some food. It now has 400 calories remaining.
Stan digested some food. It now has 300 calories remaining.
Stan digested some food. It now has 200 calories remaining.
Stan digested some food. It now has 100 calories remaining.
Stan digested some food. It now has 0 calories remaining.
Stan digested some food. It now has -100 calories remaining.
Stan digested some food. It now has -200 calories remaining.
Stan digested some food. It now has -300 calories remaining.
```

Но в чем же дело? Количество калорий у животных не должно принимать отрицательных значений. Вместо этого животные должны умирать, когда значение переменной `currentCalories` достигнет `0`. В нашей программе состояние смерти будет выражаться следующим образом.

- Если значение переменной `currentCalories` равно `0`, программа будет игнорировать любые попытки увеличить значение переменной `currentCalories` путем вызова метода `eat( )`.
- Когда значение переменной `currentCalories` станет равным `0`, программа удалит интервал, с помощью которого вызывается метод `digest( )`, и отобразит сообщение о «смерти животного».

Сначала модифицируем метод `eat( )`. С поставленной задачей должен справиться простой условный оператор:

```
public function eat (numberOfCalories) {
  // Если это животное мертво,
  if (currentCalories == 0) {
    // ...завершить метод, не изменяя значение переменной
    // currentCalories
    trace(getName( ) + " is dead. You can't feed it.");
    return;
  }

  var newCurrentCalories = currentCalories + numberOfCalories;
  if (newCurrentCalories > VirtualPet.maxCalories) {
    currentCalories = VirtualPet.maxCalories;
  } else {
    currentCalories = newCurrentCalories;
  }
  trace(getName( ) + " ate some food. It now has " + currentCalories
        + " calories remaining.");
}
```

Теперь мы должны остановить процесс вызова метода `digest( )`, когда значение переменной `currentCalories` достигнет 0. Для этого воспользуемся функцией `flash.utils.clearInterval( )`:

```
private function digest ( ) {
  // Если в результате потребления очередной порции калорий значение
  // переменной currentCalories станет равным 0 или меньше...
  if (currentCalories - VirtualPet.caloriesPerSecond <= 0) {
    // ...прекратить вызов метода digest( )
    clearInterval(digestIntervalID);
    // После чего очистить желудок животного
    currentCalories = 0;
    // и сообщить о смерти животного
    trace(getName( ) + " has died.");
  } else {
    // ...иначе употребить оговоренное количество калорий
    currentCalories -= VirtualPet.caloriesPerSecond;

    // и сообщить о новом состоянии животного
    trace(getName( ) + " digested some food. It now has "
          + currentCalories + " calories remaining.");
  }
}
```

В листинге 5.3 представлен целиком весь код класса `VirtualPet`, включая все внесенные изменения.

Листинг 5.3. Класс VirtualPet

```
package zoo {
  import flash.utils.setInterval;
  import flash.utils.clearInterval;
```

```
internal class VirtualPet {
  private static var maxNameLength = 20;
  private static var maxCalories = 2000;
  private static var caloriesPerSecond = 100;

  private var petName;
  private var currentCalories = VirtualPet.maxCalories/2;
  private var digestIntervalID;

  public function VirtualPet (name) {
    setName(name);
    digestIntervalID = setInterval(digest, 1000);
  }

  public function eat (numberOfCalories) {
    if (currentCalories == 0) {
      trace(getName( ) + " is dead. You can't feed it.");
      return;
    }

    var newCurrentCalories = currentCalories + numberOfCalories;
    if (newCurrentCalories > VirtualPet.maxCalories) {
      currentCalories = VirtualPet.maxCalories;
    } else {
      currentCalories = newCurrentCalories;
    }
    trace(getName( ) + " ate some food. It now has " + currentCalories
          + " calories remaining.");
  }

  public function getHunger ( ) {
    return currentCalories / VirtualPet.maxCalories;
  }

  public function setName (newName) {
    // Если длина заданного нового имени больше maxNameLength
    // символов...
    if (newName.length > VirtualPet.maxNameLength) {
      // ...обрезать имя
      newName = newName.substr(0, VirtualPet.maxNameLength);
    } else if (newName == "") {
      // ...в противном случае, если заданное новое имя является
      // пустой строкой, завершить выполнение метода, не изменяя
      // значения переменной petName
      return;
    }

    // Присвоить новое, проверенное имя переменной petName
    petName = newName;
  }
```

```
    public function getName ( ) {
      return petName;
    }

    private function digest ( ) {
      // Если в результате потребления очередной порции калорий значение
      // переменной currentCalories животного станет равным 0 или меньше...
      if (currentCalories - VirtualPet.caloriesPerSecond <= 0) {
        // ...прекратить вызов метода digest( )
        clearInterval(digestIntervalID);
        // После чего очистить желудок животного
        currentCalories = 0;
        // и сообщить о смерти животного
        trace(getName( ) + " has died.");
      } else {
        // ...иначе употребить оговоренное количество калорий
        currentCalories -= VirtualPet.caloriesPerSecond;

        // и сообщить о новом состоянии животного
        trace(getName( ) + " digested some food. It now has "
              + currentCalories + " calories remaining.");
      }
    }
  }
}
```

Обратно к классам

Мы рассмотрели функции в языке ActionScript. В следующей главе мы вернемся к теме классов, уделив особое внимание наследованию при создания отношений между двумя или более классами. Разобравшись с вопросами наследования, мы сможем подготовить нашу программу по созданию виртуального зоопарка для компиляции и выполнения.

Наследование

В объектно-ориентированном программировании термин *«наследование»* обозначает формальное отношение между двумя или более классами, при котором один класс заимствует (или *наследует*) описания переменных и методов другого класса. С практической, или технической, точки зрения наследование просто позволяет использовать код одного класса в другом классе.

Однако наследование подразумевает нечто гораздо большее, чем повторное использование кода. Оно в равной мере является интеллектуальным и техническим инструментом и позволяет программистам концептуально представить группу классов в виде иерархических отношений. В биологии наследование представляет собой генетический процесс, при котором одно живое существо передает по наследству свои характерные черты другому существу. Вам говорят, что вы унаследовали глаза матери или нос отца, даже несмотря на то, что вы не похожи в точности ни на одного из родителей. В объектно-ориентированном программировании наследование имеет похожий смысл. Оно позволяет одному классу быть во многом подобным другому классу, но при этом обладать собственными уникальными свойствами.

Изучение этой главы мы начнем с рассмотрения синтаксиса и примеров использования наследования. Разобравшись с наследованием с практической точки зрения, мы рассмотрим его преимущества и альтернативы. И наконец, мы применим наследование к нашей программе по созданию виртуального зоопарка.

Пример наследования

Рассмотрим простой абстрактный пример, чтобы получить общее представление о том, как работает наследование (примеры практического применения наследования будут рассмотрены после того, как мы познакомимся с базовым синтаксисом). Существует класс `A` с одним методом экземпляра `m( )` и одной переменной экземпляра `v`:

```
public class A {
  public var v = 10;

  public function m ( ) {
    trace("Method m( ) was called");
  }
}
```

Как обычно, мы можем создать экземпляр класса `A`, вызвать метод `m( )` и обратиться к переменной `v` следующим образом:

```
var aInstance = new A( );
aInstance.m( );      // Выводит: Method m( ) was called
trace(aInstance.v);  // Выводит: 10
```

Пока ничего нового. Теперь добавим второй класс B, который наследует метод m( ) и переменную v класса A. Для создания отношения наследования между классами A и B используется ключевое слово extends:

```
public class B extends A {
  // Никакие методы и переменные не определены
}
```

Поскольку класс B расширяет (унаследован от) класс A, экземпляры класса B могут автоматически использовать метод m( ) и переменную v (даже несмотря на то, что в самом классе B этот метод и переменная не определены):

```
var bInstance:B = new B( );
bInstance.m( );      // Выводит: Method m( ) was called
trace(bInstance.v);  // Выводит: 10
```

При выполнении инструкции bInstance.m( ) среда Flash проверяет, определен ли метод m( ) в классе B. Не найдя метода m( ) в классе B, Flash продолжает его поиск в *суперклассе* класса B (то есть в том классе, который расширяется классом B). Среда выполнения находит метод m( ) в классе A и вызывает его над переменной bInstance.

Обратите внимание, что в самом классе B не определены никакие методы или переменные. На практике определение класса, не добавляющего ничего нового в расширяемый класс, не имеет большого смысла, поэтому, как правило, это делать не рекомендуется. В обычном же случае, помимо методов и переменных, унаследованных от класса A, класс B определял бы свои собственные методы и/или переменные. Иными словами, подкласс на самом деле представляет собой расширенный набор возможностей, доступных в его суперклассе. Подкласс обладает всем, что доступно в суперклассе, а также дополнительными возможностями. Рассмотрим новую версию класса B, унаследовавшего метод m( ) и переменную v от класса A и определяющего свой собственный метод n( ):

```
public class B extends A {
  public function n ( ) {
    trace("Method n( ) was called");
  }
}
```

Теперь экземпляры класса B могут использовать все методы и переменные не только класса B, но и его суперкласса A:

```
var bInstance = new B( );
// Вызов унаследованного метода, определенного в классе A
bInstance.m( );      // Выводит: Method m( ) was called
// Вызов метода, определенного в классе B
bInstance.n( );      // Выводит: Method n( ) was called
// Обращение к унаследованной переменной
trace(bInstance.v);  // Выводит: 10
```

В данном случае говорят, что класс B *специализирует* класс A. Класс B использует возможности класса A в качестве своей основы, добавляя свои собственные возможности или даже — как мы увидим далее — перекрывая возможности класса A измененными для собственных нужд версиями. Таким образом, в отношении наследования между двумя классами расширяемый класс (в нашем случае A) называется *базовым*, а расширяющий класс (в нашем случае B) — *производным*. Иногда для обозначения базового и производного классов также используются понятия *«предок»* и *«потомок»* соответственно.

Наследование может (а зачастую так и происходит) включать больше чем два класса. Например, даже несмотря на то, что класс B унаследован от класса A, класс B может стать базовым для другого класса. Следующий код демонстрирует третий класс C, который расширяет класс B, а также определяет новый метод o( ). Класс C может использовать все методы и переменные, определенные в нем самом и во всех его предках, то есть в его суперклассе (B) и суперклассе суперкласса (A).

```
public class C extends B {
  public function o ( ) {
    trace("Method o( ) was called");
  }
}

// Использование:
var cInstance = new C( );
// Вызов метода, унаследованного от класса A
cInstance.m( );  // Выводит: Method m( ) was called
// Вызов метода, унаследованного от класса B
cInstance.n( );  // Выводит: Method n( ) was called
// Вызов метода, определенного в классе C
cInstance.o( );  // Выводит: Method o( ) was called
// Обращение к переменной, унаследованной от класса A.
trace(cInstance.v);  // Выводит: 10
```

Более того, каждый суперкласс может иметь любое количество подклассов (однако у суперкласса нет никакой возможности узнать, какие подклассы расширяют его возможности). Следующий код добавляет в наш пример четвертый класс D. Как и B, класс D наследуется непосредственно от A. Класс D может использовать методы и переменные, определенные в нем самом и в его суперклассе A.

```
public class D extends A {
  public function p ( ) {
    trace("Method p( ) was called");
  }
}
```

Четыре класса из нашего примера образуют так называемое *дерево наследования*, или *иерархию классов*. Наглядно эта иерархия представлена на рис. 6.1. Обратите внимание, что подкласс не может иметь более одного непосредственного суперкласса.

Все приложения, разработанные с использованием объектно-ориентированного подхода, могут быть описаны с помощью диаграммы классов наподобие той, что

изображена на рис. 6.1. На самом деле многие разработчики перед тем, как приступить к написанию кода, создают диаграммы классов. Эти диаграммы могут быть неформальными, нарисованными в соответствии с собственной иконографией разработчика, или формальными, нарисованными в соответствии со спецификацией изображений диаграмм, к которой относится, например, язык UML (Unified Modeling Language) (дополнительную информацию можно получить по адресу http://www.uml.org).

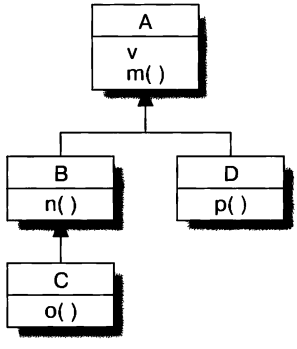

Рис. 6.1. Иерархия классов

Аналогично тому, как мы разрабатываем собственные иерархии классов для наших приложений, реализуемых с помощью объектно-ориентированного подхода, язык ActionScript тоже организует свои собственные классы в соответствии с иерархией. На самом деле любой класс в языке ActionScript (как собственный, так и пользовательский) унаследован прямо или косвенно от корневого элемента внутренней иерархии языка — класса `Object`. Класс `Object` определяет несколько базовых методов и переменных, доступных всем классам через наследование. Например, любой класс может воспользоваться методом `Object.toString( )`, возвращающим строковое представление объекта.

Статические методы и статические переменные не наследуются. В отличие от методов и переменных экземпляра, подкласс не наследует статические методы и статические переменные своего суперкласса.

Например, в следующем коде мы определяем статический метод `s( )` в классе `A`. Метод `s( )` не наследуется подклассом `B` класса `A`, и, следовательно, к этому методу нельзя обратиться в виде `B.s( )`.

```
public class A {
  public static function s ( ) {
    trace("A.s( ) was called");
  }
}

public class B extends A {
  public function B ( ) {
    B.s( ); // Ошибка! Недопустимая попытка обращения
            // к методу A.s( ) через класс B
  }
}
```

Тем не менее в теле любого из классов A или B к статическим методам и переменным, определенным в классе A, можно обращаться непосредственно, не указывая имя класса, например `s( )` вместо `A.s( )`. Но несмотря на это, при обращении к статическим методам или статическим переменным вообще разумно указывать имя класса. Когда указано имя класса, становится совершенно ясно, к какому классу относится метод или переменная.

Перекрытие методов экземпляра

В этой главе мы уже познакомились с такими методиками наследования, как *повторное использование*, когда подкласс использует методы и переменные своего суперкласса, и *расширение*, когда подкласс добавляет собственные методы и переменные. Сейчас мы рассмотрим еще одну методику — *переопределение*, при которой подкласс реализует альтернативную версию метода, определенного в его суперклассе.

Имейте в виду, что методики повторного использования, расширения и переопределения не являются взаимоисключающими. В подклассе могут применяться все три методики.

Переопределение позволяет приспособить существующий класс для решения специфической задачи путем дополнения, наложения ограничений или даже аннулирования одной или нескольких исходных возможностей. Переопределение метода на техническом языке называется *перекрытием*.

Язык ActionScript 3.0 позволяет переопределять методы экземпляра, но не допускает переопределения переменных экземпляра, статических переменных и статических методов.

Чтобы перекрыть метод экземпляра суперкласса, мы должны добавить в подкласс описание метода экземпляра с таким же именем, предварив его ключевым словом `override`. Например, рассмотрим следующий код, в котором создается класс A с методом экземпляра `m( )`:

```
public class A {
  // Объявление метода экземпляра в суперклассе
  public function m ( ) {
    trace("A's m( ) was called");
  }
}
```

Рассмотрим также следующий код, в котором создается класс B, унаследованный от класса A:

```
// Класс B является подклассом класса A
public class B extends A {
}
```

Чтобы перекрыть метод `m( )` в классе B, мы используем следующий код:

```
public class B extends A {
  // Перекрытие метода суперкласса m( )
```

```
  override public function m ( ) {
    trace("B's m( ) was called");
  }
}
```

Обратите внимание, что версия метода `m( )` класса `B` обладает не только таким же именем, как у версии метода класса `A`, но и таким же модификатором управления доступом (то есть `public`).

Для успешного перекрытия метода необходимо, чтобы у перекрывающей версии метода и у перекрываемого метода совпадали имя, модификатор управления доступом, список параметров и возвращаемый тип (возвращаемые типы будут рассмотрены в гл. 8). В противном случае произойдет ошибка.

Когда метод `m( )` вызывается через экземпляр класса `A`, среда выполнения Flash использует описание метода из класса `A`. Однако когда метод `m( )` вызывается через экземпляр класса `B`, среда Flash использует описание метода из класса `B` вместо описания из класса `A`:

```
var aInstance = new A( );
aInstance.m( );  // Выводит: A's m( ) was called

var bInstance = new B( );
bInstance.m( );  // Выводит: B's m( ) was called
```

Рассмотрим более реальный пример. Предположим, что мы разрабатываем геометрическую программу, отображающую прямоугольники и квадраты. Прямоугольники в нашей программе будет представлять класс `Rectangle`, продемонстрированный в следующем коде:

```
public class Rectangle {
  protected var w = 0;
  protected var h = 0;

  public function setSize (newW, newH) {
    w = newW;
    h = newH;
  }

  public function getArea ( ) {
    return w * h;
  }
}
```

Для представления квадратов в программе мы *могли бы* создать совершенно независимый класс `Square`. Однако квадрат на самом деле представляет собой не что иное, как прямоугольник с равными сторонами. Чтобы воспользоваться этим подобием, мы создадим класс `Square`, расширяющий класс `Rectangle`, но при этом модифицирующий метод `setSize( )` для предотвращения присваивания значений переменным `w` и `h` в тех случаях, когда значения параметров `newW` и `newH` не равны между собой. Это ограничение относится только к квадратам, а не к прямоугольникам вообще, поэтому оно не реализуется в классе `Rectangle`.

Рассмотрим код класса Square, в котором продемонстрирован перекрытый метод setSize( ):

```
public class Square extends Rectangle {
  override public function setSize (newW, newH) {
    // Это ограничение, накладываемое классом Square
    if (newW == newH) {
      w = newW;
      h = newH;
    }
  }
}
```

При вызове метода setSize( ) через экземпляр класса Square или Rectangle среда выполнения Flash использует ту версию метода, которая соответствует фактическому классу экземпляра. Например, в следующем коде мы вызываем метод setSize( ) через экземпляр класса Rectangle. Среда Flash знает, что классом экземпляра является Rectangle, поэтому вызывается версия метода setSize( ) из этого класса:

```
var r = new Rectangle( );
r.setSize(4,5);
trace(r.getArea( ));  // Выводит: 20
```

В отличие от этого, в следующем коде мы вызываем метод setSize( ) через экземпляр класса Square. На этот раз среда выполнения Flash знает, что классом экземпляра является Square, поэтому версия метода setSize( ) вызывается именно из этого класса, а не из класса Rectangle:

```
var s = new Square( );
s.setSize(4,5);
trace (s.getArea( )); // Выводит: 0 (Метод setSize( ) предотвращает
                      //             присваивание недопустимых значений)
```

В предыдущем коде результат вызова метода s.getArea( ) равен 0. Это говорит о том, что значения переменных w и h не были установлены при вызове метода s.setSize( ). В версии метода setSize( ) класса Square значения переменным w и h присваиваются только в том случае, когда значения параметров newW и newH равны между собой.

Вызов перекрытого метода экземпляра. Когда подкласс перекрывает метод экземпляра, версия этого метода, определенная в суперклассе, не теряется. Экземпляры подкласса могут обращаться к этой версии метода посредством оператора super, позволяющего вызывать перекрытый метод следующим образом:

```
super.имяМетода (аргумент1, аргумент2... аргументn);
```

В этом коде *имяМетода* обозначает имя вызываемого перекрытого метода, а *аргумент1, аргумент2... аргументn* — список аргументов, передаваемых в этот метод (другие варианты использования оператора super будут рассмотрены далее в этой главе).

В качестве примера вызова перекрытого метода вернемся к сценарию с классами Square и Rectangle. В предыдущем разделе наш метод Square.setSize( ) без необходимости дублировал код метода Rectangle.setSize( ). Рассмотрим версию метода, определенную в классе Rectangle:

```
public function setSize (newW, newH) {
  w = newW;
  h = newH;
}
```

В версии метода `setSize( )`, определенной в классе `Square`, просто добавлен оператор `if`:

```
override public function setSize (newW, newH) {
  if (newW == newH) {
    w = newW;
    h = newH;
  }
}
```

Чтобы избежать дублирования кода, используемого в обоих методах для присваивания значений переменным `w` и `h`, мы можем воспользоваться оператором `super`, как показано в следующей модифицированной версии метода `Square.setSize( )`:

```
override public function setSize (newW, newH) {
  if (newW == newH) {
    // Вызов метода setSize( ) суперкласса над текущим экземпляром
    super.setSize(newW, newH);
  }
}
```

Модифицированный метод `setSize( )` класса `Square` проверяет, одинаковы ли значения параметров `newW` и `newH`. Если значения одинаковы, то вызывается метод `setSize( )` класса `Rectangle` над текущим экземпляром. Метод `setSize( )` класса `Rectangle` позаботится о присваивании значений переменным `w` и `h`.

Приведенный пример с методом `setSize( )` демонстрирует, как подкласс может перекрывать метод для ограничения его поведения. Кроме того, подкласс может перекрывать метод для дополнения его поведения. Например, следующий код создает класс `ScreenRectangle`, являющийся подклассом класса `Rectangle` и отображающий на экране прямоугольник. Подкласс `ScreenRectangle` перекрывает метод `setSize( )`, сохраняя поведение перекрываемого метода, но при этом добавляя вызов метода `draw( )`, в результате чего размеры прямоугольника на экране изменяются всякий раз при вызове метода `setSize( )`:

```
public class ScreenRectangle extends Rectangle {
  override public function setSize (newW, newH) {
    // Вызов версии метода setSize( ) класса Rectangle
    super.setSize(newW, newH);

    // Теперь отображаем прямоугольник на экране
    draw( );
  }

  public function draw ( ) {
    // Здесь размещается код, отображающий прямоугольник на экране
  }
}
```

Перекрытие можно использовать и для аннулирования поведения метода. Методика очень проста: версия перекрытого метода подкласса не выполняет никаких действий. Например, следующий код демонстрирует подкласс `ReadOnlyRectangle`, блокирующий метод `setSize( )` класса `Rectangle`, в результате чего исключается возможность изменения размера для экземпляра этого подкласса:

```
public class ReadOnlyRectangle extends Rectangle {
  // Следующее определение фактически блокирует метод setSize( )
  // для экземпляров класса ReadOnlyRectangle.
  override public function setSize (newW, newH) {
    // Никаких действий
  }
}
```

Методы-конструкторы в подклассах

Теперь, когда мы рассмотрели поведение методов и переменных экземпляра относительно наследования, обратим наше внимание на методы-конструкторы.

Вспомним, что метод-конструктор инициализирует экземпляры класса следующими способами:

- вызывая методы, которые выполняют задачи настройки;
- присваивая значения переменным созданного объекта.

Когда происходит расширение класса, подкласс может определять собственный метод-конструктор. На конструктор подкласса возлагаются следующие функции:

- выполнять задачи настройки, относящиеся к подклассу;
- присваивать значения переменным, описанным в подклассе;
- вызывать метод-конструктор подкласса (иногда называемый *суперконструктором*).

Если в подклассе определен метод-конструктор, в нем обязательно должен вызываться конструктор суперкласса с помощью ключевого слова `super`. Более того, конструктор суперкласса должен вызываться до обращения к любой переменной или методу экземпляра. Если конструктор суперкласса не будет вызван явно, компилятор автоматически добавит вызов конструктора суперкласса без аргументов. И наконец, ключевое слово `super` не должно использоваться в методе-конструкторе более одного раза.

Запрещение использования ключевого слова `super` после того, как произошло обращение к любой переменной или методу экземпляра, имеет следующие преимущества:

- исключается вызов методов над объектом, который еще не был проинициализирован;
- устраняется доступ к переменным объекта, который еще не был проинициализирован;
- исключается возможность перезаписи значений переменных, присвоенных в конструкторе подкласса, в результате последующего вызова конструктора суперкласса.

Не путайте две разновидности оператора super. Первая разновидность — super( ) — вызывает метод-конструктор суперкласса. Вторая разновидность — super.имяМетода( ) — вызывает метод суперкласса. Использование первой разновидности допустимо только в методе-конструкторе. Вторая разновидность может многократно применяться в любом месте метода-конструктора или метода экземпляра.

Рассмотрим применение оператора `super` для вызова метода-конструктора суперкласса в простейшем случае. Следующий код описывает класс A с пустым методом-конструктором:

```
public class A {
  public function A ( ) {
  }
}
```

Следующий код описывает класс B, который расширяет класс A. Внутри метода-конструктора класса B мы используем оператор `super` для вызова метода-конструктора класса A:

```
public class B extends A {
  // Конструктор подкласса
  public function B ( ) {
    // Вызов метода-конструктора суперкласса
    super( );
  }
}
```

С точки зрения функциональности следующие описания двух методов-конструкторов являются синонимами. В первом случае метод-конструктор суперкласса вызывается явно; во втором случае среда выполнения Flash вызывает метод-конструктор суперкласса неявно.

```
public function B ( ) {
  // Явный вызов метода-конструктора суперкласса
  super( );
}

public function B ( ) {
  // Вызов конструктора отсутствует.
  // Среда Flash вызовет конструктор автоматически
}
```

Если в подклассе метод-конструктор не определен вообще, то компилятор языка ActionScript автоматически создаст метод-конструктор и добавит в него одну инструкцию — вызов оператора `super`. Следующие два описания класса B функционально являются идентичными. Первое описание в явном виде представляет то, что для второго описания автоматически создает компилятор:

```
// Определение конструктора в явном виде
public class B extends A {
  // Явное объявление конструктора
  public function B ( ) {
    // Явный вызов метода-конструктора суперкласса
```

```
    super( );
  }
}

// Позволить компилятору автоматически создать конструктор по умолчанию
public class B extends A {
}
```

Метод-конструктор подкласса может (а зачастую так и происходит) иметь другие параметры, нежели его коллега из суперкласса. Например, в нашем классе `Rectangle` можно было бы определить конструктор с параметрами `width` и `height`, а класс `Square` мог бы иметь собственный конструктор с одним параметром `side` (у квадратов ширина и высота совпадает, поэтому не нужно указывать оба значения). В листинге 6.1 продемонстрирован этот код.

Листинг 6.1. Конструкторы классов Rectangle и Square

```
public class Rectangle {
  protected var w = 0;
  protected var h = 0;

  // Конструктор класса Rectangle
  public function Rectangle (width, height) {
    setSize(width, height);
  }

  public function setSize (newW, newH) {
    w = newW;
    h = newH;
  }

  public function getArea ( ) {
    return w * h;
  }
}

public class Square extends Rectangle {
  // Конструктор класса Square
  public function Square (side) {
    // Передаем параметр side в конструктор класса Rectangle
    super(side, side);
  }

  override public function setSize (newW, newH) {
    if (newW == newH) {
      // Вызов метода setSize( ) суперкласса над текущим экземпляром
      super.setSize(newW, newH);
    }
  }
}
```

Кстати, вы могли бы задаться вопросом, а не лучше ли определить метод `setSize( )` класса `Square` с одним параметром `side`, чем иметь два отдельных параметра `width`

и height. Это демонстрирует следующая версия метода setSize( ) (обратите внимание, что в методе больше не нужна проверка значений параметров newW и newH на равенство).

```
override public function setSize (side) {
  // Вызов метода setSize( ) суперкласса над текущим экземпляром
  super.setSize(side, side);
}
```

Хотя приведенная версия метода setSize( ), несомненно, является более подходящей для класса Square, она приведет к ошибке, поскольку имеет меньшее количество параметров, чем версия метода setSize( ) класса Rectangle (помните, что количество параметров, определенных в перекрывающем методе, должно совпадать с количеством параметров перекрываемого метода). Позднее, в подразд. «Наследование в сравнении с композицией» разд. «Теория наследования», мы рассмотрим альтернативный допустимый вариант реализации версии метода setSize( ) с одним параметром в классе Square.

При определении метода-конструктора подкласса обязательно указывайте все требуемые аргументы для конструктора суперкласса. Метод-конструктор класса ColoredBall из следующего примера определен неправильно, поскольку в метод-конструктор суперкласса не передается необходимая информация:

```
public class Ball {
  private var r;
  public function Ball (radius) {
    r = radius;
  }
}

public class ColoredBall extends Ball {
  private var c;

  // Это проблематичный конструктор...
  public function ColoredBall (color) {
    // Ой! Отсутствует вызов оператора super( ). Здесь произойдет ошибка,
    // поскольку в конструктор класса Ball необходимо передать аргумент
    // для параметра radius
    c = color;
  }
}
```

Далее приводится исправленная версия класса ColoredBall, в которой требуемый аргумент передается в конструктор класса Ball:

```
public class ColoredBall extends Ball {
  private var c;

  // Все исправлено...
  public function ColoredBall (radius, color) {
    super(radius);
    c = color;
  }
}
```

Обратите внимание, что, если придерживаться хорошего тона программирования, в конструкторе подкласса сначала перечисляются параметры суперкласса (в данном случае `radius`), а затем дополнительные аргументы конструктора подкласса (в данном случае `color`).

Исключение возможности расширения классов и перекрытия методов

Чтобы исключить возможность расширения класса или перекрытия метода, перед описанием класса или метода необходимо добавить атрибут `final`. Например, следующий код описывает класс `A`, не допускающий расширения:

```
final public class A {
}
```

Поскольку класс `A` описан с использованием атрибута `final`, попытка расширить этот класс следующим образом:

```
public class B extends A {
}
```

завершится ошибкой на этапе компиляции программы:

```
Base class is final. (Базовый класс является конечным.)
```

Подобным образом следующий код описывает метод `m( )`, не допускающий перекрытия:

```
public class A {
  final public function m ( ) {
  }
}
```

Поскольку метод `m( )` описан с помощью атрибута `final`, попытка перекрыть этот метод следующим образом:

```
public class B extends A {
  override public function m ( ) {
  }
}
```

завершится ошибкой на этапе компиляции программы:

```
Cannot redefine a final method. (Невозможно переопределить конечный метод.)
```

В языке ActionScript атрибут `final` используется по нескольким причинам.

- В некоторых ситуациях методы, описанные с помощью атрибута `final`, выполняются быстрее, чем методы, описанные без него. Если вы хотите улучшить производительность вашего приложения всеми возможными способами, попробуйте описать методы, используя атрибут `final`. Однако стоит отметить, что в будущих версиях среды выполнения Flash корпорация Adobe планирует увеличить скорость выполнения методов, описанных без использования атрибута `final`, в результате чего она не будет отличаться от скорости выполнения методов, описанных с помощью атрибута `final`.

- Методы, описанные с помощью атрибута `final`, помогают скрыть детали внутренней реализации класса. Если описать класс или метод, используя этот атрибут, другие программисты не смогут расширить такой класс или перекрыть метод с целью изучения внутренней структуры класса. Такая мера предосторожности является одним из способов защиты приложения от вредоносного кода.

Если оставить в стороне проблемы эффективности и обеспечения безопасности, сообщество программистов разделилось на два лагеря в обсуждении вопроса, является ли описание методов и классов с помощью атрибута `final` хорошей практикой объектно-ориентированного программирования. С одной стороны, часть программистов утверждает, что методы и классы, описанные с помощью атрибута `final`, являются полезными, поскольку в данном случае можно быть уверенным в том, что объект будет вести себя в соответствии с четко определенным поведением, а не в соответствии с непредсказуемым (и потенциально проблематичным) перекрытым поведением. В то же время другие программисты утверждают, что методы и классы, описанные с помощью атрибута `final`, противоречат основному принципу объектно-ориентированного программирования — полиморфизму, который предполагает, что экземпляр подкласса может быть использован везде, где допустимо применение его суперкласса. С полиморфизмом мы познакомимся далее в этой главе.

Создание подклассов внутренних классов

Точно так же, как мы создаем подклассы для наших собственных классов, мы можем создавать подклассы для любого внутреннего класса, описанного без использования атрибута `final`, что позволит реализовать специализированную функциональность на базе существующего класса языка ActionScript. Пример расширения внутреннего класса `Array` можно найти в разделе Programming ActionScript 3.0 ▸ Core ActionScript 3.0 Data Types and Classes ▸ Working with Arrays ▸ Advanced Topics документации по программированию на языке ActionScript 3.0 корпорации Adobe. Пример расширения внутреннего класса `Shape` среды выполнения Flash можно найти в разд. «Пользовательские графические классы» гл. 20.

Некоторые внутренние классы языка ActionScript представляют собой простые коллекции методов и переменных класса, например классы `Math`, `Keyboard` и `Mouse` существуют только для хранения связанных методов и переменных (например, `Math.random( )` и `Keyboard.ENTER`). Такие классы называют *библиотеками статических методов*. Они объявляются в основном с использованием атрибута `final`.

Вместо того чтобы расширять эти классы, вы должны создавать свои собственные библиотеки статических методов. Например, вместо добавления метода `factorial( )` в подкласс класса `Math` следует создать собственный класс, скажем, `AdvancedMath`, который будет хранить ваш метод `factorial( )`. Класс `AdvancedMath` не может быть связан с классом `Math` через отношение наследования.

Теория наследования

До настоящего момента основное внимание уделялось рассмотрению деталей практического применения наследования в языке ActionScript. Однако теория о том, где и когда применять наследование, гораздо глубже технической реализации. Рассмотрим несколько основных теоретических принципов, принимая во внимание тот факт, что для освещения этой темы целиком нескольких страниц явно недостаточно. Более подробное описание теории наследования можно найти по адресу http://archive.eiffel.com/doc/manuals/technology/oosc/inheritance-design/page.html. Страница представляет собой онлайновую выдержку из книги «Object-Oriented Software Construction» (издательство Prentice Hall) Бертранда Мейера (Bertrand Meyer).

Почему наследование?

Наиболее очевидным преимуществом наследования является возможность повторного использования кода. Наследование позволяет отделять набор базовых возможностей от их специализированных версий. Код базовых возможностей хранится в суперклассе, а код специализаций полностью содержится в подклассе. Более того, суперкласс могут расширять несколько подклассов, благодаря чему одновременно может существовать несколько специализированных версий определенного набора возможностей. Если в суперклассе изменяется реализация некоторой возможности, все подклассы автоматически наследуют это изменение.

Кроме того, наследование позволяет выражать архитектуру приложения в иерархических терминах, отражающих реальный мир и человеческую психологию. Например, в реальном мире мы считаем, что растения отличаются от животных, но в то же время и тех и других мы относим к живым существам. Мы считаем, что автомобили отличаются от самолетов, но и те и другие являются средством передвижения. Соответствующим образом в приложении для управления кадрами может существовать суперкласс `Employee` с подклассами `Manager`, `CEO` и `Worker`. В банковском приложении можно создать суперкласс `BankAccount` с подклассами `CheckingAccount` и `SavingsAccount`. Все эти канонические примеры демонстрируют одну из разновидностей наследования, иногда называемую *наследованием подтипов*, когда иерархия классов приложения моделирует ситуацию в реальном мире (называемую *доменом* или *проблемной областью*).

Несмотря на то что примеры классов `Employee` и `BankAccount` демонстрируют привлекательные возможности наследования, далеко не каждое наследование отражает реальный мир. На самом деле чрезмерный акцент на моделировании реального мира может привести к неправильному пониманию наследования и, как следствие, к его неправильному использованию. Например, в случае с классом `Person` мы могли бы поддаться искушению и создать подклассы `Female` и `Male`. В реальном мире данные категории являются логичными, но если бы эти классы использовались, скажем, в приложении для генерации отчетов в учебном заведении, нам пришлось бы создать классы `MaleStudent` и `FemaleStudent` только для того, чтобы сохранить иерархию реального мира. В нашей программе операции,

описанные для студентов мужского пола, ничем не отличаются от операций, описанных для студентов женского пола, и, следовательно, должны использоваться одинаково. В данном случае иерархия реального мира конфликтует с иерархией нашего приложения. Если вам необходима информация о поле, лучше создать один класс `Student` и добавить переменную `gender` в класс `Person`. Насколько бы это ни было заманчивым, мы должны избегать создания структур наследования, основываясь исключительно на реальном мире, а не на требованиях нашей программы.

И наконец, помимо возможности повторного использования кода и создания логической иерархии, наследование позволяет применять различные типы объектов там, где требуется только один тип. Эта важная возможность, называемая полиморфизмом, заслуживает отдельного рассмотрения.

Полиморфизм и динамическое связывание

Полиморфизм — это возможность, присущая всем настоящим объектно-ориентированным языкам программирования, которая заключается в том, что экземпляр подкласса может быть использован везде, где допустимо применение экземпляра его суперкласса. Само по себе слово «полиморфизм» буквально обозначает «множество форм» — любой объект можно рассматривать как экземпляр собственного класса или как экземпляр любого из его суперклассов.

Напарником полиморфизма является *динамическое связывание*, гарантирующее, что в результате вызова метода над объектом будут выполнены именно те инструкции, которые определены в фактическом классе данного объекта.

В качестве канонического примера полиморфизма и динамического связывания можно привести графическое приложение, отображающее фигуры на экране. В этом приложении определен класс `Shape` с нереализованным методом `draw( )`:

```
public class Shape {
  public function draw ( ) {
    // Реализация метода отсутствует. В некоторых других языках
    // метод draw( ) был бы объявлен с помощью атрибута abstract,
    // который синтаксически обязует подклассы класса Shape
    // предоставить реализацию данного метода.
  }
}
```

Класс `Shape` имеет несколько подклассов — `Circle`, `Rectangle` и `Triangle`, каждый из которых предоставляет собственное описание метода `draw( )`:

```
public class Circle extends Shape {
  override public function draw ( ) {
    // Код для отрисовки окружности на экране не показан...
  }
}

public class Rectangle extends Shape {
  override public function draw ( ) {
```

```
    // Код для отрисовки прямоугольника на экране не показан...
  }
}

public class Triangle extends Shape {
  override public function draw ( ) {
    // Код для отрисовки треугольника на экране не показан...
  }
}
```

Чтобы нарисовать новую фигуру на экране, мы передаем экземпляр класса `Circle`, `Rectangle` или `Triangle` в метод `addShape ( )` основного класса приложения `DrawingApp`. Код метода `addShape ( )` класса `DrawingApp` выглядит следующим образом:

```
public function addShape (newShape) {
  newShape.draw( );

  // Оставшаяся часть метода (код не показан) занималась бы добавлением
  // новой фигуры во внутренний список фигур, отображаемых
  // на экране
}
```

Теперь рассмотрим пример добавления фигуры, представленной классом `Circle`, на экран:

```
drawingApp.addShape(new Circle( ));
```

Метод `addShape ( )` вызывает метод `draw ( )` для новой фигуры и добавляет эту фигуру во внутренний список фигур, отображаемых на экране. И самое главное — метод `addShape ( )` вызывает метод `draw ( )`, не зная (и не заботясь о том), экземпляром какого класса является новая фигура — `Circle`, `Rectangle` или `Triangle`. Благодаря процессу *динамического связывания*, происходящему на этапе выполнения программы, среда Flash использует подходящую реализацию данного метода. Иными словами, если новая фигура является экземпляром класса `Circle`, то среда выполнения Flash вызовет метод `Circle.draw ( )`; если новая фигура является экземпляром класса `Rectangle`, то Flash вызовет метод `Rectangle.draw ( )`; если же новая фигура является экземпляром класса `Triangle`, то среда Flash вызовет метод `Triangle.draw ( )`. Важно отметить, что на этапе компиляции конкретный класс добавляемой фигуры неизвестен. По этой причине динамическое связывание часто называют *поздним связыванием*: вызов метода *связывается* с конкретной реализацией «с опозданием» (то есть на этапе выполнения).

Ключевым преимуществом динамического связывания и полиморфизма является возможность локализации изменений кода. Полиморфизм позволяет одной части приложения оставаться неизменной даже при изменении другой части. Например, рассмотрим, каким образом мы могли бы нарисовать фигуры, если бы полиморфизм не существовал. Во-первых, нам бы пришлось использовать уникальные имена для каждой версии метода `draw ( )`:

```
public class Circle extends Shape {
  public function drawCircle ( ) {
```

```
    // Код для отрисовки окружности на экране не показан...
  }
}

public class Rectangle extends Shape {
  public function drawRectangle ( ) {
    // Код для отрисовки прямоугольника на экране не показан...
  }
}

public class Triangle extends Shape {
  public function drawTriangle ( ) {
    // Код для отрисовки треугольника на экране не показан...
  }
}
```

Далее внутри метода `addShape( )` класса `DrawingApp` нам бы пришлось использовать оператор `is`, чтобы вручную определять класс каждой новой фигуры и вызывать подходящий метод для отрисовки, как показано в следующем коде. Оператор `is` возвращает значение `true` в том случае, если указанное выражение принадлежит заданному типу данных; в противном случае возвращается значение `false`. Типы данных и оператор `is` будут рассмотрены в гл. 8.

```
public function addShape (newShape) {
  if (newShape is Circle) {
    newShape.drawCircle( );
  } else if (newShape is Rectangle) {
    newShape.drawRectangle( );
  } else if (newShape is Triangle) {
    newShape.drawTriangle( );
  }

  // Оставшаяся часть метода (код не показан) занималась бы добавлением
  // новой фигуры во внутренний список фигур, отображаемых на экране
}
```

Уже сейчас очевидны трудности выбранного подхода. Теперь представьте, что произойдет, если мы добавим 20 новых типов фигур. Для каждого нового типа нам придется вносить изменения в метод `addShape( )`. В мире, где существует полиморфизм, нам не пришлось бы изменять код, вызывающий метод `draw( )` над каждым экземпляром класса `Shape`. Поскольку каждый подкласс класса `Shape` предоставляет собственное подходящее описание метода `draw( )`, наше приложение будет «просто работать» без необходимости внесения других изменений.

Полиморфизм не только упрощает взаимодействие между программистами, но и позволяет применять и расширять библиотеки, не требуя при этом доступа к их исходному коду. Некоторые разработчики утверждают, что полиморфизм — это величайший вклад объектно-ориентированного программирования в компьютерную науку.

Наследование в сравнении с композицией

Композиция, представляющая альтернативный вид отношений между объектами, зачастую соперничает с наследованием в качестве методики объектно-ориентиро-

ванного дизайна. В композиции один класс *(внешний)* хранит экземпляр другого класса *(внутреннего)* в переменной экземпляра. Внешний класс поручает работу внутреннему классу, вызывая методы над этим экземпляром. Вот базовый подход, представленный обобщенным кодом:

```
// Внутренний класс является аналогом суперкласса в наследовании
public class BackEnd {
  public function doSomething ( ) {
  }
}

// Внешний класс является аналогом подкласса в наследовании
public class FrontEnd {
  // Экземпляр внутреннего класса сохраняется в закрытой переменной
  // экземпляра, в данном случае в закрытой пременной be
  private var be;

  // Конструктор создает экземпляр внутреннего класса
  public function FrontEnd ( ) {
    be = new BackEnd( );
  }

  // Этот метод поручает работу методу doSomething( ) класса BackEnd
  public function doSomething ( ) {
    be.doSomething( );
  }
}
```

Обратите внимание, что класс `FrontEnd` не расширяет класс `BackEnd`. Композиция не требует использования собственного особого синтаксиса, как это происходит с наследованием. Более того, внешний класс может использовать подмножество методов внутреннего класса, все методы или добавлять собственные несвязанные методы. Имена методов во внешнем классе могут полностью совпадать с именами методов во внутреннем классе или совершенно отличаться. Внешний класс может ограничивать, расширять или переопределять возможности внутреннего класса, играя такую же роль, как подкласс в наследовании.

Ранее в этой главе было описано, как с помощью наследования класс `Square` может ограничивать поведение класса `Rectangle`. В листинге 6.2 показано, как одно и то же отношение между классами может быть реализовано с использованием композиции вместо наследования. В предлагаемом коде класс `Rectangle` остался неизменным. Однако на этот раз класс `Square` не расширяет `Rectangle`. Вместо этого в классе `Square` описана переменная `r`, содержащая экземпляр класса `Rectangle`. Фильтрация всех операций над экземпляром, хранящимся в переменной `r`, происходит через public-методы класса `Square`. Класс `Square` переадресует, или *делегирует*, вызовы методов экземпляру, хранящемуся в переменной `r`. Обратите внимание, что, поскольку метод `setSize( )` класса `Square` не перекрывает метод `setSize( )` класса `Rectangle`, сигнатура метода `setSize( )` класса `Square` не обязана совпадать с сигнатурой метода `setSize( )` класса `Rectangle`. В методе `setSize( )` класса `Square` можно определить один-единственный параметр, в отличие от метода `setSize( )` класса `Rectangle`, в котором определено два параметра.

Листинг 6.2. Пример отношения композиции

```
// Класс Rectangle
public class Rectangle {
  protected var w = 0;
  protected var h = 0;

  public function Rectangle (width, height) {
    setSize(width, height);
  }

  public function setSize (newW, newH) {
    w = newW;
    h = newH;
  }

  public function getArea ( ) {
    return w * h;
  }
}

// Это новый класс Square
public class Square {
  private var r;

  public function Square (side) {
    r = new Rectangle(side, side);
  }

  public function setSize (side) {
    r.setSize(side, side);
  }

  public function getArea ( ) {
    return r.getArea( );
  }
}
```

Отношения «является», «имеет» и «использует». В разговорной речи отношение наследования, присущее объектно-ориентированным языкам программирования, называется отношением «является» (Is-A), поскольку экземпляр подкласса в буквальном смысле можно рассматривать как экземпляр его суперкласса (то есть экземпляр подкласса может быть использован везде, где это допустимо применением экземпляра его суперкласса). В предыдущем примере полиморфизма экземпляр класса `Circle` «является» экземпляром класса `Shape`, поскольку класс `Circle` унаследован от `Shape` и, следовательно, может использоваться везде, где используется `Shape`.

Отношение композиции называется отношением «имеет», поскольку внешний класс содержит экземпляр внутреннего класса. Не следует путать отношение «имеет» с отношением «использует», когда некоторый класс создает объект другого класса, но не присваивает созданный объект переменной экземпляра. В отношении «использует» класс использует объект, а затем выбрасывает его. Например, класс `Circle` мо-

жет хранить числовое значение цвета в переменной `color` («имеет» объект класса `uint`), но впоследствии может временно воспользоваться объектом класса `Color`, чтобы отобразить этот цвет на экране («использует» объект класса `Color`).

В листинге 6.2 класс `Square` «имеет» экземпляр класса `Rectangle` и налагает на него ограничения, которые фактически превращают класс `Rectangle` в `Square`. В случае с классами `Square` и `Rectangle` отношение «является» выглядит более естественным, однако можно использовать и отношение «имеет». В этой связи возникает вопрос: какое отношение лучше?

Когда использовать композицию вместо наследования. Листинг 6.2 поднимает серьезный вопрос проектирования. Как сделать выбор между композицией и наследованием? Вообще, достаточно легко определить ситуацию, в которой наследование неприменимо. Экземпляр класса `AlertDialog` в приложении «имеет» кнопку OK, но сам экземпляр класса `AlertDialog` кнопкой OK не «является». Сложнее определить ситуацию, когда неприменимой оказывается композиция, поскольку наследование, используемое для создания отношения между двумя классами, всегда можно заменить композицией. Если в одной и той же ситуации применимы оба подхода, какой из них окажется лучшим выбором?

Для новичков в объектно-ориентированном программировании будет неожиданностью услышать, что зачастую при выборе стратегии проектирования приложений предпочтение отдают композиции, а не наследованию. На самом деле некоторые известные теоретики в области объектно-ориентированного проектирования недвусмысленно советуют использовать композицию вместо наследования (книга «Design Patterns: Elements of Reusable Object-Oriented Software» издательства Addison-Wesley, авторы Эрих Гамма (Erich Gamma) и др.). Таким образом, здравый смысл подсказывает нам хотя бы рассмотреть возможность применения композиции даже в том случае, когда выбор наследования кажется очевидным. И все-таки, вот несколько общих рекомендаций, которые помогут сделать выбор между наследованием и композицией:

- если вы желаете воспользоваться преимуществом полиморфизма, рассмотрите возможность применения наследования;
- когда классу необходимы сервисы другого класса, рассмотрите возможность использования отношения композиции;
- если поведение разрабатываемого вами класса очень похоже на поведение существующего класса, рассмотрите возможность применения отношения наследования.

Дополнительные советы по выбору подхода проектирования между композицией и наследованием можно найти в прекрасной статье JavaWorld Билла Веннера (Bill Venner), которая хранится в архиве на сайте автора: http://www.artima.com/designtechniques/compoinh.html. Мистер Веннер приводит неоспоримые доказательства, что:

- изменение кода, использующего композицию, влечет за собой меньше последствий, чем изменение кода, использующего наследование;
- код, основанный на наследовании, зачастую выполняется быстрее, чем код, в основе которого лежит композиция.

Отсутствие поддержки абстрактных классов и методов

Во многих объектно-ориентированных проектах программ требуется использовать так называемые *абстрактные классы*. Абстрактным считается любой класс, в котором определены один или несколько *абстрактных методов*. Это методы, которые имеют имя, параметры и возвращаемый тип, но не имеют реализации (то есть не имеют тела метода). Класс, желающий расширить абстрактный класс, должен либо реализовать все абстрактные методы суперкласса, либо сам являться абстрактным классом; в противном случае на этапе компиляции произойдет ошибка. Подклассы абстрактного класса фактически обещают предоставить некий существующий код, который выполняет задачу, описанную абстрактным классом только в теории.

Абстрактные классы являются широко распространенной, важной частью проектов, в которых применяется полиморфизм. Например, ранее при обсуждении полиморфизма мы рассмотрели класс `Shape` и его подклассы `Circle`, `Rectangle` и `Triangle`. В обычной ситуации метод `draw( )` класса `Shape` был бы объявлен абстрактным методом, гарантируя, что:

- каждый подкласс класса `Shape` предоставляет средства для его отображения на экране;
- внешний код может безопасно вызывать метод `draw( )` над любым подклассом класса `Shape` (поскольку компилятор не позволит классу расширить класс `Shape`, не реализовав метод `draw( )`).

К сожалению, язык ActionScript не поддерживает абстрактные классы и абстрактные методы. Объявление абстрактного метода в языке ActionScript заменяется простым описанием метода, который не содержит кода в своем теле, и указанием в документации, что этот метод является абстрактным. Позаботиться о том, чтобы все подклассы предполагаемого абстрактного класса реализовали соответствующие методы, должен программист (а не компилятор).

В большинстве случаев для реализации конкретной объектно-ориентированной архитектуры вместо абстрактных классов могут быть использованы интерфейсы языка ActionScript. О том, что такое интерфейсы, вы прочтете в гл. 9.

Мы рассмотрели понятие наследования. В конце этой главы применим полученные знания к программе по созданию виртуального зоопарка.

Применение наследования в программе по созданию виртуального зоопарка

В нашей программе «Зоопарк» наследование будет добавлено для решения двух различных задач. Во-первых, мы используем его для описания видов пищи, поглощаемой нашими животными, тем самым, заменив предыдущий подход, заключав-

шийся в добавлении определенного количества калорий животному с помощью метода `eat( )`. Во-вторых, мы используем наследование, чтобы основной класс нашего приложения `VirtualZoo` мог отображаться на экране.

Создание видов пищи

До этого момента наша реализация процесса принятия пищи в программе по созданию виртуального зоопарка была чрезмерно упрощена. Чтобы накормить животное, мы просто вызывали метод `eat( )` над желаемым объектом класса `VirtualPet` и указывали количество калорий, поглощаемое животным. Для реализма нашей имитации добавим в программу зоопарка *виды пищи*.

Чтобы не усложнять процесс, мы позволим животному принимать только два вида пищи: суши и яблоки. Суши будут представлены новым классом `Sushi`, а яблоки — классом `Apple`. Поскольку оба класса — `Sushi` и `Apple` — концептуально представляют пищу, они будут иметь почти одинаковую функциональность. Следовательно, в нашем приложении мы реализуем всю функциональность, необходимую обоим классам `Sushi` и `Apple`, в одном суперклассе `Food`. Классы `Sushi` и `Apple` расширят класс `Food` и через наследование получат доступ к его возможностям.

Класс `Food` описывает четыре простых метода для получения и изменения имени и значения калорий для заданного продукта. Рассмотрим его код:

```
package zoo {
  public class Food {
    private var calories;
    private var name;

    public function Food (initialCalories) {
      setCalories(initialCalories);
    }

    public function getCalories ( ) {
      return calories;
    }

    public function setCalories (newCalories) {
      calories = newCalories;
    }

    public function getName ( ) {
      return name;
    }

    public function setName (newName) {
      name = newName;
    }
  }
}
```

Класс Apple задаст количество калорий, используемое по умолчанию, для каждого объекта Apple и определяет название продукта для всех объектов Apple. Рассмотрим его код:

```
package zoo {
  public class Apple extends Food {
    // Количество калорий, используемое по умолчанию, для объекта
    // Apple равно 100
    private static var DEFAULT_CALORIES = 100;

    public function Apple (initialCalories = 0) {
      // Если для данного конкретного объекта количество калорий не указано
      // или если было указано отрицательное число...
      if (initialCalories <= 0) {
        // ...использовать значение по умолчанию
        initialCalories = Apple.DEFAULT_CALORIES;
      }
      super(initialCalories);

      // Определить название продукта для всех объектов Apple
      setName("Apple");
    }
  }
}
```

Класс Sushi задает количество калорий, используемое по умолчанию, для каждого объекта Sushi и определяет название продукта для всех объектов Sushi. Рассмотрим его код:

```
package zoo {
  public class Sushi extends Food {
    private static var DEFAULT_CALORIES = 500;

    public function Sushi (initialCalories = 0) {
      if (initialCalories <= 0) {
        initialCalories = Sushi.DEFAULT_CALORIES;
      }
      super(initialCalories);

      setName("Sushi");
    }
  }
}
```

Чтобы объекты класса VirtualPet могли есть яблоки и суши, мы должны модифицировать метод eat( ) класса VirtualPet. Вот как выглядел метод eat( ) до настоящего времени:

```
public function eat (numberOfCalories) {
  if (currentCalories == 0) {
    trace(getName( ) + " is dead. You can't feed it.");
    return;
  }
```

```
  var newCurrentCalories = currentCalories + numberOfCalories;
  if (newCurrentCalories > VirtualPet.maxCalories) {
    currentCalories = VirtualPet.maxCalories;
  } else {
    currentCalories = newCurrentCalories;
  }
  trace(getName( ) + " ate some food. It now has " + currentCalories
        + " calories remaining.");
}
```

В новой версии метода `eat( )` мы переименуем параметр `numberOfCalories` в `foodItem`, введя тем самым логическое соглашение, что аргументом метода `eat( )` должен быть экземпляр любого класса, унаследованного от класса `Food` (в гл. 8 будет рассмотрено, как обеспечить выполнение этого соглашения с помощью *объявления типа*). Внутри метода `eat( )` значение переменной `newCurrentCalories` будет вычисляться путем сложения значения калорий принимаемого куска пищи (то есть `foodItem.getCalories( )`) и существующего количества калорий у животного (то есть `currentCalories`). Наконец, при выводе информации о том, что животное съело пищу, мы воспользуемся методом `getName( )` класса `Food`, чтобы указать название съеденной пищи. Рассмотрим измененный метод `eat( )`:

```
public function eat (foodItem) {
  if (currentCalories == 0) {
    trace(getName( ) + " is dead. You can't feed it.");
    return;
  }

  var newCurrentCalories = currentCalories + foodItem.getCalories( );
  if (newCurrentCalories > VirtualPet.maxCalories) {
    currentCalories = VirtualPet.maxCalories;
  } else {
    currentCalories = newCurrentCalories;
  }
  trace(getName( ) + " ate some " + foodItem.getName( ) + "."
        + " It now has " + currentCalories + " calories remaining.");
}
```

Теперь попробуем покормить животное в конструкторе класса `VirtualZoo`. Код будет выглядеть следующим образом:

```
package zoo {
  public class VirtualZoo {
    private var pet;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      pet.eat(new Apple( )); // Дать Стэну яблоко
      pet.eat(new Sushi( )); // Дать Стэну суши
    }
  }
```

В настоящий момент классы Sushi и Apple очень просты, однако они формируют основу для более сложного поведения. Например, сейчас, когда в нашей программе появились виды пищи, мы можем достаточно легко создать животных, которым нравятся только яблоки или которые едят суши только после шести часов вечера. Кроме того, мы можем легко изменить поведение каждого вида пищи. В качестве примера половину всех яблок в случайном порядке сделаем червивыми, при этом животные не должны есть такие яблоки.

Для хранения информации о том, является объект Apple «червивым» или нет, мы добавим новую переменную экземпляра wormInApple в класс Apple:

```
private var wormInApple:
```

Чтобы выбрать случайное число в диапазоне от 0 до 0,9999..., внутри конструктора класса Apple воспользуемся функцией Math.random( ). Если число окажется равным или больше 0,5, присвоим переменной wormInApple значение true, указывающее на то, что данный объект Apple является «червивым». В противном случае переменной wormInApple будет присвоено значение false, указывающее на то, что данный объект Apple не является «червивым». Вот этот код:

```
wormInApple = Math.random( ) >= .5;
```

Чтобы другие классы имели возможность определить, является ли объект Apple «червивым», опишем новый открытый метод hasWorm( ), который просто возвращает значение переменной wormInApple. Рассмотрим его код:

```
public function hasWorm ( ) {
  return wormInApple;
}
```

Наконец, чтобы животные не ели червивые яблоки, изменим метод eat( ) класса VirtualPet. Вот фрагмент кода из метода eat( ):

```
if (foodItem is Apple) {
  if (foodItem.hasWorm( )) {
    trace("The " + foodItem.getName( ) + " had a worm. " + getName( )
          + " didn't eat it.");
    return;
  }
}
```

В предыдущем коде обратите внимание на использование оператора is, который проверяет, является ли объект экземпляром указанного класса или любого класса, унаследованного от него. Выражение foodItem is Apple возвращает значение true в том случае, когда параметр foodItem ссылается на экземпляр класса Apple (или любого класса, унаследованного от класса Apple). В противном случае возвращается false. Если параметр foodItem представляет объект класса Apple, а метод hasWorm( ) этого объекта возвращает значение true, метод eat( ) завершается, при этом значение переменной currentCalories не изменяется.

В программе по созданию виртуального зоопарка помимо создания видов пищи у наследования есть еще одно применение. Познакомимся с ним.

Подготовка класса VirtualZoo для отображения на экране

Поскольку язык ActionScript применяется для создания графического содержимого и пользовательских интерфейсов, основной класс любой программы на языке ActionScript должен расширять либо класс `flash.display.Sprite`, либо класс `flash.display.MovieClip`. И `Sprite` и `MovieClip` представляют собой контейнеры для графического содержимого, отображаемого на экране.

Класс `MovieClip` используется в тех случаях, когда основной класс программы связан с FLA-файлом (документ среды разработки Flash) (более подробную информацию можно найти в гл. 29). В остальных случаях используется класс `Sprite`.

При открытии нового SWF-файла среда выполнения Flash создает экземпляр основного класса этого файла и добавляет созданный экземпляр в иерархический список объектов, которые отображаются в данный момент на экране. Этот список называется *списком отображения*. Попав в список отображения, экземпляр класса может использовать унаследованные методы класса `DisplayObject` (потомками которого являются классы `Sprite` и `MovieClip`) для добавления другого графического содержимого на экран.

Наша программа по созданию виртуального зоопарка в конечном итоге будет отображать графическое содержимое на экране. Однако перед этим придется многое узнать о списке отображения и программировании графики. Все эти темы подробно рассматриваются в части II книги.

Пока же, чтобы запустить нашу программу в ее текущем состоянии, мы должны выполнить требование, заключающееся в том, что основной класс любой программы на языке ActionScript должен расширять либо класс `Sprite`, либо класс `MovieClip`.

Наша программа не содержит никаких элементов среды разработки Flash, поэтому основной класс программы `VirtualZoo` расширяет класс `Sprite`:

```
package zoo {
  import flash.display.Sprite;

  public class VirtualZoo extends Sprite {
    private var pet;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      pet.eat(new Apple( ));
      pet.eat(new Sushi( ));
    }
  }
}
```

В этой главе мы внесли много изменений в нашу программу, создающую виртуальный зоопарк. Рассмотрим весь код целиком.

Код программы Virtual Zoo

Листинг 6.3 демонстрирует код класса `VirtualZoo` — основного класса программы.

Листинг 6.3. Класс VirtualZoo

```
package zoo {
  import flash.display.Sprite;

  public class VirtualZoo extends Sprite {
    private var pet;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      pet.eat(new Apple( ));
      pet.eat(new Sushi( ));
    }
  }
}
```

Листинг 6.4 демонстрирует код класса `VirtualPet`, экземпляры которого представляют животных в зоопарке.

Листинг 6.4. Класс VirtualPet

```
package zoo {
  import flash.utils.setInterval;
  import flash.utils.clearInterval;

  internal class VirtualPet {
    private static var maxNameLength = 20;
    private static var maxCalories = 2000;
    private static var caloriesPerSecond = 100;

    private var petName;
    private var currentCalories = VirtualPet.maxCalories/2;
    private var digestIntervalID;

    public function VirtualPet (name) {
      setName(name);
      digestIntervalID = setInterval(digest, 1000);
    }

    public function eat (foodItem) {
      if (currentCalories == 0) {
        trace(getName( ) + " is dead. You can't feed it.");
        return;
      }

      if (foodItem is Apple) {
        if (foodItem.hasWorm( )) {
          trace("The " + foodItem.getName( ) + " had a worm.." + getName( )
```

```
            + " didn't eat it.");
        return;
      }
    }

    var newCurrentCalories = currentCalories + foodItem.getCalories( );
    if (newCurrentCalories > VirtualPet.maxCalories) {
      currentCalories = VirtualPet.maxCalories;
    } else {
      currentCalories = newCurrentCalories;
    }
    trace(getName( ) + " ate some " + foodItem.getName( ) + "."
          + " It now has " + currentCalories + " calories remaining.");
  }

  public function getHunger ( ) {
    return currentCalories / VirtualPet.maxCalories;
  }

  public function setName (newName) {
    // Если длина заданного нового имени больше maxNameLength символов...
    if (newName.length > VirtualPet.maxNameLength) {
      // ...обрезать имя
      newName = newName.substr(0, VirtualPet.maxNameLength);
    } else if (newName == "") {
      // ...в противном случае, если заданное новое имя является
      // пустой строкой, завершить выполнение метода, не изменяя
      // значения переменной petName
      return;
    }

    // Присвоить новое проверенное имя переменной petName
    petName = newName;
  }

  public function getName ( ) {
    return petName;
  }

  private function digest ( ) {
    // Если в результате потребления очередной порции калорий
    // значение переменной currentCalories животного
    // станет равным 0 или меньше...
    if (currentCalories - VirtualPet.caloriesPerSecond <= 0) {
      // ...прекратить вызов метода digest( )
      clearInterval(digestIntervalID);
      // После чего очистить желудок животного
      currentCalories = 0;
      // и сообщить о смерти животного
      trace(getName( ) + " has died.");
    } else {
```

```
        // ...иначе употребить оговоренное количество калорий
        currentCalories -= VirtualPet.caloriesPerSecond;

        // и сообщить о новом состоянии животного
        trace(getName( ) + " digested some food. It now has "
              + currentCalories + " calories remaining.");
      }
    }
  }
}
```

Листинг 6.5 демонстрирует код класса `Food`, являющегося суперклассом для различных видов пищи, принимаемой животными.

Листинг 6.5. Класс Food

```
package zoo {
  public class Food {
    private var calories;
    private var name;

    public function Food (initialCalories) {
      setCalories(initialCalories);
    }

    public function getCalories ( ) {
      return calories;
    }

    public function setCalories (newCalories) {
      calories = newCalories;
    }

    public function getName ( ) {
      return name;
    }

    public function setName (newName) {
      name = newName;
    }
  }
}
```

Листинг 6.6 демонстрирует код класса `Apple`, представляющего конкретный вид пищи, принимаемой животными.

Листинг 6.6. Класс Apple

```
package zoo {
  public class Apple extends Food {
    private static var DEFAULT_CALORIES = 100;
    private var wormInApple;

    public function Apple (initialCalories = 0) {
      if (initialCalories <= 0) {
```

```
        initialCalories = Apple.DEFAULT_CALORIES;
      }
      super(initialCalories);

      wormInApple = Math.random( ) >= .5;

      setName("Apple");
    }

    public function hasWorm ( ) {
      return wormInApple;
    }
  }
}
```

Наконец, листинг 6.7 демонстрирует код класса `Sushi`, также представляющего конкретный вид пищи, принимаемой животными.

Листинг 6.7. Класс Sushi

```
package zoo {
  public class Sushi extends Food {
    private static var DEFAULT_CALORIES = 500;

    public function Sushi (initialCalories = 0) {
      if (initialCalories <= 0) {
        initialCalories = Sushi.DEFAULT_CALORIES;
      }
      super(initialCalories);

      setName("Sushi");
    }
  }
}
```

Первый запуск!

Благодаря изменениям, внесенным в программу по созданию виртуального зоопарка в этой главе, наше приложение теперь готово к компиляции и выполнению. Надеюсь, что вы испытываете нескрываемое волнение в преддверии изучения следующей главы. Из нее вы узнаете, как запустить программу после того, как она будет скомпилирована с помощью среды разработки Flash, приложения Flex Builder или компилятора mxmlc.

Компиляция и выполнение программы

После напряженной работы над программой по созданию виртуального зоопарка мы готовы скомпилировать и выполнить наш код. В этой главе врассказывается, как скомпилировать программу с помощью среды разработки Flash, приложения Flex Builder и компилятора mxmlc. В каждом случае мы исходим из того, что компилируемая программа находится в папке /virtualzoo, а исходный код программы (то есть AS-файлы) размещается в папке /virtualzoo/src.

Приступим же к компиляции!

Компиляция с помощью среды разработки Flash

Для компиляции программы «Зоопарк» с помощью среды разработки Flash мы должны сначала связать основной класс программы с FLA-файлом, используя описанную последовательность действий.

1. В среде разработки Flash выберите команду меню File ▸ New (Файл ▸ Создать).
2. В появившемся окне New Document (Новый документ) выберите в списке пункт Flash File (ActionScript 3.0) и нажмите кнопку OK.
3. Выберите команду меню File ▸ Save As (Файл ▸ Сохранить как).
4. В открывшемся окне Save as (Сохранить как) найдите папку /virtualzoo/src.
5. В поле File name (Имя файла) диалогового окна Save as (Сохранить как) введите `VirtualZoo.fla` и нажмите кнопку OK.
6. На панели Properties (Свойства) в поле Document class (Класс документа) введите `zoo.VirtualZoo`.

Теперь, когда основной класс программы связан с FLA-файлом (как описано в предыдущих шагах), можно выбрать команду меню Control ▸ Test Movie (Управление ▸ Проверка фильма), чтобы скомпилировать программу и запустить полученный SWF-файл в отладочной версии приложения Flash Player непосредственно в среде разработки Flash. Когда программа выполняется в режиме Test Movie (Проверка фильма), сообщения, генерируемые функцией `trace( )`, отображаются в окне Output (Вывод) среды разработки Flash.

Когда программа компилируется с помощью команды Test Movie (Проверка фильма), среда разработки Flash генерирует SWF-файл, имя которого совпадает с именем соответствующего FLA-файла. Например, если мы компилируем программу «Зоопарк» с помощью команды Test Movie (Проверка фильма), в папке /src по-

явится новый файл `VirtualZoo.swf`. Найти информацию о том, как изменить папку, в которую помещается сгенерированный SWF-файл, можно в разделе документации по среде разработки Flash, посвященном команде File ▸ Publish Settings (Файл ▸ Настройки публикации).

Если бы нам потребовалось опубликовать файл `VirtualZoo.swf` в Интернете, мы бы добавили его в HTML-страницу. Дополнительную информацию можно получить в разделе документации по среде Flash, посвященном команде File ▸ Publish (Файл ▸ Опубликовать). Для распространения файла `VirtualZoo.swf` в качестве настольного приложения нам потребуется включить его в инсталлируемый AIR-файл. Подробную информацию можно найти в документации по среде выполнения Adobe AIR.

Компиляция с помощью приложения Flex Builder

Перед тем как скомпилировать программу по созданию виртуального зоопарка, используя приложение Flex Builder, мы должны внести некоторые изменения в наш код, чтобы выполнить требования, предъявляемые компилятором этого приложения. Компилятор приложения Flex Builder требует, чтобы основной класс программы принадлежал безымянному пакету. На данный момент пакет, которому принадлежит наш класс `VirtualZoo`, называется `zoo`.

Перемещение основного класса в безымянный пакет

Чтобы переместить класс `VirtualZoo` из `zoo` в безымянный пакет, мы должны выполнить такую последовательность действий.

1. Сначала переместить файл `VirtualZoo.as` из папки /virtualzoo/src/zoo в папку /virtualzoo/src.
2. В файле `VirtualZoo.as` добавить следующий код прямо перед описанием класса `VirtualZoo` (этот код импортирует классы из пакета `zoo`):

   ```
   import zoo.*;
   ```

3. В файле `VirtualZoo.as` удалить имя пакета `"zoo"` из объявления пакета, то есть изменить этот код:

   ```
   package zoo {
   ```

   на следующий:

   ```
   package {
   ```

4. В файле `VirtualPet.as` изменить модификатор управления доступом для класса `VirtualPet` с `internal` на `public`, как показано ниже (это позволит классу `VirtualZoo` обращаться к классу `VirtualPet`):

   ```
   public class VirtualPet {
   ```

После того как все описанные изменения будут внесены, мы сможем откомпилировать программу.

Компиляция программы

Чтобы откомпилировать программу по созданию виртуального зоопарка, мы сначала создадим новый проект ActionScript, как описано далее.

1. Выберите команду меню File ▸ New ▸ ActionScript Project (Файл ▸ Создать ▸ Проект ActionScript).
2. В появившемся окне New ActionScript Project (Новый проект ActionScript) в поле Project name (Название проекта) введите `virtualzoo`.
3. В области Project contents (Содержание проекта) снимите флажок Use default location (Использовать местоположение по умолчанию).
4. В поле Folder (Папка) области Project contents (Содержание проекта) укажите (или найдите с помощью кнопки Browse (Обзор)) местоположение папки virtualzoo на жестком диске вашего компьютера.
5. Нажмите кнопку Next (Далее).
6. В поле Main source folder (Основная папка с исходными файлами) введите `src`.
7. В поле Main application file (Основной файл приложения) введите название `VirtualZoo.as`.
8. Нажмите кнопку Finish (Готово).

Выполнение программы

Теперь, когда проект ActionScript создан, мы можем выполнить следующие шаги, чтобы запустить программу, имитирующую виртуальный зоопарк, в режиме отладки, при котором все сообщения, генерируемые функцией `trace( )`, будут отображаться в окне Console (Консоль).

1. В окне Navigator (Навигация) выделите любой класс, принадлежащий проекту виртуального зоопарка.
2. Выберите команду меню Run ▸ Debug VirtualZoo (Выполнить ▸ Отладка Virtual Zoo). Если стандартные настройки не изменялись, программа запустится в браузере, используемом по умолчанию.

При компиляции приложение Flex Builder генерирует следующие файлы, которые размещаются в автоматически создаваемой папке /virtualzoo/bin.

- SWF-файл `VirtualZoo.swf`.
- SWF-файл `VirtualZoo-debug.swf`, используемый для отладки.
- HTML-файл `VirtualZoo.html`, который внедряет файл `VirtualZoo.swf` для публикации в Интернете.
- HTML-файл с именем `VirtualZoo-debug.html`, который внедряет файл `VirtualZoo-debug.swf` для отладки программы в браузере.
- Несколько вспомогательных файлов, позволяющих из браузера определить наличие приложения Flash Player на компьютере пользователя и при необходимости установить его автоматически.

Чтобы опубликовать файл `VirtualZoo.swf` в Интернете, просто поместите все файлы из папки /bin — кроме `VirtualZoo-debug.html` и `VirtualZoo-debug.swf` —

в папку на общедоступном веб-сервере. Для распространения файла `VirtualZoo.swf` в качестве настольного приложения обратитесь к документации по среде выполнения Adobe AIR.

Компиляция с помощью компилятора mxmlc

Как и компилятор приложения Flex Builder, mxmlc требует, чтобы основной класс программы принадлежал безымянному пакету. Таким образом, перед компиляцией программы по созданию виртуального зоопарка с помощью mxmlc мы должны переместить класс `VirtualZoo` из пакета `zoo` в безымянный пакет, выполнив шаги из подразд. «Перемещение основного класса в безымянный пакет» предыдущего раздела.

После этого мы должны найти сам компилятор, имеющий имя `mxmlc.exe`. Местоположение компилятора зависит от его версии и операционной системы. Обычно он находится в папке **Flex SDK [версия]\bin**, однако вы должны самостоятельно определить его местоположение на своем компьютере, руководствуясь документацией по инструментарию Flex SDK. Для данного примера предположим, что мы компилируем программу в операционной системе Windows XP и что компилятор находится по адресу `C:\Flex SDK 2\bin\mxmlc.exe`.

Предположим также, что папка нашей программы **/virtualzoo** находится по адресу `C:\data\virtualzoo\`.

При компиляции программы «Зоопарк» с помощью компилятора mxmlc выполняется такая последовательность действий.

1. Используя меню **Пуск** операционной системы Windows, откройте окно командной строки, выбрав команду **Пуск ▸ Все программы ▸ Стандартные ▸ Командная строка**.
2. В окне командной строки перейдите к папке `C:\Flex SDK 2\bin\`, введя следующую команду:

   ```
   cd C:\Flex SDK 2\bin
   ```

3. Далее в окне командной строки введите следующую команду и нажмите клавишу **Enter**:

   ```
   mxmlc C:\data\virtualzoo\src\VirtualZoo.as
   ```

В результате выполнения описанных шагов компилятор mxmlc откомпилирует программу и сгенерирует SWF-файл с именем `VirtualZoo.swf`, который будет помещен в папку **virtualzoo\src**. Стоит отметить, что компилятор mxmlc имеет множество параметров компиляции; подробную информацию можно получить в документации по инструментарию Flex SDK.

Чтобы запустить файл `VirtualZoo.swf`, сгенерированный компилятором mxmlc, просто откройте его в отдельной версии приложения Flash Player или в браузере, в котором установлено это приложение. Для просмотра сообщений программы, генерируемых функцией `trace( )`, используйте отладочную версию приложения Flash Player (поставляемую в составе инструментария Flex SDK) и сконфигурируйте ее для вывода таких сообщений в файл журнала. Дополнительную информацию можно получить по адресу **http://livedocs.adobe.com/flex/201/html/logging_125_07.html**.

В результате компиляции программы, следуя описанным в этом разделе шагам, вы увидите ряд предупреждений компилятора наподобие var 'pet' has no type declaration (переменная 'var' не имеет объявления типа). Пока вы можете просто проигнорировать эти предупреждения. В следующей главе будет описано, почему они возникают.

Ограничения компиляторов

Компилируя программы, написанные на языке ActionScript, с помощью среды разработки Flash, приложения Flex Builder или компилятора mxmlc, нужно принимать во внимание следующие ограничения компиляторов.

- Основной класс программы должен быть открытым.
- Для приложения Flex Builder и компилятора mxmlc основной класс программы должен находиться в безымянном пакете.
- Основной класс программы должен расширять либо класс `Sprite`, либо класс `MovieClip`, как было рассмотрено в гл. 6.
- Любой файл с исходным кодом ActionScript (AS-файл) должен иметь только одно определение, видимое извне. Таким определением могут являться класс, переменная, функция, интерфейс или пространство имен, описанные с помощью модификатора управления доступом `internal` или `public` внутри тела пакета.
- Название файла с исходным кодом ActionScript должно совпадать с именем единственного определения, видимого извне, которое находится в этом файле.

Например, следующий файл с исходным кодом будет считаться недопустимым, поскольку он содержит два класса, видимых извне:

```
package {
  public class A {
  }

  public class B {
  }
}
```

Подобным образом следующий файл с исходным кодом будет считаться недопустимым, поскольку он не содержит ни одного определения, видимого извне.

```
class C {
}
```

Процесс компиляции и путь к классам

Когда экспортируется SWF-файл, компилятор языка ActionScript создает список всех классов, которые необходимы данному файлу. В частности, в список требуемых классов включаются следующие.

- Все классы, явно или неявно связанные с основным классом программы.
- Для среды разработки Flash все классы, явно или неявно связанные с исходным FLA-файлом экспортируемого SWF-файла (то есть со сценариями кадров).

Компилятор ищет все AS-файлы с исходным кодом, соответствующие связанным классам, и компилирует каждый исходный файл в формат байт-кода, после чего помещает этот двоичный код в SWF-файл. Набор папок, в которых компилятор осуществляет поиск AS-файлов, называется *путем к классам*.

Файлы классов, не требующиеся для данного SWF-файла, но хранящиеся в файловой системе, не будут компилироваться в SWF-файл, однако если не будет найден файл требуемого класса, возникнет ошибка на этапе компиляции.

Любая среда разработки на языке ActionScript автоматически включает в путь к классам несколько папок, а также позволяет пользователю самостоятельно указывать папки, которые должны быть включены в этот путь. Например, среда разработки Flash автоматически включает в путь к классам папку, содержащую исходный FLA-файл для данного SWF-файла. Подобным образом приложение Flex Builder и компилятор mxmlc автоматически включают в путь к классам папку, содержащую основной класс программы. Инструкции по включению дополнительных папок в путь к классам можно найти в соответствующей документации по продукту.

Путь к классам иногда называют *путем сборки* или *исходным путем*.

Строгий режим компиляции в сравнении со стандартным режимом

Для компиляции программы, написанной на языке ActionScript, можно использовать два различных режима: *строгий* и *стандартный*.

В строгом режиме компилятор сообщает о большем количестве ошибок по сравнению со стандартным режимом. Дополнительные ошибки, появляющиеся при компиляции программы в строгом режиме, призваны помочь программистам выявить потенциальные источники проблем еще до того, как будет запущена программа. По этой причине во всех компиляторах компании Adobe строгий режим включен по умолчанию. Программисты, которые желают использовать динамические возможности языка ActionScript (рассматриваемые в гл. 15) или просто предпочитают решать проблемы (то есть отлаживать программу) на этапе выполнения, а не на этапе компиляции программы, могут выполнять компиляцию в стандартном режиме.

Следующие непроверенные моменты при программировании приведут к ошибкам на этапе компиляции только в том случае, если используется строгий режим; в стандартном режиме эти ошибки не возникнут.

- Передача в функцию неправильного количества параметров или параметров неверных типов (дополнительную информацию можно найти в гл. 8).
- Определение двух переменных или методов с одинаковым именем.
- Обращение к методам и переменным, не определенным на этапе компиляции (но которые могут быть определены на этапе выполнения с помощью методик, описанных в гл. 15).
- Присваивание значения несуществующей переменной экземпляра объекта, чей класс не является динамическим.
- Присваивание значения константной переменной за пределами инициализатора переменной или, в случае переменной экземпляра, за пределами метода-конструктора класса, содержащего определение данной переменной.
- Попытка удалить (с помощью оператора `delete`) метод экземпляра, переменную экземпляра, статический метод или статическую переменную.
- Сравнение двух выражений с несовместимыми типами (дополнительную информацию можно найти в разд. «Типы данных и аннотации типов» гл. 8).
- Присваивание значения переменной с объявленным типом, когда присваиваемое значение не является членом указанного типа (исключения из этого правила можно найти в разд. «Три особых случая строгого режима» гл. 8).
- Обращение к несуществующим пакетам.

Включение стандартного режима компиляции в приложении Flex Builder

Выполните следующие шаги, чтобы включить стандартный режим компиляции для проекта в приложении Flex Builder.

1. В окне Navigator (Навигация) выделите папку проекта.
2. Выберите команду меню Project ▸ Properties (Проект ▸ Свойства).
3. На странице ActionScript Compiler (Компилятор ActionScript) снимите флажок Enable strict type checking (Включить строгую проверку типов).

Включение стандартного режима компиляции в среде разработки Flash

Выполните следующие действия, чтобы включить стандартный режим компиляции для документа в среде разработки Flash.

1. Выберите команду меню File ▸ Publish Settings (Файл ▸ Настройки публикации).
2. На вкладке Flash появившегося окна Publish Settings (Настройки публикации) нажмите кнопку Settings (Параметры).
3. В области Errors (Ошибки) появившегося окна ActionScript 3.0 Settings (Параметры ActionScript 3.0) снимите флажок Strict Mode (Строгий режим).

Чтобы включить стандартный режим компиляции для компилятора mxmlc, присвойте параметру компилятора `strict` значение `false`.

Далее: типы данных

Мы сумели откомпилировать и запустить нашу программу по созданию виртуального зоопарка, однако разработка программы еще далека от завершения. Чтобы сделать зоопарк полностью интерактивным, добавить в него графику и кнопки для кормления животных, мы должны продолжить наше изучение основ языка ActionScript. В следующей главе будет рассказано, как система проверки типов языка ActionScript помогает выявить распространенные ошибки в программе.

Типы данных и проверка типов

До сих пор мы разрабатывали наш виртуальный зоопарк, не допустив ни одной ошибки кодирования. Разработка без ошибок происходит в обучающих курсах и книгах — и *больше нигде*. При разработке реальных программ программисты всегда допускают ошибки. Например, при вызове метода `eat( )` над объектом `VirtualPet` программист может допустить опечатку, как показано в следующем фрагменте кода (обратите внимание на лишнюю букву «t»):

```
pet.eatt(new Sushi( ))
```

Или же программист может сделать неправильное предположение о возможностях объекта. Например, он может по ошибке попытаться вызвать метод `jump( )` над объектом `VirtualPet`, хотя в классе `VirtualPet` такой метод не определен:

```
pet.jump( )
```

В обоих предыдущих случаях, если программа выполняется в отладочной версии среды выполнения Flash, произойдет *ошибка обращения* — она означает, что программа попыталась обратиться к несуществующей переменной или методу.

Ошибки, происходящие на этапе выполнения программы, называются исключениями. С исключениями и способами их обработки вы познакомитесь в гл. 13.

Когда ошибка возникает в разрабатываемой вами программе, вы должны быть счастливы. Ошибки указывают на точное место и причину какой-либо проблемы в вашей программе, которая в дальнейшем, скорее всего, приведет к сбою, если не будет вовремя исправлена. Например, в ответ на предыдущую опечатку `eatt( )` отладочная версия среды выполнения Flash отобразит на экране предупреждающее сообщение следующего содержания:

```
ReferenceError: Error #1069: Property eatt not found on
zoo.VirtualPet and there is no default value.
at VirtualZoo$iinit( )[C:\data\virtualzoo\src\VirtualZoo.as:8]
```

Это можно перевести следующим образом: Ошибка обращения: Ошибка #1069: Свойство eatt не найдено в объекте zoo.VirtualPet, и не указано значение по умолчанию.

Обратите внимание, что в сообщении об ошибке для обозначения переменной или метода используется термин *«свойство»*, рассмотренный в разд. «Члены и свойства» гл. 1.

Сообщение об ошибке указывает не только на имя файла, в котором возникла данная ошибка, но и на определенную строку кода, содержащую ошибку. Эта информация очень полезна.

Какими бы полезными ни были ошибки, происходящие на этапе выполнения, они обладают потенциальным недостатком. Данные ошибки возникают только

в тот момент, когда выполняется ошибочная строка кода. Таким образом, в очень большой программе до момента возникновения подобной ошибки может пройти слишком много времени. Например, если прохождение сюжетной компьютерной игры занимает 10 часов, то до появления ошибки, допущенной на последнем уровне, пройдет 10 часов!

К счастью, вместо того, чтобы ожидать возникновения ошибок обращения на этапе выполнения, мы можем поручить компилятору сообщать о таких ошибках на этапе компиляции еще до того, как будет запущена программа. Для этого используются *аннотации типов* в сочетании со строгим режимом компиляции.

Типы данных и аннотации типов

В языке ActionScript термин *«тип данных»* означает просто «набор значений». Язык ActionScript определяет три фундаментальных типа данных: `Null`, `void` и `Object`. Каждый из типов данных `Null` и `void` включает по одному значению — `null` и `undefined` соответственно (значения `null` и `undefined` рассматриваются далее в разд. «Значения null и undefined»). Тип данных `Object` включает все экземпляры всех классов.

Кроме трех фундаментальных типов данных (`Null`, `void` и `Object`), любой внутренний или пользовательский класс формирует уникальный тип данных, набором значений которого являются непосредственные экземпляры данного класса и экземпляры его классов-потомков. Например, класс `Food` из нашей программы по созданию виртуального зоопарка формирует тип данных, набором значений которого являются все экземпляры класса `Food` и все экземпляры классов `Apple` и `Sushi` (поскольку оба класса `Apple` и `Sushi` унаследованы от класса `Food`). Таким образом, говорят, что экземпляры классов `Apple` и `Sushi` принадлежат типу данных `Food`.

Однако каждый из классов `Apple` и `Sushi` также формирует собственный тип данных. Например, набором значений типа данных `Apple` являются все экземпляры класса `Apple` и все экземпляры любого класса, унаследованного от него. Подобным образом набором значений типа данных `Sushi` являются все экземпляры класса `Sushi` и все экземпляры любого класса, унаследованного от него. По этой причине помимо того, что экземпляр класса `Apple` принадлежит типу данных `Food`, он *также* принадлежит типу данных `Apple`. Однако экземпляр класса `Apple` *не* принадлежит типу данных `Sushi`, поскольку класс `Sushi` не унаследован от `Apple`. Точно так же экземпляр класса `Sushi` принадлежит типам данных `Food` и `Sushi`, но не принадлежит типу данных `Apple`, поскольку класс `Sushi` не унаследован от `Apple`. Наконец, несмотря на то, что экземпляры обоих классов принадлежат типу данных `Food`, экземпляр класса `Food` не принадлежит ни одному из типов данных `Apple` или `Sushi`, поскольку класс `Food` не унаследован от классов `Apple` или `Sushi`.

Обратите внимание на важное различие между отдельно взятым классом и типом данных, представляемым этим классом. Набором значений, принадлежащим данному классу, являются только экземпляры этого класса. Однако набором значений, принадлежащим типу данных этого класса, являются экземпляры данного класса и экземпляры его классов-потомков.

Подобно тому как каждый класс формирует тип данных, каждый *интерфейс* также формирует тип данных. Набором значений типа данных интерфейса являются все экземпляры любого класса, реализующего этот интерфейс, а также все экземпляры любого класса, унаследованного от класса, реализующего данный интерфейс. Мы еще не рассматривали интерфейсы, поэтому отложим разговор об их использовании в качестве типа данных до гл. 9.

Если у нас есть два типа данных — A и B, причем класс (или интерфейс), представленный типом данных B, унаследован от класса (или интерфейса), представленного типом данных A, то A называется *супертипом* для B. И наоборот, тип данных B называется *подтипом* A. Например, тип данных `Food` является супертипом для `Apple`, в то время как `Apple` является подтипом `Food`.

Совместимые типы

Поскольку любой отдельно взятый класс через наследование может использовать все незакрытые члены экземпляра своего суперкласса (или суперинтерфейса), любой отдельно взятый подтип считается *совместимым* с любым из своих супертипов. Например, тип данных `Apple` считается совместимым с типом данных `Food`, поскольку он является его подтипом.

Обратное, однако, неверно. Класс не может использовать никакие члены экземпляра, определенные в его классах-потомках. Таким образом, любой отдельно взятый супертип считается *несовместимым* с любым из своих подтипов. Например, тип данных `Food` считается несовместимым с типом данных `Apple`, поскольку `Food` не является подтипом `Apple`.

Подтип считается совместимым с супертипом, поскольку программа может рассматривать экземпляр подтипа как экземпляр супертипа. Например, программа может рассматривать любой экземпляр типа данных `Apple` как экземпляр типа данных `Food` — возможно, вызывая метод `getCalories( )` класса `Food` над данным экземпляром.

```
// Создаем новый экземпляр класса Apple
var apple = new Apple( );

// Допустимо вызвать метод getCalories( ) над экземпляром класса Apple
apple.getCalories( );
```

Для сравнения отметим, что супертип считается несовместимым с подтипом, поскольку программа *не может* рассматривать экземпляр супертипа как экземпляр подтипа. Например, программа не может вызвать метод `hasWorm( )` класса `Apple` над экземпляром типа данных `Food`:

```
// Создаем новый экземпляр класса Food
var food = new Food(200);

// Следующая строка приведет к возникновению ошибки обращения,
// поскольку класс Food не имеет доступа к методу hasWorm( )
food.hasWorm( );  // Ошибка!
```

Выявление ошибок несоответствия типов с помощью аннотаций типов

Аннотация типа (или *объявление типа*) — это суффикс, определяющий тип данных для переменной, параметра или возвращаемого функцией значения. Общим синтаксисом для аннотации типа является двоеточие (:), за которым указывается тип данных, как показано в следующем примере:

```
:тип
```

Например, определение переменной с использованием аннотации типа имеет следующий обобщенный вид:

```
var идентификатор:тип = значение;
```

В предыдущем коде *тип* должен быть именем класса или интерфейса (представляющего тип данных) либо специальным символом * (обозначающим «нетипизированные» данные).

Определение функции или метода с использованием аннотации типа параметра и возвращаемого значения имеет следующий обобщенный вид:

```
function идентификатор (параметр:типПараметра):типВозвращаемогоЗначения {
}
```

В предыдущем коде аннотация типа параметра задается с помощью указания типа *типПараметра*, перед которым ставится двоеточие (:); аннотация типа возвращаемого значения задается указанием типа *типВозвращаемогоЗначения*, перед которым также ставится двоеточие (:). Тип *типПараметра* должен быть одним из следующих:

- имя класса или интерфейса (представляющего тип данных);
- специальный символ * (обозначающий «нетипизированные» данные).

Тип *типВозвращаемогоЗначения* должен быть одним из следующих:

- имя класса или интерфейса (представляющего тип данных);
- специальный символ * (обозначающий «нетипизированные» данные);
- специальная, «не возвращающая значение», аннотация типа `void` (которая обозначает, что функция не имеет возвращаемого значения).

Программисты на языке ActionScript 2.0 должны обратить внимание, что в языке ActionScript 3.0 ключевое слово Void больше не записывается с прописной буквы.

Аннотация типа для переменной, параметра или результата функции ограничивает значение этой переменной, параметра или результата указанным типом. Способ ограничения значения зависит от используемого режима компиляции кода (как уже было сказано, в компиляторах компании Adobe по умолчанию выбран строгий режим компиляции).

В независимости от используемого режима компиляции — стандартного или строгого, — если значение принадлежит указанному типу данных, попытка присвоить или вернуть значение окажется успешной.

Если значение *не* принадлежит указанному типу данных, то при использовании строгого режима компилятор сгенерирует ошибку (называемую *ошибкой несоответствия типов*) и прекратит компиляцию кода. При использовании стандартного режима код будет скомпилирован и среда Flash попытается преобразовать значение в указанный тип данных на этапе выполнения программы. Если указанным типом данных является один из внутренних классов `String`, `Boolean`, `Number`, `int` или `uint` (называемых *примитивными типами*), преобразование будет выполнено в соответствии с правилами, описанными в разд. «Преобразование в примитивные типы» этой главы. В противном случае преобразование завершится неудачей и среда Flash сгенерирует ошибку несоответствия типов на этапе выполнения программы. На формальном языке автоматическое преобразование значения на этапе выполнения программы называется *приведением*.

Например, следующий код определяет переменную `meal` типа `Food` и присваивает этой переменной экземпляр класса `Apple`:

```
var meal:Food = new Apple( );
```

Указанный код скомпилируется успешно как в строгом режиме компиляции, так и в стандартном режиме, поскольку класс `Apple` расширяет `Food`. В итоге экземпляры класса `Apple` принадлежат типу данных `Food`.

В отличие от этого следующий код присваивает переменной `meal` экземпляр класса `VirtualPet`:

```
var meal:Food = new VirtualPet("Lucky");
```

При использовании строгого режима компиляции указанный код приведет к ошибке несоответствия типов, поскольку экземпляры класса `VirtualPet` не принадлежат типу данных `Food`. По этой причине компиляция кода прекратится.

При использовании стандартного режима указанный код будет успешно откомпилирован. Тем не менее, поскольку значение (экземпляр класса `VirtualPet`) не принадлежит типу данных переменной (`Food`), среда выполнения Flash попытается привести (то есть преобразовать) значение к типу данных переменной на этапе выполнения программы. В данном случае тип данных переменной не является одним из примитивных типов, поэтому преобразование завершится неудачей и среда Flash сгенерирует ошибку несоответствия типов на этапе выполнения программы.

Рассмотрим другой пример. Следующий код определяет переменную `petHunger` типа `int` и присваивает этой переменной экземпляр класса `VirtualPet`:

```
var pet:VirtualPet = new VirtualPet("Lucky");
var petHunger:int = pet;
```

При компиляции указанного кода с использованием строгого режима произойдет ошибка несоответствия типов, поскольку экземпляры класса `VirtualPet` не принадлежат типу данных `int`. По этой причине компиляция кода прекратится.

При использовании стандартного режима указанный код будет скомпилирован успешно. Тем не менее, поскольку значение (экземпляр класса `VirtualPet`) не принадлежит типу данных переменной (`int`), среда Flash попытается преобразовать значение в тип данных переменной на этапе выполнения программы. В данном слу-

чае тип данных переменной *является* одним из примитивных типов, поэтому преобразование будет выполнено в соответствии с правилами, описанными в разд. «Преобразование к примитивным типам» этой главы. Таким образом, после выполнения указанного кода значением переменной `petHunger` будет являться `0`.

Разумеется, предыдущий код наверняка разрабатывался *не* для того, чтобы переменной `petHunger` присвоить значение `0`. Скорее программист просто забыл вызвать метод `getHunger( )` над экземпляром класса `VirtualPet`, как показано в следующем коде:

```
var pet:VirtualPet = new VirtualPet("Lucky");
var petHunger:int = pet.getHunger( );
```

В строгом режиме компилятор честно предупредит нас о проблеме, а в стандартном режиме — нет, предположив, что, поскольку типом данных переменной `petHunger` является `int`, мы хотим преобразовать объект `VirtualPet` в тип `int`. В нашем случае это предположение оказалось неверным, в результате чего программа получила неожидаемое значение.

Некоторые программисты считают снисходительность стандартного режима удобной возможностью, особенно в простых приложениях. Однако в более сложных программах из-за гибкости стандартного режима предполагаемые ошибки зачастую скрываются, что в конечном итоге приводит к появлению труднораспознаваемых ошибок.

В оставшейся части книги мы будем полагать, что весь код компилируется в строгом режиме и для всех переменных, параметров и возвращаемых значений указываются аннотации типа.

Нетипизированные переменные, параметры, возвращаемые значения и выражения

Переменная или параметр, чье определение включает аннотацию типа, называются *типизированной* переменной или *типизированным* параметром. Подобным образом определение функции, которое включает аннотацию типа возвращаемого значения, называется определением, имеющим *типизированное возвращаемое значение*. Кроме того, выражение, которое обращается к типизированной переменной или типизированному параметру либо вызывает функцию с типизированным возвращаемым значением, называется *типизированным выражением*.

Напротив, переменная или параметр, чье определение не включает аннотацию типа, называется *нетипизированной* переменной или *нетипизированным* параметром. Подобным образом определение функции, которое не включает аннотацию типа возвращаемого значения, называется определением, имеющим *нетипизированное возвращаемое значение*. Кроме того, выражение, которое обращается к нетипизированной переменной или нетипизированному параметру либо вызывает функцию с нетипизированным возвращаемым значением, называется *нетипизированным выражением*.

Нетипизированные переменные, параметры и возвращаемые значения не ограничены определенным типом данных (в отличие от типизированных переменных, параметров и возвращаемых значений). Например, нетипизированной переменной можно присвоить значение типа `Boolean` в одной строке кода, а в следующей строке присвоить этой же переменной объект `VirtualPet` без каких-либо ошибок:

```
var stuff = true;
stuff = new VirtualPet("Edwin");  // Ошибки нет
```

Компилятор языка ActionScript не генерирует ошибки несоответствия типов для нетипизированных переменных, параметров и возвращаемых значений.

Если программист хочет явно указать, что переменная, параметр или возвращаемое значение намеренно являются нетипизированными, он может использовать специальную аннотацию типа `:*`. Например, следующий код определяет *явно* нетипизированную переменную `totalCost`:

```
var totalCost:* = 9.99;
```

Следующий код определяет ту же переменную, но на этот раз она является нетипизированной *неявно*:

```
var totalCost = 9.99;
```

Неявно нетипизированные переменные, параметры и возвращаемые значения обычно используются в тех случаях, когда в программе вообще не применяются аннотации типов, что дает программисту возможность обрабатывать любые типы ошибок на этапе выполнения. Явно нетипизированные переменные, параметры и возвращаемые значения обычно используются в тех случаях, когда программист желает явно указать место в программе, компилируемой в строгом режиме, где допустимо применение нескольких типов данных. Аннотация типа `:*` позволяет предотвратить появление предупреждения об «отсутствующей аннотации типа» для нетипизированной переменной. Более подробно этот вопрос будет рассмотрен в разд. «Предупреждения об отсутствующих аннотациях типов».

Три особых случая строгого режима

Существует три ситуации, в которых компилятор игнорирует ошибки несоответствия типов при компиляции программы в строгом режиме, не позволяя выявить возможные ошибки до этапа выполнения:

- когда нетипизированное выражение присваивается типизированной переменной или параметру либо возвращается из функции с объявленным типом возвращаемого значения;
- если любое выражение присваивается типизированной переменной или параметру, чьим объявленным типом является `Boolean`, либо возвращается из функции, объявленным типом возвращаемого значения которой является `Boolean`;
- когда любое числовое значение используется там, где должен быть указан экземпляр другого числового типа.

Рассмотрим каждый из описанных случаев на примере. Сначала создадим нетипизированную переменную pet и присвоим ее значение типизированной переменной d:

```
var pet:* = new VirtualPet("Francis");
pet = new Date( );
var d:Date = pet;
```

Поскольку переменная pet может содержать значение любого типа, в третьей строке компилятор не сможет определить, принадлежит ли значение переменной pet типу данных Date. Чтобы определить это, код должен быть не только откомпилирован, но и выполнен. Во время исполнения кода среда Flash сможет узнать результат попытки присваивания. В случае с предыдущим кодом значение, хранящееся в переменной pet (присвоенное во второй строке), на самом деле принадлежит типу данных Date (даже несмотря на то, что изначально значение переменной pet, присвоенное в первой строке кода, было несовместимо с типом данных Date). Таким образом, операция присваивания будет выполнена без ошибок.

Теперь рассмотрим следующий код, в котором определяется переменная b типа Boolean и этой переменной присваивается целочисленное значение 5:

```
var b:Boolean = 5;
```

Даже несмотря на то, что значение 5 не принадлежит типу данных Boolean, компилятор не генерирует ошибку несоответствия типов. Вместо этого он делает предположение, что программист желает привести значение 5 к типу данных Boolean (в соответствии с правилами, описанными в разд. «Преобразование в примитивные типы») и генерирует соответствующее предупреждение. Такая «мягкость» компилятора позволяет сократить количество кода в программе. Например, предположим, что в качестве типа возвращаемого значения метода getHunger( ) класса VirtualPet указан тип данных Number. Программа может создать переменную, содержащую информацию о том, является животное живым или мертвым, используя следующий код:

```
var isAlive:Boolean = somePet.getHunger( );
```

В соответствии с правилами, описанными в разд. «Преобразование в примитивные типы», число 0 преобразуется в значение false, а остальные числа — в значение true. Таким образом, если метод getHunger( ) возвращает любое значение, отличное от 0, переменной isAlive присваивается значение true; в противном случае переменной isAlive присваивается значение false (животное оказывается мертвым, когда у него не остается калорий).

Для сравнения приведем альтернативный, чуть более длинный код, который пришлось бы использовать в том случае, если бы компилятор проверял типы для переменных типа Boolean (не позволяя преобразовывать их на этапе выполнения программы):

```
var isAlive:Boolean = somePet.getHunger( ) > 0;
```

Наконец, рассмотрим код, в котором определяется переменная xCoordinate типа int и этой переменной присваивается значение 4,6459 типа Number:

```
var xCoordinate:int = 4.6459;
```

Даже несмотря на то, что значение `4,6459` не принадлежит типу данных `int`, компилятор не генерирует ошибку несоответствия типов. Вместо этого он делает предположение, что вы желаете преобразовать значение `4,6459` к типу данных `int` (в соответствии с правилами, описанными в разд. «Преобразование в примитивные типы»). Это позволяет максимально упростить взаимодействие между числовыми типами данных языка ActionScript.

Предупреждения об отсутствующих аннотациях типов

Как известно из предыдущих разделов, строгий режим компиляции языка ActionScript предоставляет чрезвычайно полезную возможность выявления ошибок программы на самых ранних стадиях.

Неудивительно, что в своем стремлении создавать безошибочный код многие программисты очень полагаются на проверку типов, выполняемую в строгом режиме на этапе компиляции. Однако, как известно из разд. «Нетипизированные переменные, параметры, возвращаемые значения и выражения», ошибки несоответствия типов, выявляемые при использовании строгого режима компиляции, генерируются только для типизированных переменных, параметров и возвращаемых значений. Всякий раз, когда программист случайно пропускает аннотацию типа, он теряет преимущество проверки типов, выполняемой в строгом режиме на этапе компиляции.

К счастью, компиляторы языка ActionScript компании Adobe предлагают режим предупреждений, при использовании которого на этапе компиляции сообщается обо всех отсутствующих аннотациях типов. Разработчики могут использовать эти предупреждения для поиска случайно пропущенных аннотаций типов. В приложении Flex Builder и компиляторе mxmlc предупреждения об отсутствующих аннотациях типов включены по умолчанию. В среде разработки Flash предупреждения об отсутствующих типах должны быть включены вручную, для чего используется такая последовательность действий.

1. Используя любой текстовый редактор, откройте файл `EnabledWarnings.xml`, находящийся в папке **/en/Configuration/ActionScript 3.0**, которая расположена внутри папки приложения Flash CS3.
2. Найдите следующие строки:
```
<warning id="1008" enabled="false" label="kWarning_NoTypeDecl">
  Missing type declaration.</warning>
```
3. Измените `enabled="false"` на `enabled="true"`.
4. Сохраните файл `EnabledWarnings.xml`.

Обратите внимание, что предупреждения об отсутствующих аннотациях типов генерируются только для неявно нетипизированных переменных, параметров и возвращаемых значений. Для явно нетипизированных (то есть для тех, которые объявлены с использованием специальной аннотации типа `:*`) предупреждения об отсутствующих аннотациях типов не генерируются.

Выявление ошибок обращения на этапе компиляции

В начале этой главы рассказывалось, что попытка обратиться к несуществующим переменной или методу приведет к возникновению ошибки обращения. Если для компиляции программы используется стандартный режим, то компилятор не сообщает об ошибках обращения. Вместо этого, когда программа выполняется в отладочной версии среды Flash, ошибки обращения проявляются в виде исключений. Напротив, если для компиляции программы используется строгий режим, компилятор *сообщает* об обращениях к несуществующим переменным и методам в типизированных выражениях и прекращает компиляцию.

Например, в следующем коде создается переменная `pet` типа `VirtualPet` и этой переменной присваивается экземпляр класса `VirtualPet`:

```
var pet:VirtualPet = new VirtualPet("Stan");
```

Затем в следующем коде происходит попытка обращения к несуществующему методу `eatt( )` через типизированную переменную `pet`:

```
pet.eatt(new Sushi( ));
```

В стандартном режиме предыдущий код успешно откомпилируется, однако на этапе выполнения программы произойдет ошибка обращения. В строгом режиме предыдущий код приведет к генерации следующей ошибки обращения на этапе компиляции и процесс компиляции будет завершен.

```
1061: Call to a possibly undefined method eatt through a
  reference with static type zoo:VirtualPet.
```

На русском языке текст ошибки будет следующим: Вызов, вероятно, неопределенного метода eatt через ссылку статического типа zoo:VirtualPet.

Однако стоит отметить, что компилятор не сообщает об ошибках обращения, происходящих в нетипизированных выражениях. Более того, обращения к несуществующим переменным и методам через экземпляры динамических классов (таких как `Object`) не приводят к генерации вообще никаких ошибок обращения; вместо этого в результате возвращается значение `undefined`.

Дополнительную информацию о динамических классах можно найти в гл. 15.

Существует дополнительное преимущество при использовании аннотаций типов: в приложении Flex Builder и среде разработки Flash аннотации типов для переменных, параметров и возвращаемых значений активизируют подсказки кода. Подсказка кода представляет собой удобное всплывающее меню, содержащее список свойств и методов объектов, которые можно добавить в ваш код путем выбора требуемого элемента.

Приведение типов

В предыдущем разделе рассказывалось, что в строгом режиме компилятор сообщает об ошибках обращения на этапе компиляции. Для выявления ошибок обращения

компилятор полагается на аннотации типов. Предположим, что компилятор встретил обращение к методу, выполненное через типизированную переменную. Чтобы выяснить, является ли данное обращение корректным, компилятор проверяет наличие описания вызываемого метода в классе или интерфейсе, указанном в аннотации типа данной переменной. Если метод, к которому происходит обращение, не определен в указанном классе или интерфейсе, компилятор генерирует ошибку обращения.

Обратите внимание, что необходимость генерации ошибки обращения определяется исходя из класса или интерфейса, указанного в аннотации типа, а не фактического класса значения, хранящегося в данной переменной.

Рассмотрим следующий код, в котором метод `hasWorm( )` вызывается над объектом `Apple` через переменную типа `Food`:

```
var meal:Food = new Apple( );
meal.hasWorm( ); // Попытка вызвать метод hasWorm( ) над объектом,
                 // хранящимся в переменной meal
```

При компиляции предыдущего кода в строгом режиме компилятор должен решить, может ли метод `hasWorm( )` быть вызван над значением переменной `meal`. Для этого компилятор проверяет, определен ли в классе `Food` (то есть в классе, который указан в аннотации типа переменной `meal`) метод `hasWorm( )`. В этом классе определение данного метода отсутствует, поэтому компилятор генерирует ошибку обращения. Конечно, глядя на этот код, *мы* знаем, что значение переменной `meal` (объект `Apple`) поддерживает метод `hasWorm( )`. *Однако этого не знает компилятор.* Среда выполнения Flash только на этапе выполнения узнает, что значением переменной на самом деле является объект `Apple`.

Каково же решение? Использовать *операцию приведения типов*, чтобы заставить компилятор разрешить предыдущий вызов метода `hasWorm( )`. Операция приведения типов приказывает компилятору рассматривать данное значение как принадлежащее указанному типу данных. Эта операция имеет следующий обобщенный вид:

тип(*выражение*)

В этом коде *тип* — это любой тип данных, а *выражение* — любое выражение. Говоря простым языком, эта операция «приводит *выражение* к указанному типу *тип*». Например, следующий код приводит выражение `meal` к типу данных `Apple` перед вызовом метода `hasWorm( )` над значением переменной `meal`:

```
Apple(meal).hasWorm( )
```

Независимо от действительного значения переменной `meal` компилятор верит, что типом данных выражения `meal` является тип `Apple`. По этой причине, принимая решение о возможности вызова метода `hasWorm( )` над значением переменной `meal`, компилятор проверяет, определен ли метод `hasWorm( )` в классе `Apple`, но *не* в классе `Food`. Метод `hasWorm( )` определен в классе `Apple`, и следовательно, компилятор не генерирует никаких ошибок.

Однако операция приведения типов используется не только на этапе компиляции. На этапе выполнения программы ей также присуще определенное поведение: если выражение *выражение* преобразуется в объект, который принадлежит указанному типу

тип, то среда Flash просто возвращает этот объект. Однако если выражение *выражение* преобразуется в объект, который *не* принадлежит указанному типу *тип*, возможен один из двух вариантов завершения операции приведения типов. Если указанный тип *тип* не является примитивным, операция приведения типов вызовет ошибку на этапе выполнения; в противном случае значение объекта преобразуется в указанный тип (в соответствии с правилами, перечисленными в разд. «Преобразование в примитивные типы»), и результатом операции будет преобразованное значение.

Например, в следующем коде значение переменной `meal`, сформированное на этапе выполнения программы, принадлежит типу данных `Apple`, поэтому операция приведения типов во второй строке просто вернет объект `Apple`, на который ссылается переменная `meal`:

```
var meal:Food = new Apple( );
Apple(meal);  // На этапе выполнения вернет объект Apple
```

Для сравнения, в следующем коде значение переменной `meal`, сформированное на этапе выполнения программы, не принадлежит типу данных `VirtualPet`, и, поскольку тип данных `VirtualPet` не является примитивным, операция приведения типов во второй строке вызовет ошибку типа:

```
var meal:Food = new Apple( );
VirtualPet(meal);  // На этапе выполнения будет вызвана ошибка типа
```

Наконец, в следующем коде значение переменной `meal`, сформированное на этапе выполнения программы, не принадлежит типу данных `Boolean`, но, поскольку тип данных `Boolean` является примитивным, операция приведения типов во второй строке преобразует значение переменной `meal` в указанный тип и вернет результат данного преобразования (`true`):

```
var meal:Food = new Apple( );
Boolean(meal);  // На этапе выполнения будет возвращено значение true
```

Избавление от нежелательных ошибок несоответствия типов

Как уже известно, операции приведения типов могут быть использованы для выявления нежелательных ошибок обращения на этапе компиляции. Подобным образом операции приведения типов могут применяться для избавления от нежелательных ошибок несоответствия типов.

В качестве примера представьте программу, которая преобразует температуру, заданную по шкале Фаренгейта, в температуру по шкале Цельсия. Значение температуры по шкале Фаренгейта вводится в текстовое поле, представленное экземпляром внутреннего класса `TextField`. Для получения введенного значения мы обращаемся к переменной `text` экземпляра класса `TextField`, как показано в следующем коде:

```
var fahrenheit:Number = inputField.text;
```

В данных условиях такой код вызовет ошибку несоответствия типов, поскольку типом данных переменной `text` является `String`. Чтобы избавиться от этой ошибки, мы используем операцию приведения типов, как показано в следующем коде:

```
var fahrenheit:Number = Number(inputField.text);
```

На этапе выполнения программы этот код преобразует строковое значение, хранящееся в переменной `inputField.text`, в тип `Number` и присвоит преобразованное значение переменной `fahrenheit`.

Восходящее и нисходящее приведения типов

Приведение типа объекта к одному из его супертипов (суперклассу или суперинтерфейсу) называется *восходящим приведением типов*. Например, следующая операция осуществляет восходящее приведение типов, поскольку тип данных `Food` является супертипом типа `Apple`:

```
Food(new Apple( ))
```

И наоборот, приведение типа объекта к одному из его подтипов (подклассу или подинтерфейсу) называется *нисходящим приведением*, поскольку эта операция выполняет приведение типа объекта к типу, который находится ниже текущего в иерархии. Следующая операция осуществляет нисходящее приведение типов, поскольку тип данных `Apple` является подтипом типа `Food`:

```
Apple(new Food( ))
```

Говорят, что восходящее приведение типов «расширяет» тип объекта, поскольку супертип является более обобщенным по сравнению с его подтипом. В то же время нисходящее приведение типов «сужает» тип объекта, поскольку подтип является более специализированным, чем его супертип.

Кроме того, восходящее приведение типов считается *безопасным*, поскольку оно никогда не вызывает ошибку на этапе выполнения. Как было сказано ранее, экземпляр подтипа можно всегда рассматривать как экземпляр любого из его супертипов, потому что он гарантированно (через наследование) обладает всеми методами и переменными экземпляра своих супертипов в том случае, если эти методы и переменные объявлены с использованием любых модификаторов управления доступом, кроме `private`.

И наоборот, нисходящее приведение типов считается *небезопасным*, поскольку оно способно вызвать ошибку на этапе выполнения. Чтобы гарантировать, что операция нисходящего приведения типов не вызовет ошибку на этапе выполнения, перед приведением типов мы должны убедиться в фактической принадлежности рассматриваемого объекта к целевому типу данных. Для выполнения проверки типа данных объекта используется оператор `is`, имеющий следующий вид:

выражение `is` *тип*

В предыдущем коде *выражение* обозначает любое выражение, а *тип* — любой класс или интерфейс (*тип* не должен принимать значения `undefined` или `null`). Оператор `is` возвращает значение `true`, если заданное выражение принадлежит указанному типу *тип*, в противном случае возвращается значение `false`.

В следующем коде используется оператор `is`, чтобы гарантировать, что нисходящее приведение типов не вызовет ошибку на этапе выполнения:

```
var apple:Food = new Apple( );
if (apple is Apple) {
```

```
  Apple(apple).hasWorm( );
}
```

В предыдущем коде блок условного оператора будет выполнен только в том случае, если переменная apple ссылается на объект, принадлежащий типу Apple. По этой причине операция приведения типов Apple(apple) никогда не вызовет ошибку.

Использование оператора as для приведения к типам Date и Array

По многим причинам, связанным с поддержкой старого кода в языке ActionScript 3.0, синтаксис приведения типов, описанный в предыдущих разделах, не может быть использован для приведения типа значения к внутренним классам Date и Array. Результат выражения Date(*некотороеЗначение*) идентичен результату выражения new Date( ).toString( ) (оно возвращает строковое представление текущего времени). Результат выражения Array(*некотороеЗначение*) идентичен результату выражения new Array(*некотороеЗначение*) (оно создает новый объект Array, первым элементом которого является значение *некотороеЗначение*).

Для приведения типа результата выражения либо к классу Date, либо к классу Array используется оператор as, который действует точно так же, как операция приведения типов, но с одним исключением — данный оператор возвращает значение null в тех случаях, когда операция приведения типов вызывает ошибку на этапе выполнения. Оператор as имеет следующий вид:

выражение as *тип*

В приведенном коде *выражение* представляет любое выражение, а *тип* — любой класс или интерфейс (*тип* не должен принимать значения undefined или null). Оператор as возвращает значение выражения *выражение*, если указанное *выражение* принадлежит указанному типу *тип*, в противном случае возвращается значение null.

Например, в следующем коде результат выражения (meal as Apple) идентичен результату операции приведения типов Apple(meal):

```
var meal:Food = new Apple( );
(meal as Apple).hasWorm( );
```

В следующем коде оператор as используется для «приведения» объекта Array к типу данных Array с тем, чтобы данный объект можно было присвоить переменной типа Array.

```
public function output (msg:Object):void {
  if (msg is String) {
    trace(msg);
  }

  if (msg is Array) {
    var arr:Array = msg as Array;  // Приведение к типу Array
    trace(arr.join("\n"));
  }
}
```

Следующий код демонстрирует результат передачи тестового объекта `Array` в метод `output( )`:

```
var numbers:Array = [1,2,3]
output(numbers);

// Вывод:
1
2
3
```

Преобразование в примитивные типы

В предыдущем разделе рассказывалось, что если выражение приводится к примитивному типу, к которому оно не принадлежит, то его значение будет преобразовано в указанный тип. Например, рассмотрим следующий код, который выполняет приведение объекта `Date` к примитивному типу данных `Boolean`:

```
Boolean(new Date( ))
```

Поскольку тип данных `Boolean` является примитивным типом, а объект `Date` не принадлежит типу `Boolean`, среда выполнения Flash преобразует значение объекта `Date` в тип `Boolean`. Результатом данного преобразования будет являться значение `true` типа `Boolean`.

Операции приведения типов иногда используются не для того, чтобы сообщить компилятору тип заданного выражения, а для того, чтобы преобразовать значение этого выражения в примитивный тип данных.

Операция приведения типов может преобразовать любое значение в конкретный примитивный тип.

Например, следующий код преобразует число с плавающей точкой (с дробной частью) в целое число (без дробной части):

```
int(4.93)
```

Результатом этой операции приведения типов является целое число `4`. Подобным образом следующий код преобразует значение `true` типа `Boolean` в целое число `1`, а значение `false` типа `Boolean` — в целое число `0`:

```
int(true);  // Возвращает 1
int(false);  // Возвращает 0
```

Такая методика может применяться для уменьшения размера передаваемых на сервер данных, включающих ряд значений типа `Boolean`.

В табл. 8.1 представлены результаты преобразования различных типов данных в тип `Number`.

Таблица 8.1. Преобразование в тип Number

Исходные данные	Результат после преобразования
undefined	NaN (специальное числовое значение «Не число» (Not a Number), представляющее некорректные числовые данные)

Исходные данные	Результат после преобразования
null	0
int	То же число
uint	То же число
Boolean	1, если исходным значением является true; 0, если исходным значением является false
Numeric в строковом представлении	Эквивалентное числовое значение, если строка состоит только из цифр десятичной или шестнадцатеричной систем счислений, пробела, экспоненты, десятичной точки, знаков + или – (например, строка ("–1.485e2") превратится в число –148.5)
Пустая строка	0
«Бесконечность»	Infinity
«Минус бесконечность»	–Infinity
Другие строки	NaN
Объект	NaN

В табл. 8.2 представлены результаты преобразования различных типов данных в тип `int`.

Таблица 8.2. Преобразование к типу int

Исходные данные	Результат после преобразования
undefined	0
null	0
Number или uint	Целое число в диапазоне от –231 до 231–1; значения, превышающие диапазон представления, включаются в указанный диапазон с помощью алгоритма, описанного в разд. 9.5 третьей редакции стандарта ECMA-262
Boolean	1, если исходным значением является true; 0, если исходным значением является false
Numeric в строковом представлении	Эквивалентное числовое значение, преобразованное в целочисленный знаковый формат
Пустая строка	0
«Бесконечность»	0
«Минус бесконечность»	0
Другие строки	0
Объект	0

В табл. 8.3 представлены результаты преобразования различных типов данных в тип `uint`.

Таблица 8.3. Преобразование в тип uint

Исходные данные	Результат после преобразования
undefined	0
null	0
Number или int	Целое число в диапазоне от 0 до 231–1; значения, превышающие диапазон представления, включаются в указанный диапазон с помощью алгоритма, описанного в разд. 9.6 третьей редакции стандарта ECMA-262

Продолжение ⇩

Таблица 8.3 (продолжение)

Исходные данные	Результат после преобразования
Boolean	1, если исходным значением является true; 0, если исходным значением является false
Numeric в строковом представлении	Эквивалентное числовое значение, преобразованное в целочисленный беззнаковый формат
Пустая строка	0
«Бесконечность»	0
«Минус бесконечность»	0
Другие строки	0
Объект	0

В табл. 8.4 представлены результаты преобразования различных типов данных в тип `String`.

Таблица 8.4. Преобразование в тип String

Исходные данные	Результат после преобразования
undefined	"undefined"
null	"null"
Boolean	"true", если исходным значением является true; "false", если исходным значением является false
NaN	"NaN"
0	"0"
«Бесконечность»	"Infinity"
«Минус бесконечность»	"–Infinity"
Другое числовое значение	Строковое представление указанного числа. Например, число 944.345 превратится в строку "944.345"
Объект	Значение, полученное в результате вызова метода toString( ) над объектом. По умолчанию метод объекта toString( ) возвращает строку "[object имяКласса]", где имяКласса представляет класс объекта. Метод toString( ) может быть переопределен для получения более полезной информации. Например, метод toString( ) объекта Date возвращает время в удобочитаемом формате наподобие "Sun May 14 11:38:10 EDT 2000", а метод toString( ) объекта Array возвращает список элементов массива, разделенных запятыми

В табл. 8.5 представлены результаты преобразования различных типов данных в тип `Boolean`.

Таблица 8.5. Преобразование в тип Boolean

Исходные данные	Результат после преобразования
undefined	false
null	false
NaN	false
0	false
Infinity	true
–Infinity	true
Другое числовое значение	true
Непустая строка	true

Исходные данные	Результат после преобразования
Пустая строка ("")	false
Объект	true

Значения переменных по умолчанию

Когда переменная объявлена *без* аннотации типа и без указания исходного значения, в качестве исходного значения этой переменной автоматически присваивается значение `undefined` (единственное значение типа данных `void`). Когда переменная объявлена *с* аннотацией типа, но без указания исходного значения, в качестве ее исходного значения автоматически присваивается значение по умолчанию, соответствующее указанному типу данных.

В табл. 8.6 перечислены значения по умолчанию для каждого конкретного типа данных в языке ActionScript, используемые при объявлении переменных.

Таблица 8.6. Значения переменных по умолчанию

Тип данных	Значение по умолчанию
String	null
Boolean	false
int	0
uint	0
Number	NaN
Все остальные типы	null

Значения null и undefined

В одной из предыдущих глав рассказывалось, что каждый из типов данных — `Null` и `void` — включает по одному-единственному значению — `null` и `undefined` соответственно. Теперь, когда мы познакомились с типами данных и аннотациями типов, рассмотрим, чем же отличаются эти два значения.

Концептуально оба значения — `null` и `undefined` — обозначают отсутствие данных. Значение `null` обозначает отсутствие данных для переменных, параметров и возвращаемых значений, объявленных с использованием аннотаций типов, кроме типов `Boolean`, `int`, `uint` и `Number`. Например, следующий код создает типизированную переменную экземпляра `pet` типа `VirtualPet`. До тех пор пока переменной не будет явно присвоено значение в программе, ее значением будет являться `null`.

```
package {
  import flash.display.Sprite;
  import zoo.*;

  public class VirtualZoo extends Sprite {
    private var pet:VirtualPet;
```

```
    public function VirtualZoo ( ) {
      trace(pet);  // Выводит: null
    }
  }
}
```

Напротив, значение undefined обозначает отсутствие данных для переменных, параметров и возвращаемых значений, объявленных *без* использования аннотаций типов. Например, следующий код создает объект с двумя динамическими переменными экземпляра — city и country. Этот код использует значение undefined при присваивании переменной country исходного значения, чтобы показать, что она пока не имеет осмысленного значения.

```
var info = new Object( );
info.city = "Toronto";
info.country = undefined;
```

Кроме того, значение undefined обозначает полное отсутствие переменной или метода у объекта, чей класс объявлен с использованием атрибута dynamic. Например, в результате следующей попытки обратиться к несуществующей переменной объекта, на который ссылается переменная info, будет возвращено значение undefined:

```
trace(info.language);  // Выводит: undefined
```

Более подробно динамические возможности языка ActionScript и значение undefined будут рассмотрены в гл. 15.

Типы данных в программе по созданию виртуального зоопарка

Теперь, когда мы получили всю информацию о типах данных, добавим аннотации типов в нашу программу «Зоопарк». В листинге 8.1 представлен измененный код класса VirtualZoo — основного класса программы.

Листинг 8.1. Класс VirtualZoo

```
package {
  import flash.display.Sprite;
  import zoo.*;

  public class VirtualZoo extends Sprite {
    private var pet:VirtualPet;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      pet.eat(new Apple( ));
      pet.eat(new Sushi( ));
    }
  }
```

Листинг 8.2 демонстрирует код класса `VirtualPet`, экземпляры которого представляют животных в зоопарке. Обратите внимание на использование операции приведения типов в методе `eat ( )`, рассмотренной в подразд. «Восходящее и нисходящее приведения типов» разд. «Приведение типов».

Листинг 8.2. Класс VirtualPet

```
package zoo {
  import flash.utils.setInterval;
  import flash.utils.clearInterval;

  internal class VirtualPet {
    private static var maxNameLength:int = 20;
    private static var maxCalories:int = 2000;
    private static var caloriesPerSecond:int = 100;

    private var petName:String;
    private var currentCalories:int = VirtualPet.maxCalories/2;
    private var digestIntervalID:int;

    public function VirtualPet (name:String):void {
      setName(name);
      digestIntervalID = setInterval(digest, 1000);
    }

    public function eat (foodItem:Food):void {
      if (currentCalories == 0) {
        trace(getName( ) + " is dead. You can't feed it.");
        return;
      }

      if (foodItem is Apple) {
        // Обратите внимание на приведение к типу Apple
        if (Apple(foodItem).hasWorm( )) {
          trace("The " + foodItem.getName( ) + " had a worm. " + getName( )
                + " didn't eat it.");
          return;
        }
      }

      var newCurrentCalories:int = currentCalories + foodItem.getCalories( );
      if (newCurrentCalories > VirtualPet.maxCalories) {
        currentCalories = VirtualPet.maxCalories;
      } else {
        currentCalories = newCurrentCalories;
      }
      trace(getName( ) + " ate some " + foodItem.getName( ) + "."
            + " It now has " + currentCalories + " calories remaining.");
    }

    public function getHunger ( ):Number {
      return currentCalories / VirtualPet.maxCalories;
    }
```

```
    public function setName (newName:String):void {
      // Если длина заданного нового имени
      // больше maxNameLength символов...
      if (newName.length > VirtualPet.maxNameLength) {
        // ...обрезать имя
        newName = newName.substr(0, VirtualPet.maxNameLength);
      } else if (newName == "") {
        // ...в противном случае, если заданное новое имя является
        // пустой строкой, завершить выполнение метода, не изменяя
        // значения переменной petName
        return;
      }

      // Присвоить новое проверенное имя переменной petName
      petName = newName;
    }

    public function getName ( ):String {
      return petName;
    }

    private function digest ( ):void {
      // Если в результате потребления очередной порции калорий
      // значение переменной currentCalories животного
      // станет равным 0 или меньше...
      if (currentCalories - VirtualPet.caloriesPerSecond <= 0) {
        // ...прекратить вызов метода digest( )
        clearInterval(digestIntervalID);
        // После чего очистить желудок животного
        currentCalories = 0;
        // и сообщить о смерти животного
        trace(getName( ) + " has died.");
      } else {
        // ...иначе употребить оговоренное количество калорий
        currentCalories -= VirtualPet.caloriesPerSecond;

        // и сообщить о новом состоянии животного
        trace(getName( ) + " digested some food. It now has "
              + currentCalories + " calories remaining.");
      }
    }
  }
}
```

Листинг 8.3 демонстрирует код класса `Food`, являющегося суперклассом для различных видов пищи, принимаемой животными.

Листинг 8.3. Класс Food

```
package zoo {
  public class Food {
    private var calories:int;
```

```
    private var name:String;
    public function Food (initialCalories:int) {
      setCalories(initialCalories);
    }

    public function getCalories ( ):int {
      return calories;
    }

    public function setCalories (newCalories:int):void {
      calories = newCalories;
    }

    public function getName ( ):String {
      return name;
    }

    public function setName (newName:String):void {
      name = newName;
    }
  }
}
```

Листинг 8.4 демонстрирует код класса `Apple`, представляющего конкретный вид пищи, принимаемой животными.

Листинг 8.4. Класс Apple

```
package zoo {
  public class Apple extends Food {
    private static var DEFAULT_CALORIES:int = 100;
    private var wormInApple:Boolean;

    public function Apple (initialCalories:int = 0) {
      if (initialCalories <= 0) {
        initialCalories = Apple.DEFAULT_CALORIES;
      }
      super(initialCalories);

      wormInApple = Math.random( ) >= .5;

      setName("Apple");
    }

    public function hasWorm ( ):Boolean {
      return wormInApple;
    }
  }
}
```

Наконец, листинг 8.5 демонстрирует код класса `Sushi`, представляющего конкретный вид пищи, принимаемой животными.

Листинг 8.5. Класс Sushi

```
package zoo {
  public class Sushi extends Food {
    private static var DEFAULT_CALORIES:int = 500;

    public function Sushi (initialCalories:int = 0) {
      if (initialCalories <= 0) {
        initialCalories = Sushi.DEFAULT_CALORIES;
      }
      super(initialCalories);

      setName("Sushi");
    }
  }
}
```

Далее: дополнительное изучение типов данных

В этой главе было рассказано, как использование типов данных помогает выявить и устранить потенциальные проблемы в программе. В следующей главе мы завершим наше обзорное изучение типов данных, рассмотрев *интерфейсы*. Интерфейсы, как и классы, применяются для создания пользовательских типов данных.

Интерфейсы

Интерфейс — это конструкция языка ActionScript, которая описывает новый тип данных подобно описанию типа данных с помощью класса. Однако, тогда как класс не только описывает тип данных, но и предоставляет для него реализацию, интерфейс только описывает тип данных в абстрактных терминах и не предоставляет реализацию для этого типа данных. Иными словами, класс не только объявляет группу методов и переменных, но и реализует определенное поведение; тела методов и значения переменных управляют поведением класса. Вместо того чтобы предоставлять собственную реализацию, интерфейс принимается одним или несколькими классами, которые согласны предоставить для него реализацию. Экземпляры класса, предоставляющего реализацию для интерфейса, принадлежат как типу данных класса, так и типу данных, описанному интерфейсом. Являясь одновременно членом нескольких типов данных, экземпляры могут выполнять в приложении различные функции.

Не путайте термин «интерфейс», обсуждаемый в данной главе, с другими применениями этого слова. В этой главе «интерфейс» обозначает конструкцию языка ActionScript, а не графический интерфейс пользователя (GUI) или открытый API класса, которые в общей теории объектно-ориентированного программирования иногда именуются интерфейсами.

Если это ваше первое знакомство с интерфейсами, то их теоретические описания могут оказаться сложными для восприятия, поэтому сразу же рассмотрим пример.

Аргумент в пользу интерфейсов

Предположим, мы создаем протоколирующий класс `Logger`, который рапортует о статусных сообщениях («записях журнала») программы в процессе ее выполнения. Многие классы получают статусные сообщения от класса `Logger` и по-разному реагируют на них. Например, один класс — `LogUI` — отображает журнальные сообщения на экране, другой класс — `LiveLog` — отправляет предупреждение специалисту службы поддержки с помощью сетевого инструмента администрирования, а еще один класс — `LogTracker` — добавляет журнальные сообщения в базу для ведения статистики. Для получения журнальных сообщений каждый класс определяет метод `update( )`. Чтобы отправить сообщение объектам всех заинтересованных классов, класс `Logger` вызывает метод `update( )`.

Пока все это кажется логичным, но что произойдет, если мы забудем определить метод `update( )` в классе `LogUI`? Статусное сообщение будет отправлено, однако объекты класса `LogUI` не получат его. Нам необходим такой подход, который

гарантирует, что каждый получатель журнальных сообщений определяет метод update( ).

Чтобы предоставить такую гарантию, предположим, что мы ввели новое требование в нашу программу: любой объект, желающий получать журнальные сообщения от класса Logger, должен быть экземпляром базового класса LogRecipient (мы предоставим его описание) или экземпляром одного из подклассов класса LogRecipient. Реализация метода update( ) в классе LogRecipient будет включать базовую функциональность, попросту отображая журнальное сообщение с помощью функции trace( ):

```
public class LogRecipient {
  public function update (msg:String):void {
    trace(msg);
  }
}
```

Теперь любой класс, который желает получать журнальные сообщения от класса Logger, просто расширяет класс LogRecipient и, если требуется специфическое поведение, перекрывает метод update( ) класса LogRecipient, реализуя желаемое поведение. Например, следующий класс LogTracker расширяет класс LogRecipient и перекрывает метод update( ), реализуя функциональность сохранения информации в базе данных:

```
public class LogTracker extends LogRecipient {
  // Перекрытый метод update( ) класса LogRecipient
  override public function update (msg:String):void {
    // Сохранение сообщения об ошибке в базе данных. Код не показан...
  }
}
```

Возвращаясь к классу Logger, мы определим метод addRecipient( ), который выполняет регистрацию объекта, желающего получать журнальные сообщения. Базовое описание метода addRecipient( ) представлено ниже. Обратите внимание, что в метод addRecipient( ) могут передаваться только экземпляры класса LogRecipient и его подклассов:

```
public class Logger {
  public function addRecipient (lr:LogRecipient):Boolean {
    // Размещаемый здесь код должен выполнять регистрацию объекта lr
    // для получения статусных сообщений и возвращать значение типа Boolean,
    // которое отражает результат выполненной операции (код не показан).
  }
}
```

Если передаваемый в метод addRecipient( ) объект не принадлежит типу LogRecipient, компилятор сгенерирует ошибку несоответствия типов. Если этот объект является экземпляром подкласса класса LogRecipient, в котором не реализован метод update( ), то по крайней мере будет выполнена базовая версия метода update( ) (определенная в классе LogRecipient).

Приемлемый вариант, не правда ли? Почти. Однако есть проблема. Что делать в том случае, если класс, желающий получать события от класса LogRecipient,

уже расширяет другой класс? Например, предположим, что класс LogUI расширяет класс flash.display.Sprite:

```
public class LogUI extends Sprite {
  public function update (msg:String):void {
    // Отображение статусного сообщения на экране, код не показан...
  }
}
```

В языке ActionScript класс не может расширять несколько классов. Класс LogUI уже расширяет класс Sprite, поэтому он не может расширять еще и класс LogRecipient. Следовательно, экземпляры класса LogUI не могут регистрироваться в классе Logger, чтобы получать от него статусные сообщения. В данной ситуации нам крайне необходим способ, позволяющий указать, что на самом деле экземпляры класса LogUI принадлежат двум типам данных: LogUI и LogRecipient.

В игру вступают... интерфейсы!

Интерфейсы и классы с несколькими типами данных

В предыдущем разделе мы создали тип данных LogRecipient, определив класс LogRecipient. Ограничение данного подхода заключается в том, что каждый получатель сообщений от класса Logger должен быть экземпляром либо класса LogRecipient, либо одного из его подклассов. Чтобы избавиться от этого ограничения, мы можем описать тип данных LogRecipient путем создания интерфейса LogRecipient, а не путем создания одноименного класса. В этом случае экземпляры любого класса, который формально согласен предоставить реализацию для метода update( ), могут регистрироваться для получения журнальных сообщений. Посмотрим, как это работает.

Синтаксически интерфейс представляет собой просто список методов. Например, следующий код создает интерфейс LogRecipient, содержащий один-единственный метод update( ) (обратите внимание, что интерфейсы, как и классы, могут быть описаны с помощью модификаторов управления доступом public и internal).

```
public interface LogRecipient {
  function update(msg:String):void;
}
```

Как только интерфейс будет описан, любое количество классов может использовать ключевое слово implements, чтобы вступить в соглашение с этим интерфейсом, пообещав определить содержащиеся в нем методы. Как только класс даст такое обещание, его экземпляры будут считаться членами не только типа данных класса, но и типа данных интерфейса.

Например, чтобы указать, что класс LogUI согласен определить метод update( ) (описанный в интерфейсе LogRecipient), мы используем следующий код:

```
class LogUI extends Sprite implements LogRecipient {
  public function update (msg:String):void {
    // Отображение статусного сообщения на экране, код не показан...
  }
}
```

Вместо того чтобы расширять класс `LogRecipient`, `LogUI` расширяет класс `Sprite` и реализует интерфейс `LogRecipient`. Поскольку класс `LogUI` реализует интерфейс `LogRecipient`, он должен определить метод `update( )`. В противном случае компилятор сгенерирует следующую ошибку:

```
Interface method update in namespace LogRecipient not implemented by class LogUI.
```

На русском языке она будет выглядеть следующим образом: **Метод интерфейса update из пространства имен LogRecipient не реализован классом LogUI.**

Поскольку класс `LogUI` обещает реализовать методы интерфейса `LogRecipient`, экземпляры этого класса могут быть использованы везде, где требуется тип данных `LogRecipient`. Экземпляры класса `LogUI` фактически принадлежат двум типам данных: `LogUI` и `LogRecipient`. Таким образом, даже несмотря на то, что класс `LogUI` расширяет класс `Sprite`, экземпляры класса `LogUI` принадлежат типу `LogRecipient` и могут благополучно передаваться в метод `addRecipient( )` класса `Logger`.

Ошибки компилятора — это ключ ко всей системе интерфейсов. Они гарантируют, что класс сдержит свои обязательства по реализации методов интерфейса, тем самым позволив внешнему коду использовать этот класс с уверенностью в его правильном поведении. Эта уверенность особенно важна при разработке приложения, возможности которого будут расширяться другим разработчиком или использоваться третьими лицами.

Теперь, когда мы получили общее представление об интерфейсах и о том, как они используются, рассмотрим детали синтаксиса.

Синтаксис и использование интерфейсов

Как уже известно, интерфейс описывает новый тип данных, не предоставляя реализацию ни для одного из методов этого типа. Таким образом, для создания интерфейса используется следующий синтаксис:

```
interface НекоеИмя {
  function метод1 (параметр1:типданных... параметрn:типданных):
типВозвращаемогоЗначения;
  function метод2 (параметр1:типданных... параметрn:типданных):
типВозвращаемогоЗначения;
  ...
  function методn (параметр1:типданных... параметрn:типданных):
типВозвращаемогоЗначения;
}
```

Здесь *НекоеИмя* — это имя интерфейса, *метод1*... *методn* — методы данного интерфейса, *параметр1*:*типданных*... *параметрn*:*типданных* — параметры методов, а *типВозвращаемогоЗначения* — тип данных возвращаемого значения каждого из методов.

Объявления методов в интерфейсах не включают (и не должны включать) фигурные скобки. Следующее объявление метода вызовет ошибку в интерфейсе на этапе компиляции, поскольку в объявление включены фигурные скобки:

```
function метод1 (параметр:типданных):типВозвращаемогоЗначения {
}
```

Возникшая ошибка:

```
Methods defined in an interface must not have a body.
```

На русском языке ошибка будет выглядеть так: **Описываемые в интерфейсе методы не должны иметь тела.**

Объявления всех методов в интерфейсе не должны включать модификаторы управления доступом. Интерфейсы в языке ActionScript не могут содержать определения переменных; описания интерфейсов не могут быть вложенными. Тем не менее интерфейсы могут включать get- и set-методы, которые могут применяться для имитации переменных (с позиции кода, использующего эти методы). Описания интерфейсов, как и описания классов, могут размещаться либо непосредственно внутри оператора `package`, либо за его пределами, но нигде больше.

Как было сказано в предыдущем разделе, класс, который желает принять тип данных интерфейса, должен согласиться реализовать методы этого интерфейса. Чтобы сформировать данное соглашение, класс использует ключевое слово `implements`, имеющее следующий синтаксис:

```
class НекоеИмя implements НекийИнтерфейс {
}
```

В данном коде *НекоеИмя* — это имя класса, который обещает реализовать методы интерфейса *НекийИнтерфейс*, а *НекийИнтерфейс* — имя интерфейса. Говорят, что класс *НекийКласс* «реализует интерфейс *НекийИнтерфейс*». Обратите внимание, что, если в описании класса используется раздел `extends`, ключевое слово `implements` должно идти после него. Более того, если после ключевого слова `implements` вместо имени интерфейса вы укажете имя класса, компилятор сгенерирует следующую ошибку:

```
An interface can only extend other interfaces, but ИмяКласса is a class.
```

По-русски она будет выглядеть так: **Интерфейс может только расширять другие интерфейсы, а ИмяКласса является классом.**

Класс *НекоеИмя* должен реализовать все методы, описанные в интерфейсе *НекийИнтерфейс*, иначе на этапе компиляции возникнет следующая ошибка:

```
Interface method имяМетода in namespace ИмяИнтерфейса not implemented by class
ИмяКласса.
```

На русском языке ошибка означает следующее: **Метод интерфейса имяМетода в пространстве имен ИмяИнтерфейса не реализован классом ИмяКласса.**

Определения методов в классе, реализующем интерфейс, должны быть открытыми и в точности соответствовать определениям методов интерфейса, включая количество и типы параметров, а также тип возвращаемого значения. Если между интерфейсом и реализующим его классом существует различие по любому из перечисленных аспектов, компилятор сгенерирует следующую ошибку:

`Interface method` *имяМетода* `in namespace` *ИмяИнтерфейса* `is implemented with an incompatible signature in class` *ИмяКласса*.

По-русски она будет звучать так: Метод интерфейса имяМетода в пространстве имен ИмяИнтерфейса реализован с несовместимой сигнатурой в классе ИмяКласса.

Класс может реализовать несколько интерфейсов, разделив их имена запятыми, как показано в следующем коде:

```
class НекоеИмя implements НекийИнтерфейс, НекийДругойИнтерфейс {
}
```

В данном случае экземпляры класса *НекоеИмя* принадлежат всем трем указанным типам данных: *НекоеИмя*, *НекийИнтерфейс* и *НекийДругойИнтерфейс*. Если класс реализует два интерфейса, в каждом из которых описан метод с одним и тем же именем, но с отличающейся сигнатурой (то есть имя метода, список параметров и тип возвращаемого значения), то компилятор сгенерирует ошибку, указывающую на то, что один из методов реализован неправильно.

С другой стороны, если класс реализует два интерфейса, в каждом из которых описан метод с одним и тем же именем и совпадающей сигнатурой, ошибки не будет. Однако в связи с этим возникает вопрос: может ли класс предоставить функциональность, необходимую обоим интерфейсам, в одном определении метода? В большинстве случаев ответ на этот вопрос отрицателен.

Добавление новых методов в интерфейс, который уже реализован одним или несколькими классами, приведет к появлению в этих классах ошибок на этапе компиляции (поскольку в классах определение новых методов отсутствует)! Таким образом, перед внесением изменений в код вы должны тщательно подумать, какие методы будут включены в интерфейс, и быть уверенными в архитектуре своего приложения.

Если класс объявит, что он реализует некий интерфейс, но компилятор не сможет найти этот интерфейс, то появится следующая ошибка:

`Interface` *ИмяИнтерфейса* `was not found.`

На русском языке ошибка будет выглядеть следующим образом: Интерфейс ИмяИнтерфейса не найден.

Соглашения по именованию интерфейсов

Имена интерфейсов, как и имена классов, должны начинаться с прописной буквы, давая понять, что они представляют типы данных. Большинству интерфейсов имена даются исходя из дополнительной возможности, описываемой этими интерфейсами.

Предположим, что приложение содержит несколько классов, представляющих визуальные объекты. Некоторые объекты можно перемещать, другие — нет. В нашем проекте объекты, которые могут быть перемещены, должны реализовать интерфейс `Moveable`. Рассмотрим пример теоретического класса `ProductIcon`, реализующего интерфейс `Moveable`:

```
public class ProductIcon implements Moveable {
  public function getPosition ( ):Point {
```

```
  }
  public function setPosition (pos:Point):void {
  }
}
```

Интерфейс с именем `Moveable` обозначает конкретную возможность, которую он добавляет в класс. Объект может быть фрагментом изображения или блоком текста, но, если он реализует интерфейс `Moveable`, его можно перемещать. Примерами похожих имен являются `Storable`, `Killable` или `Serializable`. Некоторые разработчики перед именем интерфейса дополнительно указывают букву I, например `IMoveable`, `IKillable` или `ISerializable`.

Наследование интерфейсов

Как и в случае с классами, для наследования одного интерфейса от другого может применяться ключевое слово `extends`. Например, следующий код демонстрирует интерфейс `IntA`, который расширяет другой интерфейс — `IntB`. В данной схеме интерфейс `IntB` называется *подинтерфейсом*, а интерфейс `IntA` — *суперинтерфейсом*.

```
public interface IntA {
  function methodA ( ):void;
}
public interface IntB extends IntA {
  function methodB ( ):void;
}
```

Классы, реализующие интерфейс `IntB`, должны определять не только метод `methodB( )`, но и метод `methodA( )`. Наследование интерфейсов позволяет описывать иерархию типов, во многом напоминающую иерархию, которая образуется при использовании наследования классов, но без предоставления реализаций методов.

Интерфейсы языка ActionScript также поддерживают множественное наследование, то есть один интерфейс может расширять несколько. Например, рассмотрим следующие три описания интерфейсов:

```
public interface IntC {
  function methodC ( ):void;
}

public interface IntD {
  function methodD ( ):void;
}

public interface IntE extends IntC, IntD {
  function methodE ( ):void;
}
```

Поскольку интерфейс `IntE` расширяет оба интерфейса `IntC` и `IntD`, классы, реализующие интерфейс `IntE`, должны предоставить определения для методов `methodC( )`, `methodD( )` и `methodE( )`.

Интерфейсы-маркеры

Чтобы быть полезными, интерфейсы могут вообще не содержать никаких методов. Иногда пустые интерфейсы, называемые *интерфейсами-маркерами*, применяются для «отметки» (обозначения) класса, обладающего некоторой возможностью. Требования, предъявляемые к *отмеченным классам* (классам, реализующим интерфейс-маркер), описываются в документации по каждому конкретному интерфейсу-маркеру. Например, API среды выполнения Flash включает интерфейс-маркер `IBitmapDrawable`, обозначающий класс, который может быть отображен объектом `BitmapData`. Класс `BitmapData` будет отображать только те классы, которые реализуют интерфейс `IBitmapDrawable` (хотя на самом деле этот интерфейс не определяет никаких методов). Интерфейс `IBitmapDrawable` используется просто для того, чтобы «показать», что данный класс пригоден для работы с растровым изображением. Вот исходный код интерфейса `IBitmapDrawable`:

```
package flash.display {
  interface IBitmapDrawable {
  }
}
```

Другой пример использования составных типов

Как уже известно из предыдущего примера с протоколирующим классом, класс может не только наследоваться от другого класса, но и реализовывать интерфейс. Экземпляры подкласса одновременно принадлежат типу данных суперкласса и типу данных интерфейса. Например, экземпляры класса `LogUI` из рассмотренного примера принадлежали типам данных `Sprite` и `LogRecipient`, поскольку класс `LogUI` был унаследован от класса `Sprite` и реализовывал интерфейс `LogRecipient`. Рассмотрим эту важную архитектурную структуру на новом примере.

Для понимания материала, изложенного в этом разделе, требуется предварительное знание массивов (упорядоченных списков значений), которые еще не рассматривались в этой книге. Если вы незнакомы с массивами, то пока пропустите этот раздел и вернитесь к нему после изучения гл. 11.

Предположим, мы создаем приложение, которое сохраняет объекты на сервере с помощью серверного сценария. Класс каждого сохраняемого объекта обязан предоставить метод `serialize( )`, возвращающий строковое представление экземпляров данного класса. Строковое представление используется для воссоздания определенного объекта с нуля.

Одним из классов нашего приложения является простой класс `Rectangle` с переменными экземпляра `width`, `height`, `fillColor` и `lineColor`. Для представления объектов `Rectangle` в виде строк класс `Rectangle` реализует метод `serialize( )`, который возвращает строку следующего формата:

`"width=`*значение*`|height=`*значение*`|fillColor=`*значение*`|lineColor=`*значение*`"`

Чтобы сохранить объект Rectangle на сервере, мы вызываем метод serialize( ) над данным объектом и передаем полученную строку в наш серверный сценарий. В дальнейшем мы сможем получить эту строку и создать новый экземпляр класса Rectangle, размеры и цвет которого будут соответствовать значениям исходного экземпляра.

Для упрощения данного примера мы будем полагать, что каждый сохраняемый объект в приложении должен сохранять только имена переменных и их значения. Мы также будем полагать, что никакие значения переменных сами по себе не являются объектами, для которых требуется сериализация (то есть преобразован в строку).

Когда наступит время сохранить состояние нашего приложения, экземпляр пользовательского класса StorageManager выполнит следующие задачи.

1. Соберет объекты для сохранения.
2. Преобразует каждый объект в строку (вызвав метод serialize( )).
3. Перенесет объекты на диск.

Чтобы гарантировать тот факт, что каждый сохраняемый объект может быть сериализован, класс StorageManager отклонит любые экземпляры классов, которые не принадлежат типу данных Serializable. Вот фрагмент кода класса StorageManager, демонстрирующий метод addObject( ), который используется объектами для регистрации в списке сохраняемых объектов (обратите внимание, что в этот метод могут быть переданы только экземпляры, принадлежащие типу Serializable):

```
package {
  public class StorageManager {
    public function addObject (o:Serializable):void {
    }
  }
}
```

Тип данных Serializable описывается одноименным интерфейсом, который содержит один-единственный метод serialize( ), как показано в следующем коде:

```
package {
  public interface Serializable {
    function serialize( ):String;
  }
}
```

Для выполнения сериализации создадим класс Serializer, реализующий интерфейс Serializable. Этот класс предоставляет следующие базовые методы для сериализации любого объекта:

- setSerializationObj( ) — указывает объект для сериализации;
- setSerializationVars( ) — задает, какие переменные объекта должны быть сериализованы;
- setRecordSeparator( ) — указывает строку, используемую в качестве разделителя между переменными;
- serialize( ) — возвращает строку, представляющую объект.

Рассмотрим исходный код класса `Serializer`:

```
package {
  public class Serializer implements Serializable {
    private var serializationVars:Array;
    private var serializationObj:Serializable;
    private var recordSeparator:String;

    public function Serializer ( ) {
      setSerializationObj(this);
    }

    public function setSerializationVars (vars:Array):void {
      serializationVars = vars;
    }

    public function setSerializationObj (obj:Serializable):void {
      serializationObj = obj;
    }

    public function setRecordSeparator (rs:String):void {
      recordSeparator = rs;
    }

    public function serialize ( ):String {
      var s:String = "";
      // Обратите внимание, что счетчик цикла
      // уменьшается до нуля,
      // а его обновление (декремент i) происходит внутри
      // условного выражения цикла
      for (var i:int = serializationVars.length; --i >= 0; ) {
        s += serializationVars[i]
          + "=" + String(serializationObj[serializationVars[i]]);
        if (i > 0) {
          s += recordSeparator;
        }
      }
      return s;
    }
  }
}
```

Если какой-либо класс желает воспользоваться возможностями сериализации класса `Serializer`, то может просто расширить его. Класс, непосредственно расширивший класс `Serializer`, унаследует и интерфейс `Serializable`, и реализацию этого интерфейса классом `Serializer`.

Обратите внимание на общую структуру системы сериализации: класс `Serializer` реализует интерфейс `Serializable`, предоставляя базовую реализацию, которая может быть использована в других классах через процедуру наследования. Но при этом классы могут реализовать интерфейс `Serializable` самостоятельно, предоставив желаемое поведение для метода `serialize( )`.

Например, следующий код демонстрирует класс `Point`, определяющий переменные x и y, которые должны быть сериализованы. Этот класс расширяет класс `Serializer` и непосредственно использует его возможности.

```
package {
  public class Point extends Serializer {
    public var x:Number;
    public var y:Number;

    public function Point (x:Number, y:Number) {
      super( );

      setRecordSeparator(",");
      setSerializationVars(["x", "y"]);

      this.x = x;
      this.y = y;
    }
  }
}
```

Код, желающий сохранить экземпляр класса `Point` на диск, просто вызывает метод `serialize( )` над этим экземпляром, как показано в следующем примере:

```
var p:Point = new Point(5, 6);
trace(p.serialize( ));  // Отображает: y=6,x=5
```

Стоит отметить, что класс `Point` непосредственно не реализует интерфейс `Serializable`. Этот класс расширяет класс `Serializer`, который, в свою очередь, реализует интерфейс `Serializable`.

Класс `Point` не расширяет никакой другой класс, поэтому он может свободно расширить класс `Serializer`. Тем не менее, если некий класс желает использовать класс `Serializer`, но уже расширяет другой класс, вместо наследования необходимо применять композицию. Иными словами, вместо расширения класса `Serializer` класс непосредственно реализует интерфейс `Serializable`, сохранит объект `Serializer` в переменной экземпляра и переадресует вызовы метода `serializer( )` этому объекту. Например, рассмотрим код упомянутого ранее класса `Rectangle`. Этот класс расширяет класс `Shape`, но использует возможности класса `Serializer` через композицию (обратите особое внимание на строки, выделенные полужирным шрифтом):

```
// Суперкласс Shape
package {
  public class Shape {
    public var fillColor:uint = 0xFFFFFF;
    public var lineColor:uint = 0;

    public function Shape (fillColor:uint, lineColor:uint) {
      this.fillColor = fillColor;
      this.lineColor = lineColor;
    }
  }
}
```

```
// Класс Rectangle
package {
  // Подкласс Rectangle непосредственно реализует
  // интерфейс Serializable
  public class Rectangle extends Shape implements Serializable {
    public var width:Number = 0;
    public var height:Number = 0;
    private var serializer:Serializer;

    public function Rectangle (fillColor:uint, lineColor:uint) {
      super(fillColor, lineColor)

      // Именно здесь создается композиция
      serializer = new Serializer( );
      serializer.setRecordSeparator("|");
      serializer.setSerializationVars(["height", "width",
                                       "fillColor", "lineColor"]);
      serializer.setSerializationObj(this);
    }

    public function setSize (w:Number, h:Number):void {
      width = w;
      height = h;
    }

    public function getArea ( ):Number {
      return width * height;
    }

    public function serialize ( ):String {
      // Здесь класс Rectangle переадресует вызов метода serialize( )
      // экземпляру класса Serializer, сохраненному
      // в переменной serializer
      return serializer.serialize( );
    }
  }
}
```

Как и в случае с классом `Point`, код, желающий сохранить экземпляр класса `Rectangle`, просто вызывает метод `serialize( )` над этим объектом. Вызов метода через композицию будет переадресован экземпляру класса `Serializer`, сохраненному в классе `Rectangle`. Вот пример использования этого класса:

```
var r:Rectangle = new Rectangle(0xFF0000, 0x0000FF);
r.setSize(10, 15);
// Отображает: lineColor=255|fillColor=16711680|width=10|height=15
trace(r.serialize( ));
```

Если класс желает реализовать собственную версию метода `serialize( )` вместо того, чтобы использовать базовую версию этого метода, предоставляемую классом `Serializer`, он должен непосредственно реализовать интерфейс `Serializable`, предоставив определение и само тело метода `serialize( )`.

Разделение интерфейса типа данных `Serializable` и его реализации позволяет любому классу легко выбрать один из следующих вариантов реализации метода `serialize( )`:

- расширить класс `Serializer`;
- использовать класс `Serializer` через композицию;
- непосредственно предоставить собственную реализацию метода `serialize( )`.

Если класс не расширяет другой класс, он может расширить класс `Serializer` (этот вариант наименее трудоемкий). Если класс уже расширяет другой класс, он может использовать возможности класса `Serializer` через композицию (этот вариант наиболее гибкий). Наконец, если класс нуждается в собственной уникальной процедуре сериализации, он может непосредственно реализовать интерфейс `Serializable` (этот вариант наиболее трудоемкий, но в некоторых ситуациях обойтись без него невозможно).

Принимая во внимание гибкость описанной структуры, корпорация Sun Microsystems рекомендует, чтобы в приложениях на языке Java любой класс, у которого могут быть подклассы, являлся реализацией некоего интерфейса. По существу, подклассы могут создаваться непосредственно от данного класса или же он может использоваться классом, унаследованным от другого класса, через композицию. Рекомендация корпорации Sun также имеет смысл и для больших приложений, разрабатываемых на языке ActionScript.

На рис. 9.1 показана обобщенная структура типа данных, реализация которого может применяться как через наследование, так и через композицию.

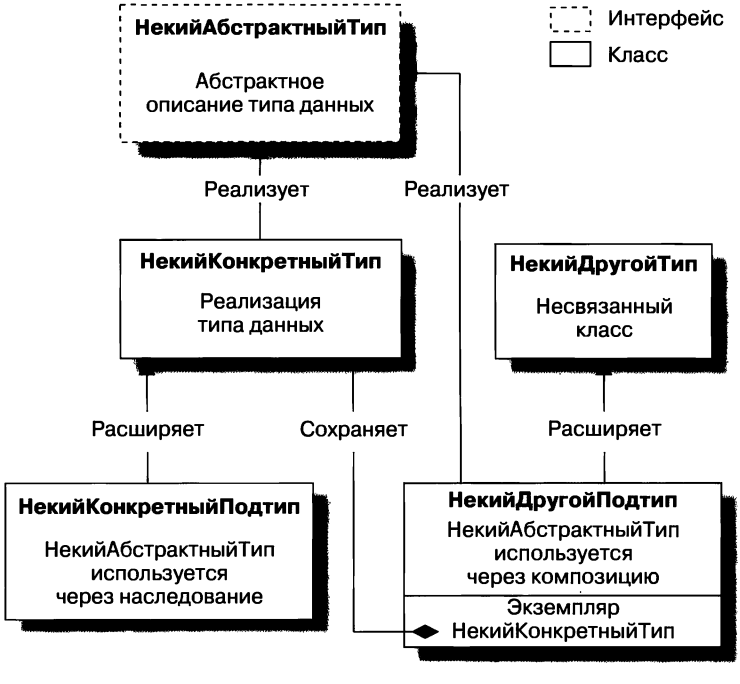

Рис. 9.1. Множественное наследование типов данных через интерфейсы

На рис. 9.2 показана структура конкретных классов `Serializable`, `Point` и `Rectangle` из предыдущего примера.

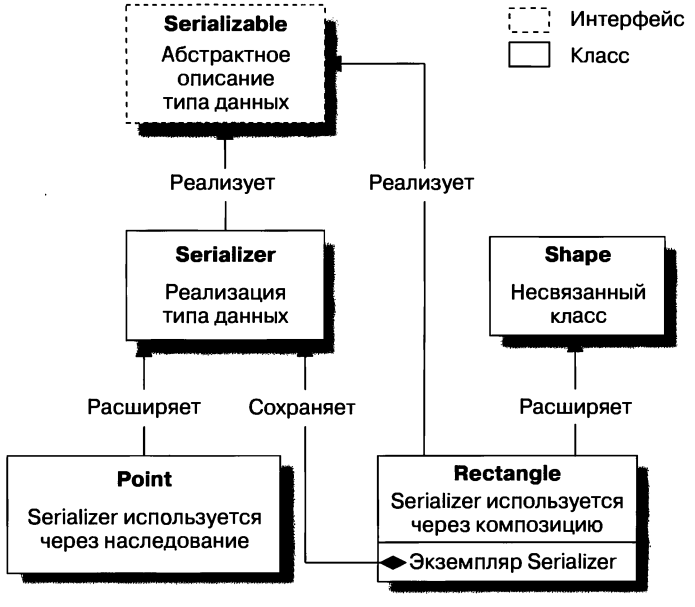

Рис. 9.2. Множественное наследование типов данных на примере интерфейса Serializable

Впереди еще много важного

Познакомившись с классами, объектами, наследованием, типами данных и интерфейсами, мы завершили изучение базовых концепций объектно-ориентированного программирования. До конца части I мы рассмотрим еще несколько других фундаментальных тем, касающихся языка ActionScript. Однако изученные концепции объектно-ориентированного программирования не останутся невостребованными, поскольку они являются основой языка ActionScript.

В следующей главе рассматриваются инструкции и операторы языка ActionScript.

Инструкции и операторы

В этой главе приводится краткое, организованное в виде справочника, описание инструкций и операторов языка ActionScript, со многими из которых вы уже встречались в данной книге. Вместо того чтобы рассматривать каждую инструкцию и оператор по отдельности, эта книга обучает использованию инструкций и операторов в контексте других тем по программированию. Таким образом, глава содержит множество перекрестных ссылок на соответствующие разделы и примеры, разбросанные по всей книге. Информацию по операторам, которые не рассматриваются в данной книге, можно найти в справочнике по языку ActionScript корпорации Adobe.

Инструкции

Инструкции представляют собой разновидность директив, или базовых команд, программы, состоящую из ключевого слова (имени команды, зарезервированного для использования в языке ActionScript) и, обычно, вспомогательного выражения.

В табл. 10.1 перечислены инструкции языка ActionScript, их синтаксис и назначение.

Таблица 10.1. Инструкции языка ActionScript

Инструкция	Использование	Описание
break	break	Завершает цикл или инструкцию switch. Подробную информацию можно найти в гл. 2
case	case выражение: вложенные_инструкции	Обозначает инструкцию, выполняемую по условию в инструкции switch. Подробную информацию можно найти в гл. 2
continue	continue;	Пропускает оставшиеся инструкции в текущем цикле и начинает новую итерацию с начала цикла. За дополнительной информацией обратитесь к документации от корпорации Adobe
default	default: вложенные_инструкции	Обозначает инструкцию (-и), выполняемую (-ые) инструкцией switch, когда результат условного выражения не соответствует ни одному из значений выражений case. Дополнительную информацию можно найти в гл. 2
do-while	do { вложенные_инструкции } while (выражение)	Разновидность цикла while, гарантирующая, что тело цикла будет выполнено по крайней мере один раз. Дополнительную информацию можно найти в гл. 2

for	for (инициализация;
	условноеВыражение;
	корректирование) {
	инструкции
	}
		Многократно выполняет блок инструкций (цикл for). Синоним цикла while, однако выражения инициализации и корректирования цикла размещаются вместе с условным выражением в верхней части цикла. Дополнительную информацию можно найти в гл. 2

Продолжение

Таблица 10.1 (продолжение)

Инструкция	Использование	Описание

for-in	for (переменная in объект) {
	инструкции
	}
		Перечисляет имена динамических переменных экземпляра или элементы массива. Дополнительную информацию можно найти в гл. 15

for-each-in	for each (переменная-ИлиЗначениеЭлемента in объект) {
	инструкции
	}
		Перечисляет значения динамических переменных экземпляра или элементы массива. Дополнительную информацию можно найти в гл. 15

if-else if-else	if (выражение) {
	вложенные_инструкции
	} else if (выражение) {
	вложенные_инструкции
	} else {
	вложенные_инструкции
	}
		Выполняет одну или несколько инструкций в зависимости от условия или ряда условий. Дополнительную информацию можно найти в гл. 2

label	label: инструкция label: инструкции	Связывает инструкцию с идентификатором. Применяется вместе с инструкциями break или continue. За дополнительной информацией обратитесь к документации от корпорации Adobe
return	return; return выражение;	Выход из функции, при этом может возвращаться значение. Дополнительную информацию можно найти в гл. 5
super	super(аргумент1, аргумент2... аргументn) super.метод(аргумент1, аргумент2... аргументn)	Вызывает метод конструктора суперкласса или перекрытый метод экземпляра. Дополнительную информацию можно найти в гл. 6

switch	switch (выражение) {
	вложенные_инструкции
	}
		Выполняет указанный код в зависимости от условия или ряда условий (является альтернативой инструкции if-else if-else). Дополнительную информацию можно найти в гл. 2

throw	throw выражение	Генерирует исключение (ошибку) на этапе выполнения. Дополнительную информацию можно найти в гл. 13

try/catch/ finally	try {
	// код, который может
	// сгенерировать
	// исключение
	} catch (error:ТипОшибки1) {
	// Код обработчика
	// ошибки типа
	// ТипОшибки1
	} catch (error:ТипОшибкиN) {
	// Код обработчика
	// ошибки типа
	// ТипОшибкиN
	} finally {
	// Код, который
	// выполняется всегда
	}
		Окружает блок кода для реакции на возможные исключения, возникающие на этапе выполнения. Дополнительную информацию можно найти в гл. 13

Инструкция	Использование	Описание

while	while (выражение) {
	вложенные_инструкции
	}
		Многократно выполняет блок инструкций (цикл while). Дополнительную информацию можно найти в гл. 2

with	with (объект) {
	вложенные_инструкции
	}
		Выполняет блок инструкций в контексте данного объекта. Дополнительную информацию можно найти в гл. 16

Операторы

Оператор — это символ или ключевое слово, предназначенные для управления, объединения или преобразования данных. Например, следующий код использует оператор умножения (*), чтобы умножить число 5 на число 6:

```
5 * 6;
```

Хотя каждый оператор выполняет свою специализированную задачу, все операторы обладают рядом общих характеристик. Перед тем как рассмотреть каждый оператор по отдельности, познакомимся с их общим поведением.

Операторы выполняют действия над указанными значениями данных (*операндами*). Например, в операции `5 * 6` числа 5 и 6 являются *операндами* оператора умножения (*).

Отдельные операции могут быть объединены в одно сложное выражение. Например:

```
((width * height) - (Math.PI * radius * radius)) / 2
```

В случае очень больших выражений, возможно, более удобным окажется использование переменных для хранения промежуточных результатов, при этом код станет более понятным. Не забывайте давать переменным описательные имена. Например, при выполнении следующего кода получается такой же результат, как и в предыдущем выражении, однако данный код гораздо проще для восприятия:

```
var radius:int = 10;
var height:int = 25;
var circleArea:Number = (Math.PI * radius * radius);
var cylinderVolume:Number = circleArea * height;
```

Количество операндов

Операторы иногда классифицируют по количеству принимаемых операндов (то есть требуемых для *выполнения операции*). Некоторые операторы языка ActionScript принимают один операнд, некоторые — два, а один оператор принимает даже три операнда:

```
-x                                          // Один операнд
x * y                                       // Два операнда
(x == y) ? "true result" : "false result"   // Три операнда
```

Операторы, принимающие один операнд, называются *унарными*; операторы, принимающие два операнда, называются *бинарными*; операторы, принимающие три операнда, называются *тернарными*. Для наших целей мы будем рассматривать операторы с точки зрения выполняемых ими действий, а не с точки зрения количества принимаемых операндов.

Приоритет операторов

Приоритет операторов определяет, какая операция будет выполнена первой в выражении, состоящем из нескольких операторов. Например, если в одном выражении встречаются операторы умножения и сложения, первой будет выполнена операция умножения:

```
4 + 5 * 6  // Возвращает 34, поскольку 4 + 30 = 34
```

Выражение `4 + 5 * 6` вычисляется, как «4 плюс произведение 5 и 6», поскольку оператор * имеет более высокий приоритет, чем оператор +.

Подобным образом, если в одном выражении встречаются оператор «меньше чем» (<) и оператор конкатенации (+), операция конкатенации будет выполнена первой. Предположим, мы хотим сравнить две строки и отобразить результат сравнения при отладке программы. Не зная приоритетов операторов < и +, мы можем по ошибке использовать следующий код:

```
trace("result: " + "a" < "b");
```

Из-за приоритетов операторов < и + код выдаст значение `false`, хотя мы ожидали увидеть несколько иной результат:

```
result: true
```

Чтобы определить результат выражения `"result: " + "a" < "b"`, среда Flash сначала выполнит операцию конкатенации (поскольку оператор + обладает более высоким приоритетом, чем <). Результатом конкатенации строки `"result: "` со строкой `"a"` является новая строка `"result: a"`. После этого среда выполнения Flash сравнивает полученную строку со строкой `"b"` и получает значение `false`, поскольку первый символ строки `"result: a"` находится дальше по алфавиту, чем символ `"b"`.

Если вы сомневаетесь в приоритетах используемых операторов или хотите указать другую последовательность выполнения операций, используйте круглые скобки, которые обладают самым высоким приоритетом:

```
"result: " + ("a" < "b")  // Возвращает: "result: true"
(4 + 5) * 6               // Возвращает 54, поскольку 9 * 6 = 54
```

Хотя использовать круглые скобки совсем не обязательно, они помогают сделать сложное выражение более читабельным. Выражение:

```
x > y || y == z  // x больше y или y равняется z
```

может оказаться сложным для восприятия без знания таблицы приоритетов. Оно становится гораздо более понятным, когда расставлены круглые скобки:

```
(x > y) || (y == z)  // Гораздо лучше!
```

Ассоциативность операторов

Как уже известно, приоритет операторов определяет очередность их выполнения в выражении: операторы, обладающие более высоким приоритетом, выполняются раньше операторов с более низким приоритетом. Однако что произойдет, если в одном выражении встретятся несколько операторов с одинаковым уровнем приоритета? В данном случае применяются правила *ассоциативности операторов*, определяющих направление операции. Операторы могут обладать либо левой (выполняются слева направо), либо правой ассоциативностью (выполняются справа налево). Например, рассмотрим следующее выражение:

```
b * c / d
```

Операторы * и / обладают левой ассоциативностью, поэтому операция умножения слева (`b * c`) выполняется первой. Предыдущий пример эквивалентен следующему выражению:

```
(b * c) / d
```

Здесь оператор = (оператор присваивания) обладает правой ассоциативностью, поэтому выражение

```
a = b = c = d
```

читается, как «присвоить значение `d` переменной `c`, затем присвоить значение `c` переменной `b`, после чего присвоить значение `b` переменной `a`», как показано в следующем примере:

```
a = (b = (c = d))
```

Унарные операторы обладают правой ассоциативностью; бинарные операторы обладают левой ассоциативностью, за исключением операторов присваивания, обладающих правой ассоциативностью. Условный оператор (`?:`) также обладает правой ассоциативностью. Ассоциативность операторов достаточно понятна на интуитивном уровне, но если сложное выражение вернуло неожидаемое значение, воспользуйтесь дополнительными круглыми скобками, чтобы указать желаемый порядок выполнения операций. Более подробную информацию по ассоциативности операторов в языке ActionScript можно найти в документации от корпорации Adobe.

Типы данных и операторы

Операнды большинства операторов являются типизированными. В строгом режиме, если значение, используемое в качестве операнда, не соответствует типу данных данного операнда, компилятор сгенерирует ошибку на этапе компиляции и прекратит компиляцию кода. В стандартном режиме код будет скомпилирован и на этапе выполнения программы. Если тип операнда является примитивным, среда выполнения Flash преобразует значение к типу данных этого операнда (в соответствии с правилами, описанными в разд. «Преобразование в примитивные типы» гл. 8). Если тип операнда не является примитивным, то среда Flash сгенерирует ошибку на этапе выполнения.

Например, при компиляции программы в строгом режиме следующий код вызовет ошибку несоответствия типов, поскольку типом данных операндов оператора

деления (/) является Number, а значение "50" не принадлежит типу данных Number:

```
"50" / 10
```

При компиляции программы в стандартном режиме предыдущий код не вызовет ошибку на этапе компиляции. Вместо этого на этапе выполнения программы среда Flash преобразует значение "50" типа String к типу данных Number, получив в результате значение 50, и указанное выражение вернет значение 5.

Чтобы при компиляции предыдущего кода в строгом режиме не возникала ошибка, мы должны привести значение типа String к требуемому типу данных, как показано в следующем примере:

```
Number("50") / 10
```

Операнды некоторых операторов являются нетипизированными, что позволяет определять результат операции на этапе выполнения программы в зависимости от типов данных указанных значений. Оператор +, например, выполняет операцию сложения, если указаны два числовых операнда. Однако если одним из операндов является строка, то будет выполнена операция конкатенации.

Типы данных операндов всех операторов описаны в справочнике по языку ActionScript корпорации Adobe.

Обзор операторов

В табл. 10.2 перечислены операторы языка ActionScript с указанием значения приоритета, кратким описанием и типовым примером использования. Операторы с самым высоким приоритетом (находятся вверху таблицы) выполняются первыми. Операторы с одинаковым приоритетом выполняются в том порядке, в котором они следуют в выражении, обычно слева направо, кроме тех случаев, когда операторы обладают правой ассоциативностью (то есть выполняются справа налево).

Стоит отметить, что книга не содержит исчерпывающей информации об операторах языка ActionScript, за исключением операторов, описанных в спецификации E4X. Дополнительную информацию о каждом конкретном операторе можно найти в справочнике по языку ActionScript корпорации Adobe. Побитовые операторы описаны в статье «Using Bitwise Operators in ActionScript», находящейся по адресу http://www.moock.org/asdg/technotes/bitwise.

Таблица 10.2. Операторы языка ActionScript

Оператор	Приоритет	Описание	Пример
.	15	Многоцелевое использование. 1. Обращается к переменной или методу. 2. Отделяет имена пакетов от имен классов и других пакетов. 3. Обращается к дочерним элементам объекта XML или XMLList (спецификация E4X)	1. Обращение к переменной: product.price 2. Ссылка на класс: flash.display.Sprite 3. Обращение к дочернему элементу объекта XML: novel.TITLE

Оператор	Приоритет	Описание	Пример
[ ]	15	Многоцелевое использование. 1. Выполняет инициализацию массива. 2. Обращается к элементу массива. 3. Обращается к переменной или методу с использованием любого выражения, возвращающего строку. 4. Обращается к дочерним элементам или атрибутам объекта XML или XMLList (спецификация E4X)	1. Инициализация массива: ["apple", "orange", "pear"] 2. Обращение к четвертому элементу массива: list[3] 3. Обращение к переменной: product["price"] 4. Обращение к дочернему элементу объекта XML: novel["TITLE"]
( )	15	Многоцелевое использование. 1. Задает особый порядок выполнения операций (приоритет). 2. Вызывает функцию или метод. 3. Содержит фильтрующий предикат спецификации E4X	1. Выполнение операции сложения до операции умножения: (5 + 4) * 2 2. Вызов функции: trace( ) 3. Фильтрация объекта XMLList: staff.*.(SALARY <= 35000)
@	15	Обращается к атрибутам XML-элемента	Получение всех атрибутов элемента novel: novel.@*
::	15	Отделяет уточняющее пространство имен от имени	Уточнение имени orange в пространстве имен fruit: fruit::orange
..	15	Обращается к потомкам XML-элемента	Получение всех элементов-потомков элемента loan с именем DIRECTOR: loan..DIRECTOR
{x:y}	15	Создает новый объект и инициализирует его динамические переменные	Создание объекта с динамическими переменными width и height: {width:30, height:5}
new	15	Создает экземпляр класса	Создание экземпляра класса TextField: new TextField( )
<тег><тег/>	15	Описывает XML-элемент	Создание XML-элемента с именем BOOK: <BOOK>Essential ActionScript 3.0</BOOK>
x++	14	Прибавляет единицу к значению переменной x и возвращает ее прежнее значение (постфиксный инкремент)	Увеличение i на 1 и возврат прежнего значения i: i++
x—	14	Вычитает единицу из значения переменной x и возвращает ее прежнее значение (постфиксный декремент)	Уменьшение i на 1 и возврат прежнего значения i: i—
++x	14	Прибавляет единицу к значению переменной x и возвращает ее новое значение (префиксный инкремент)	Увеличение i на 1 и возврат результата: ++i

Продолжение

Таблица 10.2 (продолжение)

Оператор	Приоритет	Описание	Пример
—x	14	Вычитает единицу из значения переменной x и возвращает ее новое значение (префиксный декремент)	Уменьшение i на 1 и возврат результата: —i
–	14	Изменяет знак операнда (положительное значение становится отрицательным, а отрицательное — положительным)	var a:int = 10; Присвоение –10 переменной b: var b:int = –a;
~	14	Выполняет операцию побитового НЕ	Сброс бита 2 значения переменной options: options &= ~4;

!	14	Возвращает логическое значение, противоположное его единственному операнду	Если значением переменной under18 не является true, то выполняется тело условного оператора:
			if (!under18) {
			trace("You can apply for a credit card")
			}

delete	14	Многоцелевое использование.
		1. Удаляет значение элемента массива.
		2. Удаляет динамическую переменную экземпляра объекта.
		3. Удаляет XML-элемент или атрибут
			1. Создание массива:
			var genders:Array = ["male","female"]
			2. Удаление значения первого элемента:
			delete genders[0];
			3. Создание объекта:
			var o:Object = new Object( );
			o.a = 10;
			4. Удаление динамической переменной экземпляра a:
			delete o.a;
			5. Удаление элемента <TITLE> из объекта XML, на который ссылается переменная novel:
			delete novel.TITLE;

typeof	14	Возвращает простое строковое описание различных типов объектов. Используется только для обеспечения обратной совместимости с языками ActionScript 1.0 и ActionScript 2.0	Получение строкового описания типа значения 35: typeof 35

void	14	Возвращает значение undefined	var o:Object = new Object( );
			o.a = 10;
			Сравнение значения undefined со значением o.a:
			if (o.a == void) {
			trace("o.a does not exist, or has no value");
			}

*	13	Перемножает два числа	Умножение 4 на 6: 4 * 6
/	13	Делит левый операнд на правый операнд	Деление 30 на 5: 30 / 5

Оператор	Приоритет	Описание	Пример
%	13	Возвращает остаток (то есть модуль) от деления левого операнда на правый операнд	Остаток от деления 14 на 4: 14 % 4
+	12	Многоцелевое использование. 1. Складывает два числа. 2. Объединяет (конкатенирует) две строки. 3. Объединяет (конкатенирует) два объекта XML или XMLList	1. Результат 25 плюс 10: 25 + 10 2. Объединение строк "He" и "llo" в строку "Hello": "He" + "llo" 3. Объединение двух объектов XML: <JOB>Programmer</JOB> + <AGE>52</AGE>
–	12	Вычитает правый операнд из левого операнда	Вычитание 2 из 12: 12 – 2
<<	11	Выполняет побитовый сдвиг влево	Сдвигает значение 9 на четыре бита влево: 9 << 4
>>	11	Выполняет побитовый знаковый сдвиг вправо	Сдвигает значение 8 на один бит вправо: 8 >> 1
>>>	11	Выполняет побитовый беззнаковый сдвиг вправо	Сдвигает значение 8 на один бит вправо, заполняя освободившиеся биты нулями: 8 >>> 1
<	10	Проверяет справедливость условия, что левый операнд меньше правого. В зависимости от значений операндов возвращает true или false	1. Проверяет справедливость условия, что 5 меньше 6: 5 < 6 2. Проверяет, обладает ли символ "a" более низкой кодовой позицией, чем символ "z": "a" < "z"
<=	10	Проверяет справедливость условия, что левый операнд меньше либо равен правому операнду. В зависимости от значений операндов возвращает true или false	1. Проверяет справедливость условия, что 10 меньше либо равно 5: 10 <= 5 2. Проверяет, обладает ли символ "C" более низкой или такой же кодовой позицией, что и символ "D": "C" <= "D"
>	10	Проверяет справедливость условия, что левый операнд больше правого. В зависимости от значений операндов возвращает true или false	1. Проверяет справедливость условия, что 5 больше 6: 5 > 6 2. Проверяет, обладает ли символ "a" более высокой кодовой позицией, чем символ "z": "a" > "z"
>=	10	Проверяет справедливость условия, что левый операнд больше либо равен правому операнду. В зависимости от значений операндов возвращает true или false	1. Проверяет справедливость условия, что 10 больше либо равно 5: 10 >= 5 2. Проверяет, обладает ли символ "C" более высокой или такой же кодовой позицией, что и символ "D": "C" >= "D"

Продолжение ⇩

Таблица 10.2 (продолжение)

Оператор	Приоритет	Описание	Пример
as		Проверяет, принадлежит ли левый операнд типу данных, указанному правым операндом. Если это так, возвращается объект; в противном случае возвращается значение null	Var d:Date = new Date( ) Проверяет, принадлежит ли значение переменной d типу данных Date: d as Date
is		Проверяет, принадлежит ли левый операнд типу данных, указанному правым операндом. Если это так, возвращается true; в противном случае возвращается false	var a:Array = new Array( ) Проверяет, принадлежит ли значение переменной a типу данных Array: a is Array
in		Проверяет, обладает ли объект указанной открытой переменной или методом экземпляра	Var d:Date = new Date( ) Проверяет, обладает ли значение переменной d открытой переменной или открытым методом с именем getMonth: "getMonth" in d
instanceof	10	Проверяет, включает ли цепочка прототипов левого операнда правый операнд. В зависимости от значений операндов возвращает true или false	var s:Sprite = new Sprite( ) Проверяет, включает ли цепочка прототипов значения переменной s объект DisplayObject: s instanceof DisplayObject
==	9	Проверяет, являются ли два выражения равными (условие равенства). В зависимости от значений операндов возвращает true или false	Проверяет, равняется ли выражение "hi" выражению "hello": "hi" == "hello"
!=	9	Проверяет, являются ли два выражения неравными (условие неравенства). В зависимости от значений операндов возвращает true или false	Проверяет, не равно ли выражение 3 выражению 3: 3 != 3
===	9	Проверяет, являются ли два выражения равными, не выполняя преобразование типов данных операндов к примитивным типам (условие строгого равенства). В зависимости от значений операндов возвращает true или false	Проверяет, равняется ли выражение "3" выражению 3. Этот код компилируется только в стандартном режиме: "3" === 3
!==	9	Проверяет, являются ли два выражения неравными, не выполняя преобразование типов данных операндов к примитивным типам (условие строгого неравенства). В зависимости от значений операндов возвращает true или false	Проверяет, не равно ли выражение "3" выражению 3. Этот код компилируется только в стандартном режиме: "3" !== 3
&	8	Выполняет операцию побитового И	Объединяет биты значений 15 и 4 с помощью операции побитового И: 15 & 4

Оператор	Приоритет	Описание	Пример
^	7	Выполняет операцию побитового исключающего ИЛИ	Объединяет биты значений 15 и 4 с помощью операции побитового исключающего ИЛИ: 15 ^ 4
|	6	Выполняет операцию побитового ИЛИ	Объединяет биты значений 15 и 4 с помощью операции побитового ИЛИ: 15 | 4

&&	5	Сравнивает результаты двух выражений с помощью операции логического И. Если значение левого операнда равно false или преобразуется в false, оператор возвращает левый операнд; в противном случае оператор возвращает правый операнд	var validUser:Boolean = true;
			var validPassword:Boolean = false;
			Проверяет, имеют ли обе переменные — validUser и validPassword — значение true:
			if (validUser && validPassword) {
			// Выполнить вход...
			}

||	4	Сравнивает результаты двух выражений с помощью операции логического ИЛИ. Если значение левого операнда равно true или преобразуется в true, оператор возвращает левый операнд; в противном случае оператор возвращает правый операнд	var promotionalDay:Boolean = false;
			var registeredUser:Boolean = false;
			Проверяет, чтобы значение любой из переменных promotionalDay или registeredUser равнялось true
			if (promotionalDay || registeredUser) {
			// Отобразить дополнительное содержимое...
			}

?:	3	Выполняет простое условие. Если значение первого операнда равно true или преобразуется в true, вычисляется и возвращается значение второго операнда. В противном случае вычисляется и возвращается значение третьего операнда	Вызывает один из двух методов в зависимости от того, хранит ли переменная soundMuted значение true: soundMuted ? displayVisualAlarm( ) : playAudioAlarm( )

=	2	Присваивает значение переменной или элементу массива	1. Присваивает 36 переменной age:
			var age:int = 36;
			2. Присваивает новый массив переменной seasons:
			var seasons:Array = new Array( );
			3. Присваивает значение "winter" первому элементу массива seasons:
			seasons[0] = "winter";

+=	2	Прибавляет (или конкатенирует) и присваивает результат	1. Прибавляет 10 к значению переменной n:
			n += 10; // то же самое, что
			и n = n + 10;
			2. Добавляет восклицательный знак в конец строки msg:
			msg += "!"
			3. Добавляет тег <AUTHOR> после первого тега <AUTHOR> потомка объекта novel:
			novel.AUTHOR[0] += <AUTHOR>Dave Luxton</AUTHOR>;

Продолжение

Таблица 10.2 (продолжение)

Оператор	Приоритет	Описание	Пример
-=	2	Вычитает и присваивает результат	Вычитает 10 из значения переменной n: n -= 10; // то же, что и n = n - 10;
*=	2	Умножает и присваивает результат	Умножает значение переменной n на 10: n *= 10; // то же, что и n = n * 10;
/=	2	Делит и присваивает результат	Делит значение переменной n на 10: n /= 10; // то же, что и n = n / 10;
%=	2	Выполняет операцию деления по модулю и присваивает результат	Присваивает переменной n результат операции n % 4: n %= 4; // то же, что и n = n % 4;
<<=	2	Сдвигает биты влево и присваивает результат	Сдвигает биты значения переменной n влево на две позиции: n <<= 2; // то же, что и n = n << 2;
>>=	2	Сдвигает биты вправо и присваивает результат	Сдвигает биты значения переменной n вправо на две позиции: n >>= 2; // то же, что и n = n >> 2;
>>>=	2	Сдвигает биты вправо (беззнаковый сдвиг) и присваивает результат	Сдвигает биты значения переменной n вправо на две позиции, заполняя освободившиеся биты нулями: n >>>= 2; // то же, что и n = n >>> 2;
&=	2	Выполняет операцию побитового И и присваивает результат	Объединяет биты значения переменной n со значением 4, используя операцию побитового И: n &= 4 // то же, что и n = n & 4;
^=	2	Выполняет операцию побитового исключающего ИЛИ и присваивает результат	Объединяет биты значения переменной n со значением 4, используя операцию побитового исключающего ИЛИ: n ^= 4 // то же, что и n = n ^ 4;
|=	2	Выполняет операцию побитового ИЛИ и присваивает результат	Объединяет биты значения переменной n со значением 4, используя операцию побитового ИЛИ: n |= 4 // то же, что и n = n | 4;

,	1	Вычисляет сначала значение левого операнда, а затем правого	Инициализация и увеличение двух итераторов цикла:
			for (var i:int = 0, j:int = 10; i < 5; i++, j++) {
			// i принимает значения от 0 до 4
			// j принимает значения от 10 до 14
			}

Далее: управление списками данных

В этой главе были рассмотрены некоторые основные встроенные инструменты программирования языка ActionScript. В следующей главе мы познакомимся с еще одним важным инструментом языка ActionScript: массивами. Массивы используются для управления списками данных.

Массивы

Массивы хранят упорядоченные списки данных и управляют ими, являясь, таким образом, основным инструментом в последовательном, итерационном программировании. Мы используем массивы для решения различных задач, начиная с хранения данных, введенных пользователем, и заканчивая генерацией раскрывающихся меню или описанием траектории движения вражеского космического корабля в игре. Фактически массив — это просто список элементов, похожий на список продуктов или на записи в вашей книге расходов. Только элементами в данном случае являются значения языка ActionScript.

Что такое массив?

Массив — это структура данных, которая объединяет несколько отдельных значений данных в упорядоченный список. Рассмотрим простой пример, который демонстрирует две отдельные строки и массив, содержащий две строки:

```
"cherries"              // Отдельная строка
"peaches"               // Еще одна строка
["oranges", "apples"]   // Массив, содержащий две строки
```

Массив может содержать любое количество элементов, причем различных типов. Массив даже может содержать другие массивы. Рассмотрим простой пример, демонстрирующий массив, который одновременно содержит и строки, и числа. Используя массив, можно представить список покупок, включающий названия покупаемых товаров и их необходимое количество:

```
["oranges", 6, "apples", 4, "bananas", 3];
```

Хотя массив может хранить большое количество значений, важно понимать, что сам по себе он является отдельным значением данных. Массивы представляются экземплярами класса `Array`. По существу, массив может быть присвоен переменной или использован как часть сложного выражения:

```
// Присвоить массив переменной
var product:Array = ["ladies downhill skis", 475];
// Передать этот массив в функцию
display(product);
```

Анатомия массива

Каждый отдельный объект, хранящийся в массиве, называется *элементом* массива, при этом каждый элемент обладает уникальной числовой позицией *(индексом)*, которая может использоваться для обращения к конкретному элементу.

Элементы массива

Каждому элементу массива, как и переменной, можно присвоить любое значение. Таким образом, весь массив похож на совокупность последовательно именованных переменных, но вместо уникального имени каждый элемент обладает номером (номером первого элемента является 0, а не 1). Для работы с соответствующими значениями мы обращаемся к элементам массива по их номеру.

Индексация элементов массива

Позиция элемента в массиве называется его индексом. Индекс элемента используется для присваивания или получения значения этого элемента, а также для выполнения различных действий над этим элементом. В некоторых методах обработки массивов индексы элементов используются для указания диапазона обрабатываемых элементов.

Кроме того, существует возможность добавлять и удалять элементы в начале, конце и даже в середине массива. Массив может иметь промежутки (то есть некоторые элементы могут быть незаполненными). Элементы могут размещаться в позициях 0 и 4, при этом совсем не обязательно, чтобы элементы присутствовали в позициях 1, 2 и 3. Массивы с промежутками называются *разреженными массивами*.

Размер массива

В любой момент времени своего существования каждый массив содержит определенное количество элементов (как незаполненных, так и заполненных). Количество элементов в массиве называется *длиной* массива. Это понятие будет рассмотрено чуть далее.

Создание массивов

Для создания нового массива применяется литерал массива или оператор `new` (то есть `new Array( )`).

Массивы в других языках программирования. Практически каждый язык программирования высокого уровня поддерживает массивы или похожие на них конструкции. При этом в способах реализации массивов в разных языках есть различия. Например, многие языки не позволяют хранить в массиве данные разных типов. Во многих языках массив может содержать либо числа, либо строки, но хранить в одном массиве значения обоих типов не допускается. Интересно, что в языке C отсутствует примитивный тип данных `string`. Вместо этого поддерживается односимвольный тип данных `char`; строки считаются составным типом данных и реализуются в виде массива, состоящего из элементов типа `char`.

В языке ActionScript размер массива изменяется автоматически при добавлении или удалении элементов. Во многих языках размер массива должен быть указан при первом *объявлении*, или *задании размерности*, массива (то есть в тот момент, когда *выделяется* память для хранения данных массива).

Языки различаются и по тому, что происходит при попытке обращения к элементу, находящемуся за границами (пределами) массива. Если программа попытается присвоить значение элементу, находящемуся за пределами существующих границ массива, язык ActionScript добавит недостающие элементы. Если программа попытается обратиться к элементу, индекс которого лежит за пределами границ массива, то язык ActionScript вернет значение `undefined`, тогда как язык C, например, не обратит никакого внимания на допустимость указанного номера элемента. Язык программирования C позволяет программе получать и присваивать значения элементам, находящимся за пределами границ массива, что обычно приводит к получению бессмысленных данных, не являющихся частью массива, или к перезаписыванию других данных в памяти.

Создание массивов с помощью литералов

Литерал массива состоит из квадратных скобок, обозначающих начало и конец массива, и элементов, которые перечисляются через запятую внутри квадратных скобок. Вот его обобщенный синтаксис:

`[`*выражение1*`, `*выражение2*`, `*выражение3*`]`

Сначала вычисляются результаты указанных выражений, а затем полученные результаты присваиваются элементам определяемого массива. В литерале массива могут использоваться любые допустимые выражения, включая вызовы функций, переменные, литералы и даже другие массивы (массив, содержащийся в другом массиве, называется *вложенным* или *двумерным*).

Рассмотрим несколько примеров:

```
// Простые числовые элементы
[4, 5, 63];

// Простые строковые элементы
["apple", "orange", "pear"]

// Числовые выражения с операцией
[1, 4, 6 + 10]

// В качестве элементов выступают значения переменных и строки
[firstName, lastName, "tall", "skinny"]

// Вложенный литерал массива
["month end days", [31, 30, 28]]
```

Создание массивов с помощью оператора new

Для создания массива с помощью оператора `new` используется следующий обобщенный код:

`new Array(`*аргументы*`)`

Результат выполнения этого кода зависит от количества и типа аргументов, передаваемых в конструктор класса `Array`. Если в конструктор передается несколько

аргументов или один нечисловой, то каждый аргумент становится значением отдельного элемента в новом массиве. Например, следующий код создает массив с тремя элементами:

```
new Array("sun", "moon", "earth")
```

Если в конструктор класса `Array` передается один числовой аргумент, то будет создан массив с указанным количеством незаполненных элементов, значения которым могут быть присвоены позднее (создание подобного массива с помощью литерала оказалось бы достаточно утомительным занятием). Например, следующий код создает массив, состоящий из 14 незаполненных элементов:

```
new Array(14)
```

Аргументами, передаваемыми в конструктор класса `Array`, могут быть любые допустимые выражения, включая составные выражения. Например, следующий код создает массив, первым элементом которого является число 11, а вторым элементом — число 50:

```
var x:int = 10;
var y:int = 5;
var numbers:Array = new Array(x + 1, x * y);
```

Для прямого сравнения следующий код создает массивы из предыдущего раздела, однако вместо литералов массива используется оператор `new`:

```
new Array(4, 5, 63)
new Array("apple", "orange", "pear")
new Array(1, 4, 6 + 10)
new Array(firstName, lastName, "tall", "skinny")
new Array("month end days", new Array(31, 30, 28))
```

Обращение к элементам массива

После создания массива мы, разумеется, захотим получать или изменять значения его элементов. Для этих целей применяется *оператор доступа к массиву* — `[ ]`.

Получение значения элемента

Для обращения к отдельному элементу массиву используется переменная, ссылающаяся на этот массив, а за ней в квадратных скобках указывается индекс элемента, как показано в следующем коде:

массив`[`*номерЭлемента*`]`

В предыдущем коде *массив* обозначает ссылку на массив (обычно это переменная, значением которой является массив), а *номерЭлемента* — это целое число, определяющее индекс элемента. Номером первого элемента является 0, а номер последнего элемента на единицу меньше длины массива. Если указанный номер элемента превышает последний допустимый номер элемента, среда выполнения Flash вернет значение `undefined` (поскольку указанный индекс находится за пределами границ массива).

Попробуем получить несколько значений элементов. Следующий код создает массив с помощью литерала массива и присваивает его переменной `trees`:

```
var trees:Array = ["birch", "maple", "oak", "cedar"];
```

Следующий код присваивает переменной `firstTree` значение первого элемента массива `trees` (`"birch"`):

```
var firstTree:String = trees[0];
```

Следующий код присваивает значение третьего элемента (`"oak"`) переменной `favoriteTree` (не забывайте, что индексы начинаются с 0, поэтому элемент с индексом 2 является третьим элементом массива!):

```
var favoriteTree:String = trees[2];
```

Теперь начинается самое интересное. Поскольку индекс элемента можно задавать с помощью любого выражения, возвращающего число, для указания индекса элемента мы можем запросто использовать переменные или сложные выражения вместо обычных чисел. Например, следующий код присваивает значение четвертого элемента (`"cedar"`) переменной `lastTree`:

```
var i = 3;
var lastTree:String = trees[i];
```

В качестве индексов массива мы можем использовать даже выражения вызова, возвращающие числовые значения. Например, следующий код присваивает переменной `randomTree` случайно выбранный элемент массива `trees`, индексом которого является случайное число в диапазоне от 0 до 3:

```
var randomTree:String = trees[Math.floor(Math.random( ) * 4)];
```

Прекрасно. Вы можете использовать подобный подход для выбора произвольного вопроса из массива, в котором хранятся тривиальные вопросы, или для выбора случайной карты из массива, представляющего колоду игральных карт.

Обратите внимание, что обращение к элементу массива очень похоже на обращение к значению переменной. Элементы массива могут являться частью любых составных выражений, как показано в следующем примере:

```
var ages:Array = [12, 4, 90];
var totalAge:Number = ages[0] + ages[1] + ages[2];   // Сумма значений
                                                     // элементов массива
```

Суммирование значений элементов массива вручную никак нельзя назвать образцом оптимизированного кода. Позднее мы познакомимся с более удобным способом последовательного обращения к элементам массива.

Присваивание значения элементу массива

Чтобы присвоить значение элементу массива, мы используем выражение *массив*`[`*номерЭлемента*`]` в качестве левого операнда выражения присваивания. Это демонстрирует следующий код:

```
// Создание массива
var cities:Array = ["Toronto", "Montreal", "Vancouver", "Waterloo"];
```

```
// массив cities в настоящий момент выглядит так:
//  ["Toronto", "Montreal", "Vancouver", "Waterloo"]


// Присваиваем значение первому элементу массива
cities[0] = "London";
// Массив cities теперь выглядит так:
// ["London", "Montreal", "Vancouver", "Waterloo"]

// Присваиваем значение четвертому элементу массива
cities[3] = "Hamburg";
// Теперь массив cities выглядит так:
// ["London", "Montreal", "Vancouver", "Hamburg"]

// Присваиваем значение третьему элементу массива
cities[2] = 293.3;  // Обратите внимание, что изменение типа данных
                    // значения элемента не вызывает никаких проблем
// Массив cities теперь выглядит так:
// ["London", "Montreal", 293.3, "Hamburg"]
```

Стоит отметить, что при присваивании значения элементу массива в качестве индекса мы можем использовать любое неотрицательное числовое выражение:

```
var i:int = 1;
// Присваиваем значение элементу с индексом i
cities[i] = "Tokyo";
// Массив cities теперь выглядит так: ["London", "Tokyo", 293.3, "Hamburg"]
```

Определение размера массива

У всех массивов есть переменная экземпляра `length`, обозначающая текущее значение элементов в массиве (включая незаполненные элементы). Для обращения к переменной массива `length` используется оператор «точка» (`.`), как показано в следующем коде:

```
массив.length
```

Рассмотрим несколько примеров:

```
var list:Array = [34, 45, 57];
trace(list.length);              // Выводит: 3

var words:Array = ["this", "that", "the other"];
trace(words.length);             // Выводит: 3

var cards:Array = new Array(24);  // Обратите внимание, что в конструктор
                                  // класса Array передается один числовой
                                  // аргумент
trace(cards.length);              // Выводит: 24
```

Значение переменной массива `length` всегда на единицу больше индекса последнего элемента данного массива. Например, длина массива, элементы которого имеют индексы 0, 1 и 2, равна 3. Длина массива, элементы которого имеют индексы 0, 1,

2 и 50, равна 51. Именно 51. Несмотря на то что элементы с индексами в диапазоне от 3 до 49 являются незаполненными, они все равно учитываются при определении длины массива. Индекс последнего элемента массива всегда равен результату выражения *массив*`.length - 1` (поскольку индексы начинаются с 0, а не с 1). Таким образом, для обращения к последнему элементу массива *массив* применяется следующий код:

```
массив[массив.length - 1]
```

При добавлении и удалении элементов значение переменной массива `length` обновляется автоматически, отражая внесенные изменения. На самом деле мы даже можем сами присвоить переменной `length` значение, чтобы добавить или удалить элементы в конце массива. Этим переменная `length` массива отличается от переменной `length` класса `String`, которая доступна только для чтения. Уменьшение значения переменной `length` массива приводит к удалению всех элементов, индексы которых превышают новое значение.

С помощью переменной массива `length` можно создавать циклы для обхода всех элементов массива. Обход элементов массива в цикле является фундаментальной задачей программирования. Чтобы получить представление о возможностях, открывающихся при совместном использовании циклов и массивов, изучите листинг 11.1, в котором осуществляется обход элементов массива `soundtracks` с целью нахождения позиции элемента со значением `"hip hop"`.

Листинг 11.1. Поиск значения в массиве

```
// Создание массива
var soundtracks:Array = ["electronic", "hip hop",
                         "pop", "alternative", "classical"];

// Проверять каждый элемент, чтобы узнать, содержит ли он значение "hip hop"
for (var i:int = 0; i < soundtracks.length; i++) {
  trace("Now examining element: " + i);
  if (soundtracks[i] == "hip hop") {
    trace("The location of 'hip hop' is index: " + i);
    break;
  }
}
```

Улучшим код листинга 11.1, превратив его в универсальный метод для поиска, который позволит искать произвольный элемент в любом массиве. Если элемент найден, то данный метод вернет позицию найденного элемента в массиве. В противном случае будет возвращено значение `-1`. Листинг 11.2 демонстрирует этот код.

Листинг 11.2. Универсальная функция для поиска элемента в массиве

```
public function searchArray (theArray:Array, searchElement:Object):int {
  // Проверять каждый элемент, чтобы определить, совпадает ли
  // его значение со значением параметра searchElement
  for (var i:int = 0; i < theArray.length; i++) {
    if (theArray[i] == searchElement) {
      return i;
    }
  }
```

```
  return -1;
}
```

Чтобы проверить, есть ли имя "Dan" среди имен массива userNames, который представляет собой гипотетический массив с именами авторизованных пользователей, мы можем воспользоваться нашим новым методом для поиска в массиве:

```
if (searchArray(userNames, "Dan") == -1) {
  trace("Sorry, that username wasn't found");
} else {
  trace("Welcome to the game, Dan.");
}
```

Метод searchArray( ) демонстрирует код, который необходим для выполнения обхода элементов массива в цикле, однако этот код не предназначен для использования в реальной программе. Для определения индекса заданного элемента в реальной программе используйте методы indexOf( ) и lastIndexOf( ) класса Array.

В оставшейся части этой главы речь пойдет о механизмах работы с массивами, включая использование методов класса Array.

Добавление элементов в массив

Добавить элементы в массив можно одним из следующих способов.

- Присвоить значения новому элементу, индекс которого равен значению длины массива или больше его.
- Увеличить значение переменной массива length.
- Вызвать над массивом методы push( ), unshift( ), splice( ) или concat( ).

Рассмотрим более подробно перечисленные способы.

Непосредственное добавление новых элементов

Чтобы добавить новый элемент к существующему массиву по указанному индексу, мы просто присваиваем значение этому элементу. Этот способ продемонстрирован в следующем коде:

```
// Создаем массив и добавляем в него три значения
var fruits:Array = ["apples", "oranges", "pears"];

// Добавляем четвертое значение
fruits[3] = "tangerines";
```

Размещать новый элемент сразу за последним элементом массива не обязательно. Если между новым элементом и концом массива можно разместить несколько элементов, среда выполнения Flash автоматически создаст неопределенные элементы для промежуточных индексов:

```
// Оставить элементы с индексами от 4 до 38 незаполненными
fruits[39] = "grapes";

trace(fruits[12]); // Выводит: undefined
```

Если элемент уже существует, его значение будет заменено новым значением. Если указанный элемент не существует, то он будет добавлен в массив.

Переменная length

Чтобы расширить массив, не присваивая значений новым элементам, можно просто увеличить значение переменной `length`, а среда выполнения Flash добавит необходимое количество элементов для достижения указанной длины:

```
// Создаем массив с тремя элементами
var colors = ["green", "red", "blue"];
// Добавляем в массив 47 незаполненных элементов с индексами от 3 до 49
colors.length = 50;
```

Этот подход можно использовать для создания определенного количества незаполненных элементов, которые будут хранить собираемые данные, например результаты тестов, выполняемых студентами. Хотя элементы являются незаполненными, они позволяют определить, что ожидаемое значение еще не было присвоено. Например, цикл, отображающий результаты тестов на экране, может выводить стандартное сообщение **No Score Available** (Результат недоступен) для незаполненных элементов.

Методы класса Array

Для выполнения более сложных операций по добавлению элементов можно использовать методы класса `Array`.

Метод push( )

Метод `push( )` добавляет один или более элементов в конец массива. Этот метод автоматически добавляет данные сразу за последним нумерованным элементом массива, поэтому вам не нужно беспокоиться о текущей длине массива. Кроме того, метод `push( )` позволяет добавлять в массив сразу несколько элементов. Метод имеет следующий обобщенный вид:

массив`.push(`*элемент1*`,` *элемент2*`...` *элементn*`);`

В предыдущем коде *массив* — это ссылка на объект класса `Array`, а *элемент1*`,` *элемент2*`...` *элементn* — это список элементов, разделенных запятыми, которые добавляются в конец массива и представляют новые элементы. Вот несколько примеров:

```
// Создаем массив с двумя переменными
var menuItems:Array = ["home", "quit"];

// Добавляем элемент
menuItems.push("products");
// Массив menuItems теперь выглядит так: ["home", "quit", "products"]
```

```
// Добавляем два новых элемента
menuItems.push("services", "contact");
// Теперь массив menuItems выглядит так:
// ["home", "quit", "products", "services", "contact"]
```

Метод `push( )` возвращает новую длину измененного массива (то есть значение переменной `length`):

```
var list:Array = [12, 23, 98];
trace(myList.push(28, 36));
// Добавляет в массив list значения 28 и 36 и выводит 5
```

Обратите внимание, что элементы, добавляемые в список, могут быть представлены любым выражением. Выражение вычисляется до того, как элемент будет добавлен в список:

```
var temperature:int = 22;
var sky:String = "sunny";
var weatherListing:Array = new Array( );

// Добавляем значения 22 и "sunny" в массив
weatherListing.push(temperature, sky);
```

Проталкивание, выталкивание и стеки. Метод `push( )` берет свое название из концепции программирования, которая называется *стеком*. Стек может рассматриваться как вертикальный массив, аналогичный стопке тарелок. Если вы часто посещаете кафетерии или рестораны с буфетами, вам должны быть знакомы пружинные подставки, которые удерживают тарелки для клиентов. Когда появляются чистые тарелки, они буквально *проталкиваются* на вершину стека, при этом те тарелки, которые уже находились в подставке, опускаются ниже. Когда клиент *выталкивает* тарелку с вершины стека, он забирает тарелку, которая была добавлена в стек самой последней. Этот тип стека называется «последним пришел — первым вышел» (last-in-first-out — LIFO) и обычно применяется для реализации, например, списков предыстории. Например, если вы нажмете кнопку **Назад** в своем браузере, то откроется предыдущая просмотренная страница. Если снова нажать кнопку **Назад**, то откроется страница, которая просматривалась перед предыдущей, и т. д. Такое поведение достигается путем *проталкивания* URL-адреса каждой просматриваемой страницы в стек и последующего *выталкивания* адреса из стека при нажатии кнопки **Назад**.

Примеры LIFO-стеков можно найти и в реальной жизни. Человек, последним сдавший багаж при посадке на самолет, после приземления самолета обычно получает свой багаж первым, поскольку разгрузка багажа осуществляется в порядке, обратном погрузке. Пассажир, сдавший свой багаж первым, после приземления самолета будет вынуждена простоять у ленты багажного транспортера дольше всех.

Стек типа «первым пришел — первым вышел» (first-in-first-out — FIFO) является более эгалитарным. В основе его функционирования лежит обслуживание в порядке поступления. Примером FIFO-стека является очередь в банке. FIFO-стек работает не с последним элементом массива, а с первым. После этого первый элемент массива удаляется, а все оставшиеся элементы «продвигаются» точно так же, как

продвигается очередь, когда человек, стоявший перед вами, «удаляется» (то есть либо он покинул очередь после того, как был обслужен, либо он решил покинуть очередь раньше времени, устав от ожидания). Таким образом, слово *«проталкивать»* обычно применяется в отношении LIFO-стека, тогда как слово *«добавлять»* применяется в отношении FIFO-стека. В любом случае элементы добавляются в «конец» стека. Разница заключается в том, какой конец массива хранит элемент для следующей операции.

Метод unshift( )

Метод `unshift( )` во многом похож на метод `push( )`, однако он добавляет один или несколько элементов в начало массива, смещая существующие элементы для освобождения пространства (то есть увеличивает индексы существующих элементов, чтобы разместить новые элементы в начале массива). Метод `unshift( )` имеет следующий обобщенный вид:

массив.unshift(*элемент1, элемент2... элементn*);

В приведенном коде *массив* — это ссылка на объект класса `Array`, а *элемент1, элемент2... элементn* — список элементов, разделенных запятыми, которые добавляются в начало массива и представляют новые элементы. Обратите внимание, что элементы добавляются в том порядке, в котором они передаются в метод. Рассмотрим несколько примеров:

```
var versions:Array = new Array( );
versions[0] = 6;
versions.unshift(5);      // Массив versions выглядит так: [5, 6]
versions.unshift(2,3,4);  // Массив versions выглядит так: [2, 3, 4, 5, 6]
```

Метод `unshift( )`, как и метод `push( )`, возвращает длину увеличенного массива.

Метод splice( )

Метод `splice( )` позволяет добавлять в массив или удалять из него элементы. Этот метод обычно применяется для вставки элементов в середину массива (при этом элементы, находящиеся после точки вставки, перенумеровываются, чтобы освободить пространство для добавляемых элементов) или для удаления элементов из середины массива (при этом перенумеровываются элементы, находящиеся после удаляемых элементов, для ликвидации образовавшегося промежутка). Если обе задачи выполняются одновременно за один вызов метода `splice( )`, некоторые элементы массива фактически заменяются новыми элементами (хотя количество добавляемых и удаляемых элементов может не совпадать). Метод `splice( )` имеет следующий обобщенный вид:

массив.splice(*начальныйИндекс, количествоУдаляемыхЭлементов, элемент1, элемент2... элементn*);

В предыдущем коде *массив* — это ссылка на объект класса `Array`; *начальныйИндекс* — число, определяющее индекс, начиная с которого будут выполняться удаление и необязательное добавление элементов (помните, что индексом первого элемента является 0); *количествоУдаляемыхЭлементов* — необязательный аргумент, который определяет количество удаляемых элементов (включая элемент с индексом

начальныйИндекс). Если аргумент *количествоУдаляемыхЭлементов* опущен, все элементы, расположенные после элемента с индексом *начальныйИндекс*, включая сам элемент с данным индексом, будут удалены. Необязательные параметры *элемент1, элемент2... элементn* — объекты, добавляемые в массив в качестве элементов, начиная с индекса *начальныйИндекс*.

Листинг 11.3 демонстрирует разносторонность метода `splice( )`.

Листинг 11.3. Использование метода splice( ) класса Array

```
// Создание массива
var months:Array = new Array("January", "Friday",
                             "April", "May", "Sunday", "Monday", "July");
// С нашим массивом что-то не в порядке. Подправим его.
// Во-первых, избавимся от элемента "Friday".
months.splice(1,1);
// Массив months теперь выглядит так:
// ["January", "April", "May", "Sunday", "Monday", "July"]

// Теперь добавим два месяца перед элементом "April".
// Обратите внимание, что мы ничего не удаляем (deleteCount равен 0).
months.splice(1, 0, "February", "March");
// Массив months теперь выглядит так:
// ["January", "February", "March", "April",
//  "May", "Sunday", "Monday", "July"]

// Наконец, удалим элементы "Sunday" и "Monday", одновременно
// добавив элемент "June".
months.splice(5, 2, "June");
// Массив months теперь выглядит так:
// ["January", "February", "March", "April", "May", "June", "July"]

// Теперь, когда массив months приведен в порядок, обрежем его,
// чтобы остались только названия месяцев первого квартала года,
// удалив все элементы, начиная с индекса 3 (то есть "April").
months.splice(3);  // Теперь массив months выглядит так:
                   // ["January", "February", "March"]
```

Метод `splice( )` возвращает массив удаленных элементов. Таким образом, этот метод можно использовать для извлечения набора элементов из массива:

```
var letters:Array = ["a", "b", "c", "d"];
trace(letters.splice(1, 2));  // Выводит: "b,c"
                              // Массив letters теперь выглядит так:
                              // ["a", "d"]
```

Если никакие элементы не удалены, то метод `splice( )` возвращает пустой массив (то есть массив без элементов).

Метод concat( )

Метод `concat( )` объединяет два или более массива в один новый массив, возвращаемый данным методом. Метод имеет следующий обобщенный вид:

исходныйМассив`.concat(`*списокЭлементов*`)`

Метод `concat( )` один за другим добавляет элементы, содержащиеся в списке *списокЭлементов*, в конец массива *исходныйМассив* и возвращает результат в виде *нового* массива, оставляя массив *исходныйМассив* нетронутым. Обычно возвращаемый массив сохраняется в переменной. Следующий пример демонстрирует простые числа, добавляемые в массив в качестве элементов:

```
var list1:Array = new Array(11, 12, 13);
var list2:Array = list1.concat(14, 15); // Массив list2 теперь выглядит так:
                                        // [11, 12, 13, 14, 15]
```

В следующем примере метод `concat( )` используется для объединения двух массивов:

```
var guests:Array = ["Panda", "Dave"];
var registeredPlayers:Array = ["Gray", "Doomtrooper", "TRK9"];
var allUsers:Array = registeredPlayers.concat(guests);
// Массив allUsers теперь выглядит так:
// ["Gray", "Doomtrooper", "TRK9", "Panda", "Dave"]
```

Обратите внимание, что при добавлении массива `guests` к массиву `allUsers` метод `concat( )` разбил массив `guests` на составляющие, или, иначе говоря, «выпрямил» его. Иными словами, каждый элемент массива `guests` был добавлен к массиву `allUsers` по отдельности. Тем не менее метод `concat( )` не «выпрямляет» вложенные массивы (элементы, которые сами являются массивами внутри основного массива), как показано в следующем коде:

```
var x:Array = [1, 2, 3];
var y:Array = [[5, 6], [7, 8]];
var z:Array = x.concat(y);  // Результат: [1, 2, 3, [5, 6], [7, 8]].
                            // Элементы массива y с индексами 0 и 1
                            // не были «выпрямлены»
```

Удаление элементов из массива

Для удаления элементов из массива можно воспользоваться одним из следующих способов.

- Удалить определенный элемент с помощью оператора `delete`.
- Уменьшить значение переменной массива `length`.
- Вызвать методы `pop( )`, `shift( )` или `splice( )` над массивом.

Рассмотрим подробнее перечисленные способы.

Оператор delete

Оператор `delete` присваивает элементу массива значение `undefined`, используя следующий синтаксис:

```
delete массив[индекс]
```

В этом коде *массив* — это ссылка на массив, а *индекс* — номер или имя элемента, которому должно быть присвоено значение `undefined`. Название оператора

delete, откровенно говоря, вводит в заблуждение. Этот оператор *не* удаляет нумерованный элемент из массива; он просто присваивает указанному элементу значение undefined. Таким образом, операция delete аналогична присваиванию значения undefined элементу массива. В этом легко удостовериться, сравнив значения переменной массива length до и после удаления одного из его элементов:

```
var list = ["a", "b", "c"];
trace(list.length);  // Выводит: 3
delete list[2];
trace(list.length);  // По-прежнему отображает 3. Элемент с индексом 2
                     // вместо значения "c" содержит значение undefined,
                     // но все же этот элемент существует
```

Чтобы удалить элементы на самом деле, используйте метод splice( ) (для удаления элементов из середины массива) или методы shift( ) и pop( ) (для удаления элементов с начала или конца массива соответственно).

Переменная length

Чтобы удалить элементы в конце массива (то есть обрезать массив), можно присвоить переменной массива length значение меньше, чем ее текущее значение:

```
var toppings:Array = ["pepperoni", "tomatoes",
                      "cheese", "green pepper", "broccoli"];
toppings.length = 3;
trace(toppings);  // Выводит: "pepperoni, tomatoes, cheese"
                  // Мы обрезали элементы с индексами 3 и 4 (последние два)
```

Методы класса Array

Массивы обладают несколькими встроенными методами для удаления элементов. Мы уже видели, как с помощью метода splice( ) можно удалять несколько элементов из середины массива. Методы pop( ) и shift( ) применяются для удаления элементов в конце или начале массива.

Метод pop( )

Метод pop( ) является полной противоположностью метода push( ): он удаляет последний элемент массива. Синтаксис метода pop( ) очень прост:

```
массив.pop( )
```

Не знаю почему, но процесс «выталкивания» массива у меня всегда вызывает улыбку. Тем не менее метод pop( ) уменьшает на единицу значение переменной массива length и возвращает значение удаляемого элемента. Например:

```
var numbers:Array = [56, 57, 58];
trace(numbers.pop( )); // Выводит: 58 (значение удаленного элемента)
                       // Массив numbers теперь выглядит так: [56, 57]
```

Как было отмечено ранее, метод pop( ) часто используется совместно с методом push( ) для выполнения операций над LIFO-стеком.

Метод shift( )

Помните метод `unshift( )`, который применяется для добавления элемента в начало массива? Познакомьтесь с его близким другом — методом `shift( )`, который удаляет элемент с начала массива:

массив`.shift( )`

Как и `pop( )`, метод `shift( )` возвращает значение удаляемого элемента. Все оставшиеся элементы в том же порядке продвигаются к началу массива. Например:

```
var sports:Array = ["quake", "snowboarding", "inline skating"];
trace(sports.shift( ));  // Выводит: quake
                         // Массив sports теперь выглядит так:
                         // ["snowboarding", "inline skating"]
trace(sports.shift( ));  // Выводит: snowboarding
                         // Массив sports теперь выглядит так:
                         // ["inline skating"]
```

Поскольку метод `shift( )` на самом деле удаляет элемент, он оказывается более полезным для удаления первого элемента из массива, чем оператор `delete`.

Метод splice( )

В одном из предыдущих разделов мы познакомились с возможностями метода `splice( )` по удалению из массива и добавлению в него элементов. Поскольку метод `splice( )` был рассмотрен достаточно подробно, мы не будем пересматривать его в этом разделе. Тем не менее для информации следующий код демонстрирует возможности метода `splice( )` по удалению элементов:

```
var letters:Array = ["a", "b", "c", "d", "e", "f"];
// Удаляем элементы с индексами 1, 2 и 3, оставляя ["a", "e", "f"]
letters.splice(1, 3);
// Удаляем все элементы, начиная с индекса 1, оставив только ["a"]
letters.splice(1);
```

Проверка содержимого массива с помощью метода toString( )

Метод `toString( )`, характерный для всех объектов, возвращает строковое представление того объекта, над которым он был вызван. В случае с объектом класса `Array` метод `toString( )` возвращает список элементов массива, преобразованных в строки и разделенных запятыми. Метод `toString( )` можно вызывать явно, как показано в следующем коде:

массив`.toString( )`

Однако обычно метод `toString( )` не вызывается явно; вместо этого он вызывается автоматически всякий раз, когда массив *массив* используется в строковом контексте. Например, выражение `trace(`*массив*`)` после отладки отобразит список значений элементов массива, разделенных запятыми. Выражение `trace(`*массив*`)` эквивалентно выражению `trace(`*массив*`.toString( ))`.

Метод `toString( )` часто оказывается полезным при отладке, когда необходимо получить быструю, неформатированную информацию об элементах, содержащихся в массиве. Например:

```
var sites = ["www.moock.org", "www.adobe.com", "www.oreilly.com"];
trace("The sites array is " + sites);
```

Стоит отметить, что метод `join( )` предоставляет более широкие возможности по форматированию информации, чем `toString( )`. Более подробные сведения можно получить в справочнике по языку ActionScript корпорации Adobe.

Многомерные массивы

До сих пор мы рассматривали только *одномерные* массивы, которые аналогичны одной строке или одному столбцу в электронной таблице. Что же делать в том случае, если мы захотим создать эквивалент электронной таблицы с несколькими строками и столбцами? Нам понадобится второе измерение. Язык ActionScript в прямом виде поддерживает только одномерные массивы, однако мы можем имитировать многомерный массив, создав массивы внутри массивов. Другими словами, мы можем создавать массивы, элементами которых являются другие массивы (иногда называемые *вложенными*).

Простейшим типом многомерного массива является двумерный массив, элементы которого концептуально организованы в виде сетки со строками и столбцами. Первое измерение массива представляет строки, а второе — столбцы.

Рассмотрим, как работает двумерный массив, на практическом примере. Предположим, мы обрабатываем заказ на три товара, каждый из которых имеет количество и цену. Мы хотим имитировать электронную таблицу с тремя строками (по одной строке для каждого товара) и двумя столбцами (один столбец для количества, другой — для цены). Мы создаем отдельный массив для каждой строки, при этом каждый элемент строки представляет значение в соответствующем столбце:

```
var row1:Array = [6, 2.99];   // Количество 6, Цена 2.99
var row2:Array = [4, 9.99];   // Количество 4, Цена 9.99
var row3:Array = [1, 59.99];  // Количество 1, Цена 59.99
```

Затем мы помещаем созданные строки в массив-контейнер с именем `spreadsheet`:

```
var spreadsheet:Array = [row1, row2, row3];
```

Теперь мы можем найти общую сумму заказа, перемножив значения количества и цены для каждой строки и сложив получившиеся произведения. Для обращения к элементам двумерного массива используются два индекса (один индекс обозначает строку, другой — столбец). Выражение `spreadsheet[0]`, например, представляет первую строку массива, состоящего из двух столбцов. Таким образом, чтобы обратиться ко второму столбцу первой строки массива `spreadsheet`, мы используем выражение `spreadsheet[0][1]` (оно вернет значение `2,99`). Общая стоимость товаров, содержащихся в массиве `spreadsheet`, вычисляется следующим образом:

```
// Создаем переменную для хранения общей стоимости заказа.
var total:Number;

// Теперь определяем стоимость заказа. Для каждой строки перемножаем
// значения столбцов, а полученное произведение прибавляем к значению
// переменной total.
for (var i:int = 0; i < spreadsheet.length; i++) {
  total += spreadsheet[i][0] * spreadsheet[i][1];
}

trace(total);  // Выводит: 117.89
```

Переходим к событиям

В этой главе дано общее представление о массивах, однако предложенная информация не является исчерпывающей. Класс `Array` обладает множеством полезных методов для переупорядочения и сортировки элементов массива, фильтрации элементов, преобразования элементов в строки и извлечения массивов из других массивов. Более детальное описание класса `Array` можно найти в справочнике по языку ActionScript корпорации Adobe.

Следующей темой изучения является обработка событий — встроенная система для управления взаимодействием между объектами.

События и обработка событий

Событие — это заслуживающее внимания явление, возникающее на этапе выполнения программы и обладающее потенциалом для инициирования ответной реакции. В языке ActionScript события можно разбить на две категории: *предопределенные события*, которые описывают изменения состояния среды выполнения, и *пользовательские события*, описывающие изменения состояния программы. К предопределенным событиям, например, можно отнести щелчок кнопкой мыши или завершение операции загрузки файла. В отличие от этого, к пользовательским событиям можно отнести завершение игры или отправку ответов на вопросы экзамена.

В ActionScript события используются повсеместно. На самом деле в программе, написанной полностью на языке ActionScript, сразу после того, как метод-конструктор основного класса завершает свою работу, выполнение оставшейся части кода инициируется посредством событий. Таким образом, ActionScript поддерживает событийную модель с широкими возможностями, составляющую основу не только для предопределенных, но и для пользовательских событий.

Событийная модель языка ActionScript основана на спецификации W3C Document Object Model (DOM) Level 3, доступной по адресу http://www.w3.org/TR/DOM-Level-3-Events.

В этой главе рассматриваются основы событийной модели языка ActionScript, включая обработку предопределенных событий и реализацию пользовательских событий в программе. Стоит отметить, однако, что в этой главе описываются только основы работы с событиями. Позднее, в гл. 21, будет рассказано, каким образом событийная модель языка ActionScript обеспечивает отображение объектов (объектов, представляющих экранное содержимое). Затем в гл. 22 будет описано все многообразие предопределенных событий пользовательского ввода.

Основы обработки событий в ActionScript

Для *обработки* событий (реакции на события) в программе на языке ActionScript используются *приемники событий*. Приемник событий — это функция или метод, которые выполняются при возникновении определенного события. Они называются так потому, что, по существу, ожидают возникновения событий (или, иначе говоря, принимают возникающие события). Чтобы сообщить программе, что возникло определенное событие, среда выполнения Flash вызывает все приемники событий, которые были зарегистрированы на получение информации о возникновении этого события. Описанный процесс нотификации называется *диспетчеризацией события*.

Перед началом диспетчеризации очередного события среда выполнения создает объект, называемый *событийным объектом*, который представляет данное событие. Событийный объект всегда является экземпляром класса `Event` или одного из его потомков. Все приемники событий, выполняемые в процессе диспетчеризации события, получают в качестве аргумента событийный объект. Любой приемник может использовать переменные событийного объекта для получения информации, касающейся произошедшего события. Например, приемник события, которое представляет активность мыши, может использовать переменные событийного объекта, чтобы определить положение указателя мыши в момент возникновения события.

Каждому типу событий в языке ActionScript, будь то предопределенные или пользовательские события, присваивается строковое имя. Например, именем события типа «щелчок кнопкой мыши» является `"click"`. В процессе диспетчеризации события имя обрабатываемого события может быть получено через переменную `type` событийного объекта, передаваемого в каждый приемник.

Каждая диспетчеризация события в языке ActionScript обладает *получателем*, представляющим объект, которому принадлежит данное событие. Например, для событий ввода получателем события обычно является объект, над которым выполнялись определенные действия (щелкнули кнопкой мыши, ввели информацию, переместили указатель мыши и т. д.). Подобным образом для сетевых событий получателем события обычно является объект, вызвавший сетевую операцию.

Для получения информации о возникновении определенного события приемники обычно регистрируются в получателе события. Соответственно, все объекты получателей событий являются экземплярами класса, унаследованного от класса `EventDispatcher` или реализующего интерфейс `IEventDispatcher`. Класс `EventDispatcher` предоставляет методы для регистрации и отмены регистрации приемников событий (`addEventListener( )` и `removeEventListener( )` соответственно).

В гл. 21 будет рассказано, что, если получателем события является *отображаемый объект* (объект, который может быть отображен на экране), приемники событий могут также зарегистрироваться в *контейнерах отображения* получателя события (то есть в объектах, которые визуально содержат получатель события). Тем не менее пока мы сосредоточимся исключительно на неотображаемых объектах получателей событий.

Регистрация приемника события для получения информации о событии

Основной процесс обработки события в ActionScript заключается в выполнении следующих действий.

1. Определить имя типа события.
2. Определить тип данных событийного объекта, представляющего событие.
3. Создать приемник, отвечающий на событие. Приемник события должен определять один параметр, соответствующий типу данных событийного объекта (тип данных был определен на предыдущем шаге).

4. Теперь следует использовать метод экземпляра `addEventListener( )` класса `EventDispatcher`, чтобы зарегистрировать приемник события в получателе события (или в любом контейнере отображения получателя события).
5. Откинуться на спинку кресла и ожидать возникновения события.

Рассмотрим описанные шаги на примере: создадим и зарегистрируем приемник для предопределенного события `"complete"`.

Шаг 1: Определение имени типа события

Клиентские среды выполнения Flash предлагают широкий выбор типов предопределенных событий, начиная с пользовательского ввода и заканчивая сетевой и звуковой активностью. Имя каждого типа события доступно через константу класса `Event` или одного из его потомков. Например, константой для типа события «операция завершена» является `Event.COMPLETE` со строковым значением `"complete"`. Подобным образом константа для события типа «кнопка мыши нажата» называется `MouseEvent.MOUSE_DOWN`, строковым значением которой является `"mouseDown"`.

Чтобы иметь возможность реагировать на конкретный тип предопределенного события, мы сначала должны найти константу, представляющую это событие. В справочнике по языку ActionScript корпорации Adobe константы событий перечислены в разделе **Events** для каждого класса, поддерживающего события (то есть унаследованного от класса `EventDispatcher`). Таким образом, чтобы найти константу для конкретного предопределенного события, мы обращаемся к разделу **Events** документации по тому классу, которому принадлежит это событие.

Предположим, что мы загружаем внешний текстовый файл с помощью класса `URLLoader` и хотим выполнить некоторый код по завершению загрузки. Мы обращаемся к разделу **Events** документации по классу `URLLoader`, чтобы определить, есть ли у него подходящее нам событие «загрузка завершена». В разделе **Events** мы находим описание события `"complete"`, которое, кажется, нам подходит. Описание события `"complete"` выглядит следующим образом.

```
Событие complete
Тип событийного объекта: flash.events.Event
Свойство Event.type = flash.events.Event.COMPLETE
```

Диспетчеризация события осуществляется после того, как все полученные данные декодированы и помещены в свойство `data` объекта `URLLoader`. Обращаться к полученным данным можно сразу после диспетчеризации этого события.

Подраздел *Свойство Event.type* сообщает нам название константы для события `"complete"` — `flash.events.Event.COMPLETE`. Мы будем использовать эту константу при регистрации приемника для события `"complete"`, как показано полужирным шрифтом в следующем обобщенном коде:

```
объектURLLoader.addEventListener(Event.COMPLETE, некийПриемник);
```

С этого момента при упоминании любых предопределенных событий мы будем использовать соответствующую константу события (например, Event.COMPLETE) вместо его строкового имени-литерала (например, "complete"). Хотя данный стиль является слегка громоздким, он способствует знакомству разработчика с константами событий, фактически применяемых в программах на языке ActionScript.

Шаг 2: Определение типа данных событийного объекта

Теперь, когда мы определили имя типа нашего события (`Event.COMPLETE`), нужно определить тип данных соответствующего событийного объекта. И снова обращаемся к описанию события `"complete"` класса `URLLoader` в справочнике по языку ActionScript корпорации Adobe. Подраздел *Свойство Event.type* описания события `"complete"` (которое приводилось в предыдущем разделе) сообщает нам тип данных объекта `Event` события `Event.COMPLETE` — `flash.events.Event`.

Шаг 3: Создание приемника события

Мы знаем константу и тип данных событийного объекта для нашего события (`Event.COMPLETE` и `Event` соответственно) и можем создать для него соответствующий приемник. Вот этот код:

```
private function completeListener (e:Event):void {
  trace("Load complete");
}
```

Обратите внимание, что в нашем приемнике описан параметр (`e`), который будет принимать событийный объект на этапе диспетчеризации события. Тип данных параметра соответствует типу данных события `Event.COMPLETE`, который был определен на шаге 2.

По соглашению, типом возвращаемого значения всех приемников событий является `void`. Более того, приемники событий — это методы, которые обычно объявляются с использованием модификатора управления доступом `private`, что исключает возможность их вызова кодом за пределами класса, в котором они определены.

Поскольку стандарта по именованию функций и методов приемников событий не существует, для именования приемников событий в этой книге используется формат *имяСобытия*`Listener`, где *имяСобытия* — строковое имя события (в нашем примере `"complete"`).

Шаг 4: Регистрация приемника для события

Теперь, когда мы определили наш приемник события, можно приступать к его регистрации. Если помните, мы загружаем внешний текстовый файл с помощью экземпляра класса `URLLoader`. Этот экземпляр будет нашим получателем события (поскольку он инициирует операцию загрузки, которая в конце концов завершится событием `Event.COMPLETE`). Следующий код создает экземпляр класса `URLLoader`:

```
var urlLoader:URLLoader = new URLLoader( );
```

Следующий код регистрирует наш приемник `completeListener( )` в только что созданном получателе события `urlLoader` для событий `Event.COMPLETE`:

```
urlLoader.addEventListener(Event.COMPLETE, completeListener);
```

Первый аргумент метода `addEventListener( )` задает имя типа события, для которого выполняется регистрация. Второй аргумент метода `addEventListener( )` является ссылкой на регистрируемый приемник.

Рассмотрим полную сигнатуру метода `addEventListener( )`:

`addEventListener(`*тип*`,` *приемник*`,` *использоватьПерехват*`,` *приоритет*`,` *использоватьСлабуюСсылку*`)`

Первые два параметра (*тип* и *приемник*) являются обязательными; остальные — необязательными. Параметры *приоритет* и *использоватьСлабуюСсылку* будут рассмотрены далее в этой главе, а с параметром *использоватьПерехват* мы познакомимся в гл. 21.

Шаг 5: Ожидание возникновения события

Мы создали приемник для события `Event.COMPLETE` и зарегистрировали его в получателе события. Чтобы возникло событие `Event.COMPLETE`, что, в свою очередь, приведет к выполнению метода `completeListener( )`, мы инициируем операцию загрузки файла, как показано в следующем коде:

```
urlLoader.load(new URLRequest("someFile.txt"));
```

Когда загрузка файла `someFile.txt` будет завершена, среда Flash приступит к диспетчеризации события `Event.COMPLETE`, возникшего в объекте `urlLoader`, и выполнит метод `completeListener( )`.

Листинг 12.1 демонстрирует код, представляющий пять описанных шагов в контексте функционального класса `FileLoader`:

Листинг 12.1. Регистрация приемника для событий Event.COMPLETE

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;

  public class FileLoader extends Sprite {
    public function FileLoader ( ) {
      // Создаем получатель события
      var urlLoader:URLLoader = new URLLoader( );
      // Регистрируем приемник события
      urlLoader.addEventListener(Event.COMPLETE, completeListener);
      // Запускаем операцию, которая приведет к возникновению события
      urlLoader.load(new URLRequest("someFile.txt"));
    }

    // Определяем приемник события
    private function completeListener (e:Event):void {
      trace("Load complete");
    }
  }
}
```

Чтобы попрактиковаться, зарегистрируем еще два события.

Два дополнительных примера регистрации приемников событий

В случае если при выполнении кода листинга 12.1 клиентская среда Flash не обнаружит файл `someFile.txt`, то она приступит к диспетчеризации собы-

тия IOErrorEvent.IO_ERROR, получателем которого будет являться объект urlLoader. Зарегистрируем приемник для этого события, чтобы наше приложение могло корректно обрабатывать ошибки загрузки. Сначала мы создадим новый приемник события ioErrorListener( ), как показано в следующем коде:

```
private function ioErrorListener (e:Event):void {
  trace("Error loading file.");
}
```

Теперь зарегистрируем приемник события ioErrorListener( ) в объекте urlLoader для событий IOErrorEvent.IO_ERROR:

```
urlLoader.addEventListener(IOErrorEvent.IO_ERROR, ioErrorListener);
```

Красиво и просто.

В листинге 12.2 демонстрируется новый код, который обрабатывает события IOErrorEvent.IO_ERROR, в контексте класса FileLoader.

Листинг 12.2. Регистрация приемника для событий IOErrorEvent.IO_ERROR

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;

  public class FileLoader extends Sprite {
    public function FileLoader ( ) {
      var urlLoader:URLLoader = new URLLoader( );
      urlLoader.addEventListener(Event.COMPLETE, completeListener);
      urlLoader.addEventListener(IOErrorEvent.IO_ERROR, ioErrorListener);
      urlLoader.load(new URLRequest("someFile.txt"));
    }

    private function completeListener (e:Event):void {
      trace("Load complete");
    }

    private function ioErrorListener (e:Event):void {
      trace("Error loading file.");
    }
  }
}
```

Теперь попытаемся обработать совершенно другое предопределенное событие Event.RESIZE клиентской среды выполнения Flash. Диспетчеризация события Event.RESIZE осуществляется всякий раз, когда среда выполнения Flash находится в режиме «без масштабирования» (no-scale) и изменяется ширина или высота окна приложения. Получателем событий Event.RESIZE является экземпляр класса Stage клиентской среды Flash. Для обращения к этому экземпляру мы воспользуемся переменной stage нашего основного класса приложения ResizeMonitor. Если вы еще незнакомы с экземпляром класса Stage, пока просто считайте, что он представляет область отображения клиентской среды выполнения Flash. Более подробно класс Stage будет рассмотрен в гл. 20.

Вот этот код:

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;

  public class ResizeMonitor extends Sprite {
    public function ResizeMonitor ( ) {
      // Используем режим "без масштабирования". В противном случае при
      // изменении размеров окна приложения происходит автоматическое
      // масштабирование содержимого и события Event.RESIZE не возникают.
      stage.scaleMode = StageScaleMode.NO_SCALE;
      // Регистрируем resizeListener( ) в экземпляре Stage на события
      // Event.RESIZE.
      stage.addEventListener(Event.RESIZE, resizeListener);
    }

    // Определяем приемник события, выполняющийся всякий раз, когда среда
    // выполнения Flash генерирует событие Event.RESIZE
    private function resizeListener (e:Event):void {
      trace("The application window changed size!");
      // Вывод новых размеров экземпляра Stage на консоль
      // с отладочной информацией
      trace("New width: " + stage.stageWidth);
      trace("New height: " + stage.stageHeight);
    }
  }
}
```

Обратите внимание, что внутри функции `resizeListener( )` к переменной `stage` можно обращаться напрямую, как если бы мы находились в методе-конструкторе класса `ResizeMonitor`.

Когда приемником события является метод экземпляра, приемник получает полный доступ к методам и переменным этого экземпляра. Дополнительную информацию можно найти в разд. «Связанные методы» гл. 3.

Отмена регистрации приемника для события

Чтобы остановить процесс получения приемником событий уведомлений о возникающих событиях, мы должны отменить регистрацию этого приемника с помощью метода `removeEventListener( )` экземпляра класса `EventDispatcher`, который имеет следующий обобщенный вид:

получательСобытияИлиПредокПолучателя.removeEventListener(*тип*, *приемник*, *использоватьПерехват*)

В большинстве случаев обязательными являются только первые два параметра (*тип* и *приемник*); параметр *использоватьПерехват* будет рассмотрен в гл. 21.

С целью сокращения объема используемой памяти и уменьшения загрузки процессора отмена регистраций приемников событий должна происходить сразу же после того, как в программе отпадает необходимость их применения.

Следующий код демонстрирует использование метода `removeEventListener( )`; он останавливает процесс получения методом `mouseMoveListener( )` уведомлений о возникающих событиях `MouseEvent.MOUSE_MOVE`, получателем которых является экземпляр класса `Stage`:

```
stage.removeEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
```

Дополнительную информацию о потенциальных проблемах с памятью, которые могут возникать в процессе использования событий, можно найти в разд. «Приемники событий и управление памятью» далее в этой главе.

Обзор терминологии, используемой при работе с событиями

Следующий список терминов, с которыми мы уже встречались в этой главе, представляет ключевую терминологию, используемую при работе с событиями.

Событие — по существу, нечто произошедшее (некое «асинхронное явление»), например щелчок кнопкой мыши или завершение операции загрузки. Каждое событие обозначается *именем события*, которое обычно доступно через константу. Константы для предопределенных событий определяются либо в классе `Event`, либо в одном из его подклассов, который наиболее близко связан с событием.

Событийный объект — объект, представляющий одно конкретное возникновение события. Класс событийного объекта определяет, какая информация о данном событии будет доступна приемникам событий. Все событийные объекты являются экземплярами либо класса `Event`, либо одного из его подклассов.

Получатель события — объект, которому принадлежит событие. Является целевым объектом для события, для которого осуществляется диспетчеризация, и однозначно определяется каждым типом события. Каждый получатель события (и предок получателя в случае получателей в списке отображения) может регистрировать приемники событий, которые будут уведомляться о возникновении события.

Приемник события — функция или метод, которые регистрируются для получения уведомлений о возникновении события от получателя события (или от предка получателя события).

Диспетчеризация события — отправка уведомления о возникновении события получателю события, который вызывает зарегистрированные приемники. Если получатель находится в списке отображения, диспетчеризация события осуществляется по цепочке, от начала списка до получателя и — в случае всплывающих событий — обратно к началу списка. Более подробную информацию о списке отображения и цепочке диспетчеризации событий можно найти в гл. 21. Диспетчеризация события также называется *распространением события*.

Забегая вперед, хочется привести еще несколько терминов, с которыми придется столкнуться при дальнейшем рассмотрении вопросов обработки событий. Говорят, что приемники, выполняемые в ответ на событие, *вызываются* этим событием. Когда вызванный приемник завершает свое выполнение, говорят, что *событие обработано*. Когда все приемники объекта обработают данное событие, говорят, что сам объект завершил обработку этого события.

Теперь, когда мы познакомились с основами событий и их обработки, более детально рассмотрим несколько конкретных вопросов, касающихся обработки событий.

Обращение к объекту получателя

В процессе диспетчеризации любого события объект `Event`, передаваемый в каждый приемник события, определяет переменную `target`, которая содержит ссылку на объект получателя. Таким образом, чтобы обратиться к объекту получателя, мы используем следующий обобщенный код для приемника события, который в процессе отладки просто выводит строковое значение (типа `String`) получателя события:

```
public function некийПриемник (e:SomeEvent):void {
  // Обращение к объекту получателя события
  trace(e.target);
}
```

Программы обычно используют переменную экземпляра `target` класса `Event` для управления объектом получателя. Например, вспомните код, который мы использовали для реакции на завершение операции загрузки файла (продемонстрированный в листинге 12.1):

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;

  public class FileLoader extends Sprite {
    public function FileLoader ( ) {
      var urlLoader:URLLoader = new URLLoader( );
      urlLoader.addEventListener(Event.COMPLETE, completeListener);
      urlLoader.load(new URLRequest("someFile.txt"));
    }

    private function completeListener (e:Event):void {
      trace("Load complete");
    }
  }
}
```

В представленном коде мы могли бы обратиться к объекту `urlLoader` внутри функции `completeListener( )`, чтобы получить содержимое загруженного файла. Рассмотрим код, который мы могли бы использовать (обратите внимание, что для

обеспечения безопасности типов значение переменной `target` приводится к типу `URLLoader` — фактическому типу данных объекта получателя):

```
private function completeListener (e:Event):void {
  var loadedText:String = URLLoader(e.target).data;
}
```

После выполнения этого кода значением переменной `loadedText` станет содержимое загруженного текстового файла (`someFile.txt`).

В листинге 12.3 продемонстрирован еще один пример обращения к объекту получателя события, но на этот раз объект получателя находится в списке отображения. В этом примере, когда текстовое поле получает фокус ввода, цвет фона этого поля становится красным. Для обращения к объекту `TextField` метод `focusInListener( )` использует переменную экземпляра `target` класса `Event`.

В листинге 12.3 применяется несколько методик, которые мы еще не рассматривали: создание текста, установка фокуса ввода на объект, работа со списком отображения и цепочка диспетчеризации событий. Все перечисленные темы будут рассмотрены в части II. Если вы никогда не программировали объекты, отображаемые на экране, пропустите этот пример и вернитесь к нему после прочтения части II.

Листинг 12.3. Обращение к объекту получателя

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  // Изменяет цвет фона текстового поля на красный,
  // когда поле получает фокус ввода
  public class HighlightText extends Sprite {

    // Конструктор
    public function HighlightText ( ) {
      // Создание объекта Sprite
      var s:Sprite = new Sprite( );
      s.x = 100;
      s.y = 100;

      // Создание объекта TextField
      var t:TextField = new TextField( );
      t.text = "Click here";
      t.background = true;
      t.border = true;
      t.autoSize = TextFieldAutoSize.LEFT;

      // Помещение объекта TextField в объект Sprite
      s.addChild(t);

      // Добавляем объект Sprite в иерархию отображения данного объекта
      addChild(s);
```

```
      // Регистрируем приемник для получения уведомлений об установке
      // фокуса ввода на любой из потомков объекта Sprite (в данном случае
      // существует только один потомок: объект TextField, t)
      s.addEventListener(FocusEvent.FOCUS_IN, focusInListener);
    }

    // Приемник выполняется в том случае, когда любой из потомков объекта
    // Sprite получает фокус ввода
    public function focusInListener (e:FocusEvent):void {
      // Выводит: Target of this event dispatch: [object TextField]
      trace("Target of this event dispatch: " + e.target);

      // Устанавливает красный цвет для фона текстового поля. Обратите
      // внимание, что для обеспечения безопасности типов мы приводим
      // значение переменной Event.target к типу TextField - фактическому
      // типу данных объекта получателя.
      TextField(e.target).backgroundColor = 0xFF0000;
    }
  }
}
```

Упражнение: попробуйте добавить к коду листинга 12.3 приемник события `FocusEvent.FOCUS_OUT`, который меняет цвет фона текстового поля — делает его белым.

Обращение к объекту, зарегистрировавшему приемник

В процессе диспетчеризации любого события объект `Event`, передаваемый в каждый приемник события, определяет переменную `currentTarget`, содержащую ссылку на объект, в котором зарегистрирован этот приемник события. Это демонстрирует следующий обобщенный код приемника события; он отображает строковое значение (типа `String`) объекта, в котором зарегистрирован приемник *некийПриемник( )*:

```
public function некийПриемник (e:НекоеСобытие):void {
  // Обращение к объекту, в котором зарегистрирован данный приемник события
  trace(e.currentTarget);
}
```

Для событий, получателями которых являются неотображаемые объекты, значение переменной экземпляра `currentTarget` класса `Event` всегда равняется значению переменной экземпляра `target` (поскольку приемники всегда регистрируются в получателе события). Например, вернемся к классу `FileLoader` из листинга 12.1. Если мы сравним значения переменных `e.currentTarget` и `e.target` внутри метода `completeListener( )`, то увидим, что обе ссылаются на один и тот же объект:

```
package {
  import flash.display.*;
```

```
  import flash.net.*;
  import flash.events.*;

  public class FileLoader extends Sprite {
    public function FileLoader ( ) {
      var urlLoader:URLLoader = new URLLoader( );
      urlLoader.addEventListener(Event.COMPLETE, completeListener);
      urlLoader.load(new URLRequest("someFile.txt"));
    }

    private function completeListener (e:Event):void {
      trace(e.currentTarget == e.target);  // Отображает: true
    }
  }
}
```

Тем не менее, как будет рассказано в гл. 21, для событий, получателями которых являются отображаемые объекты в иерархии отображения, приемники могут регистрироваться как в получателе события, так и в его контейнерах отображения. Для приемников событий, зарегистрированных в контейнере отображения получателя события, переменная `currentTarget` ссылается на этот контейнер, а переменная `target` — на объект получателя события.

Предположим, объект `Sprite`, содержащий объект `TextField`, зарегистрировал приемник события `clickListener( )` для события `MouseEvent.CLICK`. Когда пользователь щелкает кнопкой мыши в этом текстовом поле, происходит диспетчеризация события `MouseEvent.CLICK`, в результате чего вызывается приемник `clickListener( )`. Внутри метода `clickListener( )` переменная `currentTarget` ссылается на объект `Sprite`, а переменная `target` — на объект `TextField`.

Программы обычно используют переменную `currentTarget` для управления объектом, в котором зарегистрирован приемник. В качестве примера модифицируем функцию `focusInListener( )` из листинга 12.3. На этот раз при получении объектом `TextField` фокуса ввода наш новый код функции `focusInListener( )` будет отображать синий овал вокруг текстового поля. Синий овал рисуется на объекте `Sprite`, доступ к которому осуществляется через переменную `currentTarget`.

```
public function focusInListener (e:FocusEvent):void {
  // Установить для фона текстового поля красный цвет
  TextField(e.target).backgroundColor = 0xFF0000;

  // Получить ссылку на объект Sprite
  var theSprite:Sprite = Sprite(e.currentTarget);

  // Нарисовать эллипс на объекте Sprite
  theSprite.graphics.beginFill(0x0000FF);
  theSprite.graphics.drawEllipse(-10, -10, 75, 40);
}
```

Отмена стандартного поведения событий

Некоторые события в языке ActionScript обладают побочным эффектом, называемым *стандартным поведением*. Например, стандартным поведением события `TextEvent.TEXT_INPUT` является добавление текста в текстовое поле получателя. Подобным образом стандартным поведением события `MouseEvent.MOUSE_DOWN`, получаемого объектом класса `SimpleButton`, является отображение картинки, которая представляет нажатое состояние кнопки.

В некоторых случаях события, которые характеризуются стандартным поведением, предоставляют возможность избежать этого поведения программным путем. Такие события называются *отменяемыми*. Например, отменяемыми являются события `TextEvent.TEXT_INPUT`, `FocusEvent.KEY_FOCUS_CHANGE` и `FocusEvent.MOUSE_FOCUS_CHANGE`.

Чтобы избежать стандартного поведения для отменяемого события, мы вызываем метод экземпляра `preventDefault( )` класса `Event` над объектом этого класса, передаваемым во все приемники, зарегистрированные для данного события. Например, в следующем коде мы отменяем стандартное поведение для всех событий `TextEvent.TEXT_INPUT`, получателем которых является текстовое поле `t`. Вместо того чтобы позволить программе отображать в текстовом поле текст, вводимый пользователем, мы будем просто добавлять в это поле букву `"x"`.

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;

  // Изменить текст, вводимый пользователем, на символ "x"
  public class InputConverter extends Sprite {
    private var t:TextField;

    public function InputConverter ( ) {
      // Создаем текстовое поле
      t = new TextField( );
      t.border = true;
      t.background = true;
      t.type = TextFieldType.INPUT
      addChild(t);

      // Регистрируем приемник для события TextEvent.TEXT_INPUT
      t.addEventListener(TextEvent.TEXT_INPUT, textInputListener);
    }

    // Приемник выполняется при возникновении события TextEvent.TEXT_INPUT
    private function textInputListener (e:TextEvent):void {
      // Показать текст, введенный пользователем
      trace("Attempted text input: " + e.text);

      // Исключить отображение введенного текста в текстовом поле
      e.preventDefault( );
```

```
      // Добавить в текстовое поле букву "x" вместо введенного
      // пользователем текста
      t.appendText("x");
    }
  }
}
```

Чтобы определить, обладает ли определенное событие стандартным поведением, которое можно отменить, проверьте значение переменной экземпляра `cancelable` класса `Event` внутри приемника, зарегистрированного для получения уведомлений о возникновении данного события. Для предопределенных событий эту информацию можно также найти в разделе описания соответствующего события в справочнике по языку ActionScript корпорации Adobe.

Чтобы определить, было ли отменено стандартное поведение события, диспетчеризация которого происходит в текущий момент, проверьте возвращаемое значение метода экземпляра `isDefaultPrevented( )` класса `Event` внутри приемника, зарегистрированного для получения уведомлений о возникновении данного события.

Стоит отметить, что, как и предопределенные события, пользовательские события имеют возможность определять стандартное поведение, которое может быть отменено вызовом метода `preventDefault( )`. Дополнительную информацию вместе с примером кода можно найти в подразд. «Отмена стандартного поведения для пользовательских событий» разд. «Пользовательские события» далее в этой главе.

Еще один пример, демонстрирующий использование метода preventDefault( ) для события TextEvent.TEXT_INPUT, показан в листинге 22.8 гл. 22.

Приоритет приемника события

По умолчанию, если сразу несколько приемников событий регистрируются в конкретном объекте для получения уведомлений об одном и том же типе событий, они вызываются в том порядке, в котором были зарегистрированы. Например, в следующем коде два приемника событий — `completeListenerA( )` и `completeListenerB( )` — регистрируются в объекте `urlLoader` для получения уведомлений о событии `Event.COMPLETE`. При возникновении события `Event.COMPLETE` приемник `completeListenerA( )` будет выполнен раньше приемника `completeListenerB( )`, поскольку `completeListenerA( )` был зарегистрирован раньше `completeListenerB( )`.

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;

  public class FileLoader extends Sprite {
    public function FileLoader ( ) {
      var urlLoader:URLLoader = new URLLoader( );
      // Порядок регистрации определяет порядок выполнения
      urlLoader.addEventListener(Event.COMPLETE, completeListenerA);
```

```
      urlLoader.addEventListener(Event.COMPLETE, completeListenerB);
      urlLoader.load(new URLRequest("someFile.txt"));
    }

    private function completeListenerA (e:Event):void {
      trace("Listener A: Load complete");
    }

    private function completeListenerB (e:Event):void {
      trace("Listener B: Load complete");
    }
  }
}
```

Изменить стандартный порядок вызова приемников событий можно с помощью параметра *приоритет* метода `addEventListener( )`, показанного в следующем обобщенном коде:

```
addEventListener(тип, приемник, использоватьПерехват, приоритет,
использоватьСлабуюСсылку)
```

Параметр *приоритет* представляет собой целое число, обозначающее порядок, в котором должен вызываться регистрируемый приемник события относительно других приемников, зарегистрированных для того же события в том же объекте. Приемники, зарегистрированные с более высоким значением параметра *приоритет*, будут вызваны раньше приемников, зарегистрированных с более низким значением. Например, приемник, зарегистрированный со значением `3` параметра *приоритет*, будет вызван раньше приемника, зарегистрированного со значением `2` параметра *приоритет*. Если два приемника зарегистрированы с одним и тем же значением параметра *приоритет*, они будут выполняться в том порядке, в котором были зарегистрированы. Если значение параметра *приоритет* не указано, принимается значение по умолчанию, равное `0`.

Следующий код демонстрирует принципы использования параметра *приоритет*. Приемник `completeListenerB( )` будет выполнен раньше приемника `completeListenerA( )` несмотря на то, что `completeListenerA( )` был зарегистрирован раньше `completeListenerB( )`.

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;

  public class FileLoader extends Sprite {
    public function FileLoader ( ) {
      var urlLoader:URLLoader = new URLLoader( );
      // Параметр приоритет определяет порядок выполнения
      urlLoader.addEventListener(Event.COMPLETE,
                                 completeListenerA,
                                 false,
                                 0);
      urlLoader.addEventListener(Event.COMPLETE,
                                 completeListenerB,
```

```
                                    false,
                                    1);
        urlLoader.load(new URLRequest("someFile.txt"));
    }

    private function completeListenerA (e:Event):void {
        trace("Listener A: Load complete");
    }

    private function completeListenerB (e:Event):void {
        trace("Listener B: Load complete");
    }
  }
}
```

Параметр *приоритет* применяется крайне редко, однако в некоторых ситуациях он может оказаться весьма полезным. Например, среда разработки приложений может использовать приемник с более высоким приоритетом, чтобы инициировать загруженное приложение до того, как будут выполнены другие приемники. Кроме того, программный пакет для тестирования может использовать приемник с более высоким приоритетом, чтобы заблокировать другие приемники, которые в противном случае повлияют на результаты конкретного теста (дополнительную информацию можно получить в разд. «Остановка процесса диспетчеризации события» гл. 21).

Будьте осторожны при изменении порядка выполнения приемников событий. Программы, зависящие от порядка выполнения, предрасположены к ошибкам, поскольку приоритеты приемников событий являются непостоянными, усложняют поддержку кода и делают его более сложным для восприятия.

Приемники событий и управление памятью

В этой главе мы увидели, что событийная модель языка ActionScript основывается на приемнике (представляющем собой функцию или метод) и объекте, в котором регистрируется этот приемник. Каждый объект, регистрирующий приемник для определенного события, поддерживает связь с этим приемником, сохраняя ссылку на него во внутреннем массиве, который называется *списком приемников*. Например, в следующем коде (взятом из листинга 12.1) метод `completeListener( )` регистрируется в объекте `urlLoader` для событий `Event.COMPLETE`. В результате внутренний *список приемников* объекта `urlLoader` получает ссылку на метод `completeListener( )`.

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;

  public class FileLoader extends Sprite {
    public function FileLoader ( ) {
      var urlLoader:URLLoader = new URLLoader( );
```

```
      // Регистрация приемника completeListener( )
      urlLoader.addEventListener(Event.COMPLETE, completeListener);
      urlLoader.load(new URLRequest("someFile.txt"));
    }

    private function completeListener (e:Event):void {
      trace("Load complete");
    }
  }
}
```

По умолчанию любой объект, получивший ссылку на приемник, хранит ее до тех пор, пока регистрация данного приемника не будет явно отменена методом `removeEventListener( )`. Более того, объект продолжает хранить полученную ссылку на приемник даже тогда, когда в программе не остается других ссылок на этот приемник. Это демонстрирует следующий простой класс `AnonymousListener`. Он создает анонимную функцию и регистрирует ее для событий `MouseEvent.MOUSE_EVENT` в экземпляре `Stage` клиентской среды выполнения Flash. Хотя класс `AnonymousListener` не имеет ссылок на эту анонимную функцию, она продолжает храниться в экземпляре `Stage` и вызывается каждый раз при возникновении события `MouseEvent.MOUSE_MOVE`, даже спустя долгое время после завершения метода конструктора класса `AnonymousListener`.

```
package {
  import flash.display.*;
  import flash.events.*;

  public class AnonymousListener extends Sprite {
    public function AnonymousListener ( ) {
      // Добавляем анонимную функцию в список приемников экземпляра Stage
      stage.addEventListener(MouseEvent.MOUSE_MOVE,
                             function (e:MouseEvent):void {
                               trace("mouse move");
                             });
    }
  }
}
```

В предыдущем коде созданная анонимная функция окажется навсегда «заброшенной» в списке приемников экземпляра `Stage`. Программа не сможет отменить регистрацию анонимной функции, поскольку она не имеет ссылки на нее.

«Заброшенные» приемники представляют собой потенциальный источник существенного расходования памяти и могут привести к другим побочным эффектам в программах на языке ActionScript.

Рассмотрим пример, демонстрирующий потенциальные проблемы, которые могут возникнуть из-за «заброшенных» приемников, и способы их решения.

Предположим, мы разрабатываем игру, которая заключается в ловле бабочек: чтобы поймать бабочку, игрок должен коснуться ее указателем мыши. Бабочки

не хотят быть пойманными и всячески стараются улететь от указателя мыши. Основным классом приложения является ButterflyGame. Каждая бабочка представляется экземпляром класса Butterfly. Для управления перемещением бабочек в игре используется основной объект Timer, который генерирует событие TimerEvent.TIMER каждые 25 мс. Каждый объект класса Butterfly регистрирует приемник в основном объекте Timer и вычисляет свое новое местоположение всякий раз, когда возникает событие TimerEvent.TIMER.

Рассмотрим код класса Butterfly:

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.utils.*;

  public class Butterfly extends Sprite {
    // Каждый объект Butterfly получает ссылку на основной таймер
    // через параметр конструктора gameTimer
    public function Butterfly (gameTimer:Timer) {
      gameTimer.addEventListener(TimerEvent.TIMER, timerListener);
    }

    private function timerListener (e:TimerEvent):void {
      trace("Calculating new butterfly position...");
      // Вычисление нового местоположения бабочки (код не приводится)
    }
  }
}
```

Рассмотрим код класса ButterflyGame, который значительно упрощен, чтобы акцентировать ваше внимание на коде, отвечающем за создание и удаление бабочки. В этой версии кода игра содержит только одну бабочку.

```
package {
  import flash.display.*;
  import flash.utils.*;

  public class ButterflyGame extends Sprite {
    private var timer:Timer;
    private var butterfly:Butterfly;

    public function ButterflyGame ( ) {
      // Игровой таймер
      timer = new Timer(25, 0);
      timer.start( );
      addButterfly( );
    }

    // Добавляет бабочку в игру
    public function addButterfly ( ):void {
      butterfly = new Butterfly(timer);
    }
```

```
    // Удаляет бабочку из игры
    public function removeButterfly ( ):void {
      butterfly = null;
    }
  }
}
```

Для добавления бабочки в игру класс `ButterflyGame` использует следующий код:

```
butterfly = new Butterfly(timer);
```

Этот код приведет к выполнению конструктора класса `Butterfly`, в результате чего метод `timerListener( )` класса `Butterfly` зарегистрируется в объекте `gameTimer` для получения событий `TimerEvent.TIMER`.

Когда игрок поймает бабочку, объект `ButterflyGame` удалит соответствующий объект `Butterfly` из программы, используя следующий код:

```
butterfly = null;
```

Однако, даже несмотря на то, что предыдущий код удаляет ссылку на объект `Butterfly` из объекта `ButterflyGame`, в списке приемников объекта `gameTimer` продолжает храниться ссылка на метод `timerListener( )` объекта `Butterfly` и, соответственно, на сам объект `Butterfly`. Более того, метод `timerListener( )` продолжает выполняться каждый раз при возникновении события `TimerEvent.TIMER`. Таким образом, объект `Butterfly` продолжает потреблять память и процессорное время и способен вызвать неожиданные или нежелательные побочные эффекты в программе. Чтобы избежать подобных проблем, перед удалением объекта `Butterfly` из игры нужно сначала отменить регистрацию метода `timerListener( )` для событий `TimerEvent.TIMER`.

Добавим новую переменную `gameTimer` и новый метод `destroy( )` в класс `Butterfly`, чтобы облегчить процесс отмены регистрации приемника для события `TimerEvent.TIMER`. Основной игровой таймер присваивается переменной `gameTimer`. Метод `destroy( )` отменяет регистрацию метода `timerListener( )` для событий `TimerEvent.TIMER`. Рассмотрим код класса `Butterfly` с внесенными изменениями, которые выделены полужирным шрифтом:

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.utils.*;

  public class Butterfly extends Sprite {
    private var gameTimer:Timer;

    public function Butterfly (gameTimer:Timer) {
      this.gameTimer = gameTimer;
      this.gameTimer.addEventListener(TimerEvent.TIMER, timerListener);
    }

    private function timerListener (e:TimerEvent):void {
      trace("Calculating new butterfly position...");
```

```
      // Вычисление нового местоположения бабочки (код не показан)
    }

    public function destroy ( ):void {
      gameTimer.removeEventListener(TimerEvent.TIMER, timerListener);
    }
  }
}
```

Перед тем как удалить ссылку на объект `Butterfly`, в методе экземпляра `removeButterfly( )` класса `ButterflyGame` мы вызываем метод `destroy( )`, как показано в следующем коде:

```
public function removeButterfly ( ):void {
  butterfly.destroy( );
  butterfly = null;
}
```

Вызывая метод `destroy( )` перед удалением объекта `Butterfly` из игры, мы не позволяем методу `timerListener( )` оказаться «заброшенным» в списке приемников объекта `Timer`.

Если вы регистрируете приемник события в каком-либо объекте, убедитесь, что ваша программа с течением времени также отменяет регистрацию этого приемника.

Слабые ссылки на приемники событий. В предыдущем разделе было рассказано, что по умолчанию объект, регистрирующий приемник для определенного события, хранит ссылку на этот приемник до тех пор, пока его регистрация для указанного события не будет отменена явно, даже если в программе не остается других ссылок на этот приемник. Тем не менее это стандартное поведение можно изменить с помощью параметра *использоватьСлабуюСсылку* метода `addEventListener( )`.

Для изучения этого раздела требуется предварительное понимание механизма сборки мусора в языке ActionScript. Этот механизм рассматривается в гл. 14.

Регистрация приемника с использованием параметра *использоватьСлабуюСсылку*, для которого установлено значение `true`, не позволит этому приемнику оказаться «заброшенным» в списке приемников объекта, выполняющего регистрацию. Предположим, что объект (`О`) регистрирует приемник (`П`) для события (`С`) с использованием параметра *использоватьСлабуюСсылку*, для которого установлено значение `true`. Кроме того, предположим, что единственной ссылкой на `П`, которой обладает программа, является ссылка, хранящаяся в `О`. В обычной ситуации `П` будет храниться в `О` до тех пор, пока регистрация `П` для события `С` не будет отменена. Однако поскольку при регистрации `П` был использован параметр *использоватьСлабуюСсылку* со значением `true` и `О` хранит единственную оставшуюся ссылку на `П` в программе, `П` сразу же становится пригодным для сборки мусора. Впоследствии сборщик мусора по своему усмотрению может автоматически исключить `П` из списка приемников `О` и удалить его из памяти.

Для демонстрации работы параметра *использоватьСлабуюСсылку* вернемся к классу `AnonymousListener`. Как уже говорилось, класс `AnonymousListener` создает

анонимную функцию и регистрирует ее для событий MouseEvent.MOUSE_MOVE в экземпляре Stage клиентской среды выполнения Flash. Однако на этот раз при регистрации функции для событий MouseEvent.MOUSE_MOVE мы используем параметр *использоватьСлабуюСсылку* со значением true.

```
package {
  import flash.display.*;
  import flash.events.*;

  public class AnonymousListener extends Sprite {
    public function AnonymousListener ( ) {
      // Добавляем анонимную функцию в список приемников экземпляра Stage
      stage.addEventListener(MouseEvent.MOUSE_MOVE,
                             function (e:MouseEvent):void {
                               trace("mouse move");
                             },
                             false,
                             0,
                             true);
    }
  }
}
```

После выполнения предыдущего кода единственной ссылкой на анонимную функцию в программе является ссылка, хранящаяся в экземпляре Stage. Поскольку анонимная функция была зарегистрирована с использованием параметра *использоватьСлабуюСсылку*, для которого было установлено значение true, она сразу же становится пригодной для сборки мусора. Таким образом, сборщик мусора по своему усмотрению может впоследствии автоматически исключить анонимную функцию из списка приемников экземпляра Stage и удалить ее из памяти.

Безусловно, тот факт, что анонимная функция *пригодна* для сборки мусора, совершенно не означает, что она *будет* удалена из памяти. На самом деле в случае с предыдущим простым примером, функция, скорее всего, *не* будет удалена из памяти, поскольку объем используемой приложением памяти недостаточен для запуска механизма сборки мусора. В результате функция будет продолжать выполняться при получении события MouseEvent.MOUSE_MOVE экземпляром Stage, хотя теоретически она может быть удалена из памяти в любой момент времени. Из этого следует, что вообще не следует полагаться на параметр *использоватьСлабуюСсылку* как на способ автоматического удаления приемников событий. Наилучшее решение — просто ***избегать появления заброшенных приемников событий.***

Если вы регистрируете приемник события в каком-либо объекте, убедитесь, что ваша программа с течением времени также отменяет регистрацию этого приемника.

До сих пор в этой главе мы работали исключительно с предопределенными событиями языка ActionScript. Теперь рассмотрим процесс создания в программе своих собственных пользовательских событий.

Пользовательские события

Чтобы выполнить диспетчеризацию нового пользовательского события в языке ActionScript, достаточно расширить класс `EventDispatcher`, присвоить новому событию имя и вызвать метод экземпляра `dispatchEvent( )` класса `EventDispatcher`. Для изучения процесса создания пользовательских событий в программе мы рассмотрим два примера: в первом событие создается для игры, а во втором — для элемента пользовательского интерфейса.

Если вы хотите сделать получателем события экземпляр класса, который уже расширяет другой класс, используйте композиционный подход, рассмотренный в гл. 9: непосредственно реализуйте интерфейс IEventDispatcher и используйте методы класса EventDispatcher не через наследование, а через композицию.

Пользовательское событие "gameOver"

Предположим, что мы создаем универсальный каркас для разработки видеоигр. Каркас включает в себя следующие два класса: класс `Game`, выполняющий основные функции, необходимые для любой видеоигры, и класс `Console`, который представляет панель для запуска новых игр. Каждый раз при запуске новой игры класс `Console` создает объект класса `Game`. Любой экземпляр класса `Game`, создаваемый классом `Console`, является получателем пользовательского события `"gameOver"`, которое возникает после окончания игры.

Чтобы объекты класса `Game` могли выступать в роли получателей событий, класс `Game` расширяет класс `EventDispatcher`, как показано в следующем коде:

```
package {
  import flash.events.*;

  public class Game extends EventDispatcher {
  }
}
```

Кроме того, в классе `Game` определена константа `Game.GAME_OVER`, значением которой является имя пользовательского события: `"gameOver"`. По соглашению имена констант событий записываются полностью прописными буквами, а слова разделяются знаком подчеркивания, например: `GAME_OVER`. Константы пользовательских событий обычно определяются либо в классе получателя события (в данном случае в классе `Game`), либо, если используется подкласс класса `Event`, в этом подклассе (как показано в нашем следующем примере с элементом пользовательского интерфейса). Поскольку в нашем текущем примере подклассы класса `Event` не используются, определим константу для события `"gameOver"` в классе `Game`, как показано в следующем коде:

```
package {
  import flash.events.*;
```

```
  public class Game extends EventDispatcher {
    public static const GAME_OVER:String = "gameOver";
  }
}
```

Когда игра завершается, объект `Game` вызывает метод `endGame ( )`, который возвращает игровую среду в исходное состояние, что позволит начать новую игру. Вот код метода `endGame ( )`:

```
package {
  import flash.events.*;

  public class Game extends EventDispatcher {
    public static const GAME_OVER:String = "gameOver";

    private function endGame ( ):void {
      // Выполнение действий для завершения игры (код не показан)
    }
  }
}
```

Когда все действия, завершающие игру, выполнены, метод `endGame ( )` использует метод `dispatchEvent ( )`, чтобы приступить к диспетчеризации события `Game.GAME_OVER`, сигнализирующего об окончании игры:

```
package {
  import flash.events.*;

  public class Game extends EventDispatcher {
    public static const GAME_OVER:String = "gameOver";

    private function endGame ( ):void {
      // Выполнение действий для завершения игры
      // (код не показан)

      // ...после чего просим среду Flash выполнить
      // диспетчеризацию события, обозначающего окончание игры
      dispatchEvent(new Event(Game.GAME_OVER));
    }
  }
}
```

Обратите внимание, что, поскольку метод `dispatchEvent ( )` вызывается над объектом `Game`, этот объект и является получателем события.

Объект, над которым вызывается метод dispatchEvent( ), является получателем события.

Метод `dispatchEvent ( )`, продемонстрированный в предыдущем коде, принимает один параметр — объект `Event`, предоставляющий событие для диспетчеризации. Сам конструктор класса `Event` принимает три параметра — *тип*, *всплывающее* и *отменяемое*, как показано в следующем обобщенном коде:

```
Event(тип, всплывающее, отменяемое)
```

Тем не менее в большинстве случаев требуется только первый аргумент — *тип*; он определяет строковое имя события (в нашем случае `Game.GAME_OVER`). Параметр *всплывающее* используется только в тех случаях, когда получателем события является отображаемый объект; этот параметр обозначает присутствие (`true`) или отсутствие (`false`) всплывающей фазы в цепочке диспетчеризации событий (дополнительную информацию можно найти в разд. «Пользовательские события и цепочка диспетчеризации события» гл. 21). Параметр *отменяемое* применяется для создания пользовательских событий с избегаемым стандартным поведением (этот вопрос рассматривается далее в подразд. «Отмена стандартного поведения для пользовательских событий» разд. «Пользовательские события»).

Чтобы зарегистрировать приемник события для нашего пользовательского события `Game.GAME_OVER`, как и при регистрации приемника для предопределенного события, используется метод `addEventListener( )`. Предположим, что по окончании игры мы хотим, чтобы класс `Console` выводил окно, позволяющее пользователю вернуться к панели запуска или снова сыграть в выбранную игру. Определить момент окончания игры в классе `Console` позволит регистрация приемника для событий `Game.GAME_OVER`, как показано в следующем коде:

```
package {
  import flash.display.*;
  import flash.events.*;

  public class Console extends Sprite {

    // Конструктор
    public function Console ( ) {
      var game:Game = new Game( );
      game.addEventListener(Game.GAME_OVER, gameOverListener);
    }

    private function gameOverListener (e:Event):void {
      trace("The game has ended!");
      // Отображает пользовательский интерфейс "back to console"
      // (код не показан)
    }
  }
}
```

Обратите внимание, что тип данных событийного объекта, передаваемого в метод `gameOverListener( )`, соответствует типу данных событийного объекта, изначально передаваемого в метод `dispatchEvent( )` внутри метода экземпляра `endGame( )` класса `Game` (как показано в предыдущем коде).

При создании приемника для пользовательского события указывайте тип данных параметра приемника в соответствии с типом данных событийного объекта, изначально передаваемого в метод dispatchEvent( ).

В листинге 12.4 полностью показан код для нашего пользовательского события `Game.GAME_OVER`, который также содержит таймер, вызывающий метод

endGame( ), имитируя окончание реальной игры (подробную информацию по классу Timer можно найти в справочнике по языку ActionScript корпорации Adobe).

Листинг 12.4. Пользовательское событие "gameOver"

```
// Класс Game (получатель события)
package {
  import flash.events.*;
  import flash.utils.*;  // Требуется для класса Timer

  public class Game extends EventDispatcher {
    public static const GAME_OVER:String = "gameOver";

    public function Game ( ) {
      // Завершает игру спустя одну секунду
      var timer:Timer = new Timer(1000, 1);
      timer.addEventListener(TimerEvent.TIMER, timerListener);
      timer.start( );
      // Вложенная функция, которая выполняется через одну секунду
      // после создания данного объекта
      function timerListener (e:TimerEvent):void {
        endGame( );
      }
    }

    private function endGame ( ):void {
      // Выполнение действий для завершения игры (код не показан)

      // ...после чего просим среду Flash
      // выполнить диспетчеризацию события,
      // обозначающего окончание игры
      dispatchEvent(new Event(Game.GAME_OVER));
    }
  }
}

// Класс Console (регистрирует приемник для события)
package {
  import flash.display.*;
  import flash.events.*;

  public class Console extends Sprite {

    // Конструктор
    public function Console ( ) {
      var game:Game = new Game( );
      game.addEventListener(Game.GAME_OVER, gameOverListener);
    }

    private function gameOverListener (e:Event):void {
      trace("The game has ended!");
```

```
      // Отображает пользовательский интерфейс "back to console" (код не показан)
    }
  }
}
```

Теперь рассмотрим другой пример, создающий событие для элемента пользовательского интерфейса.

Пользовательское событие "toggle"

Предположим, что мы создаем кнопку-переключатель, которая может быть использована в пользовательском интерфейсе. Она может принимать два положения — включения и выключения. Наша кнопка-переключатель представлена классом `ToggleSwitch`. Всякий раз, когда кнопка включается или выключается, мы инициируем диспетчеризацию пользовательского события с именем `"toggle"`.

В предыдущем разделе событийный объект для нашего пользовательского события `Game.GAME_OVER` представлял собой экземпляр внутреннего класса `Event`. На этот раз наше пользовательское событие будет представлено своим собственным классом `ToggleEvent`. Этот класс выполняет две следующие функции:

- определяет константу для события `toggle` (`ToggleEvent.TOGGLE`);
- задает переменную `isOn`, которая будет использоваться приемниками для определения состояния объекта получателя `ToggleSwitch`.

Далее представлен код класса `ToggleEvent`. Обратите внимание, что каждый пользовательский подкласс класса `Event` должен переопределять методы `clone( )` и `toString( )`, предоставляя версии методов, которые учитывают все пользовательские переменные данного подкласса (например, `isOn`).

Код кнопки-переключателя в этом разделе демонстрирует исключительно реализацию события `"toggle"`. Код, необходимый для создания интерактивности и добавления графики, опущен.

```
package {
  import flash.events.*;

  // Класс, представляющий пользовательское событие "toggle"
  public class ToggleEvent extends Event {
    // Константа для типа события "toggle"
    public static const TOGGLE:String = "toggle";

    // Обозначает, включен или выключен переключатель
    public var isOn:Boolean;

    // Конструктор
    public function ToggleEvent (type:String,
                                 bubbles:Boolean = false,
                                 cancelable:Boolean = false,
                                 isOn:Boolean = false) {
      // Передаем параметры конструктора в конструктор суперкласса
      super(type, bubbles, cancelable);
```

```
      // Запоминаем состояние переключателя, которое может быть использовано
      // в приемниках события ToggleEvent.TOGGLE
      this.isOn = isOn;
    }

    // Любой класс пользовательского события должен переопределить
    // метод clone( )
    public override function clone( ):Event {
      return new ToggleEvent(type, bubbles, cancelable, isOn);
    }

    // Любой класс пользовательского события должен переопределить
    // метод toString( ). Обратите внимание, что "eventPhase" – это
    // переменная экземпляра, имеющая отношение к цепочке диспетчеризации
    // событий (гл. 21).
    public override function toString( ):String {
      return formatToString("ToggleEvent", "type", "bubbles",
                            "cancelable", "eventPhase", "isOn");
    }
  }
}
```

Теперь перейдем к классу `ToggleSwitch`, который представляет кнопку-переключатель. Единственный метод `toggle( )` класса `ToggleSwitch` изменяет состояние кнопки-переключателя, а затем выполняет диспетчеризацию события `ToggleEvent.TOGGLE`, которое обозначает изменение состояния переключателя. Следующий код демонстрирует класс `ToggleSwitch`. Обратите внимание, что класс `ToggleSwitch` расширяет класс `Sprite`, предоставляющий возможности для отображения объекта на экране. Кроме того, класс `Sprite`, являясь потомком класса `EventDispatcher`, предоставляет необходимую функциональность для диспетчеризации событий:

```
package {
  import flash.display.*;
  import flash.events.*;

  // Представляет простой элемент-переключатель
  public class ToggleSwitch extends Sprite {
    // Запоминает состояние переключателя
    private var isOn:Boolean;

    // Конструктор
    public function ToggleSwitch ( ) {
      // По умолчанию переключатель выключен
      isOn = false;
    }

    // Включает переключатель, если он был выключен, или
    // выключает его
    public function toggle ( ):void {
      // Изменяет состояние переключателя
      isOn = !isOn;
```

```
      // просим среду Flash выполнить диспетчеризацию
      // события ToggleEvent.TOGGLE, получателем которого
      // является данный объект ToggleSwitch
      dispatchEvent(new ToggleEvent(ToggleEvent.TOGGLE,
                                    true,
                                    false,
                                    isOn));
    }
  }
}
```

Чтобы продемонстрировать использование события `ToggleEvent.TOGGLE`, создадим простой класс `SomeApp`. Этот класс определяет метод `toggleListener( )` и регистрирует его в объекте `ToggleSwitch` для событий `ToggleEvent.TOGGLE`. Кроме того, для демонстрационных целей класс `SomeApp` программным путем включает переключатель, вызывая событие `ToggleEvent.TOGGLE`.

```
package {
  import flash.display.*;

  // Простое приложение, демонстрирующее применение
  // пользовательского события ToggleEvent.TOGGLE
  public class SomeApp extends Sprite {
    // Конструктор
    public function SomeApp ( ) {
      // Создание объекта ToggleSwitch
      var toggleSwitch:ToggleSwitch = new ToggleSwitch( );
      // Регистрация приемника для событий ToggleEvent.TOGGLE
      toggleSwitch.addEventListener(ToggleEvent.TOGGLE,
                                    toggleListener);

      // Изменяем состояние переключателя (обычно состояние
      // изменяется пользователем, но для демонстрационных целей
      // мы изменяем состояние программным путем)
      toggleSwitch.toggle( );
    }

    // Приемник выполняется каждый раз при возникновении события
    // ToggleEvent.TOGGLE
    private function toggleListener (e:ToggleEvent):void {
      if (e.isOn) {
        trace("The ToggleSwitch is now on.");
      } else {
        trace("The ToggleSwitch is now off.");
      }
    }
  }
}
```

Теперь, когда мы приобрели опыт создания пользовательских событий, рассмотрим более сложный сценарий: пользовательское событие со стандартным поведением.

Отмена стандартного поведения для пользовательских событий

В предыдущем разделе рассказывалось, что некоторые предопределенные события характеризуются стандартным поведением. Например, стандартное поведение события `TextEvent.TEXT_INPUT` заключается в добавлении текста в текстовое поле. Мы также знаем, что для предопределенных событий, которые относятся к категории отменяемых, избежать стандартного поведения можно с помощью метода экземпляра `preventDefault( )` класса `Event`.

Кроме того, пользовательские события могут характеризоваться определенным стандартным поведением, избежать которого можно также с помощью метода `preventDefault( )`. Стандартное поведение пользовательского события полностью определяется и реализуется в программе. Общий подход, применяемый для создания событий с отменяемым стандартным поведением, заключается в следующем.

1. На этапе диспетчеризации события создать событийный объект, представляющий событие, передав в качестве параметра *отменяемое* конструктора класса `Event` значение `true`.
2. Использовать метод `dispatchEvent( )` для диспетчеризации события.
3. После завершения метода `dispatchEvent( )` использовать метод экземпляра `isDefaultPrevented( )` класса `Event`, чтобы определить, запрашивали ли приемники отмену стандартного поведения.
4. Если метод `isDefaultPrevented( )` событийного объекта вернет значение `false`, продолжать выполнение действий, относящихся к стандартному поведению; в противном случае не выполнять действий, относящихся к стандартному поведению.

Рассмотрим обобщенный код для создания события с отменяемым стандартным поведением:

```
// Создание событийного объекта с произвольными значениями для параметров
// тип и всплывающее. Для параметра отменяемое (третий параметр) указывается
// значение true.
var e:Event = new Event(тип, всплывающее, true);

// Диспетчеризация события
dispatchEvent(e);

// Проверить, запрашивали ли приемники отмену стандартного поведения.
// Если приемники не вызывали метод preventDefault( ),...
if (!e.isDefaultPrevented( )) {
  // ...выполняем действия, относящиеся к стандартному поведению
}
```

Рассмотрим описанные шаги на примере с кнопкой-переключателем. Предположим, мы используем нашу кнопку-переключатель в приложении с панелью управления, которое назначает различные привилегии своим пользователям в зависимости от их статуса. Пользователи-«гости» могут использовать только некоторые

кнопки на панели, в то время как пользователям-«администраторам» доступны все кнопки.

Для реализации различных уровней доступа в приложении мы определяем новый тип события кнопки-переключателя: `ToggleEvent.TOGGLE_ATTEMPT`. Это событие возникает всякий раз, когда пользователь пытается включить или выключить кнопку-переключатель. Стандартным поведением, характеризующим событие `ToggleEvent.TOGGLE_ATTEMPT`, является изменение состояния переключателя.

Для упрощения будем считать, что кнопку-переключатель можно включить или выключить только щелчком кнопки мыши (не используя клавиатуру). Всякий раз, когда пользователь щелкает на кнопке-переключателе, выполняется диспетчеризация события `ToggleEvent.TOGGLE_ATTEMPT`. Затем, если никакой приемник не отменяет стандартного поведения, мы изменяем состояние переключателя. Рассмотрим соответствующий код:

```
private function clickListener (e:MouseEvent):void {
  // Пользователь попытался включить или выключить переключатель, поэтому
  // просим среду Flash выполнить диспетчеризацию события
  // ToggleEvent.TOGGLE_ATTEMPT, получателем которого является данный объект
  // ToggleSwitch. Сначала создадим событийный объект...
  var toggleEvent:ToggleEvent =
                          new ToggleEvent(ToggleEvent.TOGGLE_ATTEMPT,
                                          true,
                                          true);
  // ... затем отправляем запрос на диспетчеризацию события
  dispatchEvent(toggleEvent);

  // Диспетчеризация события ToggleEvent.TOGGLE_ATTEMPT завершена.
  // Если никакой приемник не отменил стандартное поведение события...
  if (!toggleEvent.isDefaultPrevented( )) {
    // ...изменяем состояние переключателя
    toggle( );
  }
}
```

В нашем приложении с панелью управления мы регистрируем приемник события `ToggleEvent.TOGGLE_ATTEMPT` для каждого объекта `ToggleSwitch`. Внутри этого приемника проверяется статус пользователя. Для закрытых переключателей, если пользователь является «гостем», мы отменяем стандартное поведение события. Рассмотрим этот код:

```
// Приемник выполняется всякий раз, когда возникает событие
// ToggleEvent.TOGGLE_ATTEMPT
private function toggleAttemptListener (e:ToggleEvent):void {
  // Если пользователь является «гостем»...
  if (userType == UserType.GUEST) {
    // ...запретить изменение состояния переключателя
    e.preventDefault( );
  }
}
```

В листинге 12.5 целиком показан код приложения с панелью управления, включая полнофункциональную, хотя и простую графическую версию кнопки-переключателя. Понять этот код вам помогут подробные комментарии.

Листинг 12.5. Классы приложения с панелью управления

```
// Класс ToggleEvent
package {
  import flash.events.*;

  // Класс, представляющий пользовательское событие "toggle"
  public class ToggleEvent extends Event {
    // Константа для типа события "toggle"
    public static const TOGGLE:String = "toggle";

    // Константа для типа события "toggleAttempt"
    public static const TOGGLE_ATTEMPT:String = "toggleAttempt";

    // Обозначает текущее состояние переключателя –
    // включен или выключен
    public var isOn:Boolean;

    // Конструктор
    public function ToggleEvent (type:String,
                                 bubbles:Boolean = false,
                                 cancelable:Boolean = false,
                                 isOn:Boolean = false) {
      // Передаем параметры конструктора в конструктор суперкласса
      super(type, bubbles, cancelable);

      // Запоминаем состояние переключателя, которое может быть использовано
      // в приемниках события ToggleEvent.TOGGLE
      this.isOn = isOn;
    }

    // Любой класс пользовательского события должен переопределить
    // метод clone( )
    public override function clone( ):Event {
      return new ToggleEvent(type, bubbles, cancelable, isOn);
    }

    // Любой класс пользовательского события должен переопределить
    // метод toString( )
    public override function toString( ):String {
      return formatToString("ToggleEvent", "type", "bubbles",
                            "cancelable", "eventPhase", "isOn");
    }
  }
}

// Класс ToggleSwitch
package {
```

```
import flash.display.*;
import flash.events.*;

// Представляет простую кнопку-переключатель со стандартным поведением
public class ToggleSwitch extends Sprite {
  // Запоминает состояние переключателя
  private var isOn:Boolean;
  // Содержит графическое изображение кнопки-переключателя
  private var icon:Sprite;

  // Конструктор
  public function ToggleSwitch ( ) {
    // Создаем объект Sprite, который будет содержать графическое
    // изображение кнопки-переключателя
    icon = new Sprite( );
    addChild(icon);

    // По умолчанию переключатель выключен
    isOn = false;
    drawOffState( );

    // Регистрируем приемник для получения уведомлений всякий раз,
    // Если пользователь щелкнет кнопкой мыши
    // на графическом изображении переключателя
    icon.addEventListener(MouseEvent.CLICK, clickListener);
  }

  // Приемник, выполняемый после щелчка кнопкой мыши
  // на кнопке-переключателе
  private function clickListener (e:MouseEvent):void {
    // Пользователь попытался включить или выключить переключатель,
    // поэтому просим среду Flash выполнить диспетчеризацию
    // события ToggleEvent.TOGGLE_ATTEMPT, получателем которого
    // является данный объект ToggleSwitch. Сначала создадим
    // событийный объект...
    var toggleEvent:ToggleEvent =
                              new ToggleEvent(ToggleEvent.TOGGLE_ATTEMPT,
                                              true, true);
    // ...затем отправляем запрос на диспетчеризацию события
    dispatchEvent(toggleEvent);

    // Диспетчеризация события ToggleEvent.TOGGLE_ATTEMPT завершена.
    // Если никакой приемник не отменил стандартное поведение события...
    if (!toggleEvent.isDefaultPrevented( )) {
      // ...изменяем состояние переключателя
      toggle( );
    }
  }

  // Включает переключатель, если в настоящий момент он выключен, или
  // выключает его, если переключатель включен. Стоит отметить, что состояние
```

```
    // переключателя может быть изменено программным путем, даже если пользователь
    // не имеет привилегий изменить состояние переключателя вручную.
    public function toggle ( ):void {
      // Изменяем состояние переключателя
      isOn = !isOn;

      // Рисуем для нового состояния переключателя соответствующее изображение
      if (isOn) {
        drawOnState( );
      } else {
        drawOffState( );
      }

      // Просим среду Flash выполнить диспетчеризацию события
      // ToggleEvent.TOGGLE, получателем которого является данный
      // объект ToggleSwitch
      var toggleEvent:ToggleEvent = new ToggleEvent(ToggleEvent.TOGGLE,
                                                    true, false, isOn);
      dispatchEvent(toggleEvent);
    }

    // Рисуем изображение для выключенного состояния
    private function drawOffState ( ):void {
      icon.graphics.clear( );
      icon.graphics.lineStyle(1);
      icon.graphics.beginFill(0xFFFFFF);
      icon.graphics.drawRect(0, 0, 20, 20);
    }

    // Рисуем изображение для включенного состояния
    private function drawOnState ( ):void {
      icon.graphics.clear( );
      icon.graphics.lineStyle(1);
      icon.graphics.beginFill(0xFFFFFF);
      icon.graphics.drawRect(0, 0, 20, 20);
      icon.graphics.beginFill(0x000000);
      icon.graphics.drawRect(5, 5, 10, 10);
    }
  }
}

// Класс ControlPanel (основной класс приложения)
package {
  import flash.display.*;

  // Базовое приложение, которое демонстрирует отмену
  // стандартного поведения для пользовательских событий
  public class ControlPanel extends Sprite {
    // Присваиваем уровень привилегий для пользователя данного приложения.
    // В данном примере только пользователи с привилегиями UserType.ADMIN
    // могут использовать кнопку-переключатель.
    private var userType:int = UserType.GUEST;
```

```
    // Конструктор
    public function ControlPanel ( ) {
      // Создаем объект ToggleSwitch
      var toggleSwitch:ToggleSwitch = new ToggleSwitch( );
      // Регистрируем приемник для событий
      // ToggleEvent.TOGGLE_ATTEMPT
      toggleSwitch.addEventListener(ToggleEvent.TOGGLE_ATTEMPT,
                                    toggleAttemptListener);

      // Регистрируем приемник для событий ToggleEvent.TOGGLE
      toggleSwitch.addEventListener(ToggleEvent.TOGGLE,
                                    toggleListener);
      // Добавляем кнопку-переключатель в иерархию отображения
      // данного объекта
      addChild(toggleSwitch);
    }

    // Приемник выполняется всякий раз, когда возникает событие
    // ToggleEvent.TOGGLE_ATTEMPT
    private function toggleAttemptListener (e:ToggleEvent):void {
      // Если пользователь является «гостем»...
      if (userType == UserType.GUEST) {
        // ...запретить изменение состояния переключателя
        e.preventDefault( );
      }
    }

    // Приемник выполняется всякий раз, когда возникает событие
    // ToggleEvent.TOGGLE
    private function toggleListener (e:ToggleEvent):void {
      if (e.isOn) {
        trace("The ToggleSwitch is now on.");
      } else {
        trace("The ToggleSwitch is now off.");
      }
    }
  }
}

// Класс UserType
package {
  // Определяет константы, представляющие уровни
  // пользовательских привилегий в приложении
  // с панелью управления
  public class UserType {
    public static const GUEST:int = 0;
    public static const ADMIN:int = 1;
  }
}
```

Теперь, когда мы познакомились с пользовательскими событиями в ActionScript, рассмотрим две последние темы, связанные с событиями.

Недостаток проверки типов в событийной модели языка ActionScript

Событийная модель ActionScript, основанная на приемниках, включает несколько различных участников: приемник события, объект, регистрирующий этот приемник, получатель события, событийный объект и имя события. Диспетчеризация определенного события (и его обработка) завершится успешно только в том случае, если все участники будут надлежащим образом взаимодействовать между собой. Для этого должны выполняться следующие основные условия.

- Должен существовать тип события, для которого регистрируется приемник.
- Должен существовать сам приемник.
- Приемник должен знать, как обрабатывать событийный объект, передаваемый в процессе диспетчеризации возникшего события.
- Объект, осуществляющий регистрацию приемника, должен поддерживать указанный тип события.

Когда приемник регистрируется в объекте для получения события, он вступает в соглашение, основанное на типах данных, которое гарантирует выполнение первых трех условий. Если это соглашение не выполняется, компилятор генерирует ошибку типа. Например, рассмотрим следующий код, описывающий и регистрирующий приемник, в котором умышленно допущены три нарушения (выделенные полужирным шрифтом) контракта приемника события:

```
urlLoader.addEventListener(Event.COMPLTE, completeListenr);

private function completeListener (e:MouseEvent):void {
  trace("Load complete");
}
```

В приведенном коде нарушения контракта приемника события заключаются в следующем.

- Константа `Event.COMPLTE` записана с ошибкой: пропущена буква E. Компилятор сгенерирует ошибку, которая предупредит программиста о том, что тип события, для получения которого пытается зарегистрироваться приемник, не существует.
- Название приемника события записано с ошибкой: пропущена еще одна буква e. Компилятор сгенерирует ошибку, которая предупредит программиста о том, что регистрируемый приемник не существует.
- Типом данных первого параметра, передаваемого в метод `completeListener( )`, является `MouseEvent`, который не соответствует типу данных событийного объекта для события `Event.COMPLETE`. На этапе диспетчеризации события среда выполнения Flash сгенерирует ошибку, которая предупредит программиста, что приемник не может обработать переданный в него событийный объект.

Если мы изменим предыдущий код, исправив все указанные ошибки типа, диспетчеризация события и его обработка будут успешно выполнены.

Соглашение между приемником события и объектом, который регистрирует этот приемник, основанное на типах данных, позволяет гарантировать корректное выполнение нашего кода, обрабатывающего события.

Тем не менее соглашение между приемником и объектом, регистрирующим этот приемник, имеет один недостаток: оно не гарантирует, что объект поддерживает указанный тип события. Например, рассмотрим следующим код, который регистрирует приемник в объекте `urlLoader` для событий `TextEvent.TEXT_INPUT`:

```
urlLoader.addEventListener(TextEvent.TEXT_INPUT, textInputListener);
```

Хотя с практической точки зрения у нас есть все основания полагать, что объект `URLLoader` никогда не будет получателем события `TextEvent.TEXT_INPUT`, приведенный код не вызовет ошибки. В языке ActionScript регистрация приемников для событий осуществляется по любому имени. Например, следующий бессмысленный код также является допустимым:

```
urlLoader.addEventListener("dlrognw", dlrognwListener);
```

Каким бы очевидным ни казалось то, что объект `urlLoader` никогда не будет получателем события с именем `"dlrognw"`, программа на самом деле может выполнить диспетчеризацию такого события. Это демонстрирует следующий код:

```
urlLoader.dispatchEvent(new Event("dlrognw"));
```

Учитывая, что программа имеет возможность назначать любой объект в качестве получателя любого события, язык ActionScript намеренно не использует концепцию «поддерживаемых событий». Подобная гибкость является предметом споров, поскольку она может привести к трудновыявляемым ошибкам в коде. Например, предположим, что мы используем класс `Loader` для загрузки внешнего изображения, как показано в следующем коде:

```
var loader:Loader = new Loader( );
loader.load(new URLRequest("image.jpg"));
```

Предположим также, что объект `loader` будет являться получателем событий, информирующих о завершении загрузки изображения (во многом аналогично тому, как объект `URLLoader` является получателем событий о завершении загрузки элемента). Таким образом, мы пытаемся обработать событие `Event.COMPLETE` для нашего загружаемого элемента, зарегистрировав приемник непосредственно в объекте `Loader`, как показано в следующем коде:

```
loader.addEventListener(Event.COMPLETE, completeListener);
```

Запустив код, мы будем удивлены, поскольку, даже несмотря на отсутствие ошибок, метод `completeListener( )` не был вызван ни разу. Так как никаких ошибок не возникло, мы не сможем сразу же определить источник проблемы в нашем коде. Последующий анализ и отладка кода будут стоить нам времени и, по всей вероятности, значительного неудовлетворения. Только прочитав соответствующий раздел документации корпорации Adobe, мы определим проблему: объекты `Loader` на самом деле *не* являются получателями событий о завершении загрузки; вместо этого события о завершении загрузки должны обрабатываться

экземпляром класса `LoaderInfo` каждого объекта `Loader`, как показано в следующем коде:

```
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, completeListener);
```

В будущем язык ActionScript, возможно, позволит классам перечислять события, которые они поддерживают, и генерировать соответствующие предупреждения компилятора на попытки зарегистрировать приемник на события, которые не поддерживаются данным классом. Пока же классы, которые реализуют пользовательские события, путем переопределения метода `addEventListener( )` могут самостоятельно генерировать пользовательские ошибки в тех случаях, когда программа пытается зарегистрировать приемник для неподдерживаемых событий, как показано в следующем коде:

```
public override function addEventListener(eventType:String,
                                          handler:Function,
                                          capture:Boolean = false,
                                          priority:int = 0,
                                          weakRef:Boolean = false):void {
  // Метод canDispatchEvent( ) (не показан) проверяет наличие
  // указанного типа eventType в списке поддерживаемых
  // данным классом событий и возвращает значение типа Boolean,
  // которое говорит о том, является ли указанный тип eventType
  // поддерживаемым типом событий
  if(canDispatchEvent(eventType)) {
    // Событие поддерживается, поэтому приступаем к регистрации
    super.addEventListener(eventType, handler, capture, priority, weakRef);
  } else {
    // Событие не поддерживается, поэтому генерируем ошибку
    throw new Error(this + " does not support events of type '"
                    + eventType + "'" );
  }
}
```

Мораль этой истории такова: будьте особенно внимательны при регистрации приемника для события. Всегда убеждайтесь в том, что объект, в котором регистрируется приемник, на самом деле поддерживает требуемое событие.

Теперь рассмотрим последний вопрос, касающийся событийной модели: обработку событий в приложениях, состоящих из нескольких SWF-файлов, которые размещены в различных интернет-доменах. Для изучения следующего раздела необходимо иметь общее представление о методах загрузки SWF-файлов, которые рассматриваются в гл. 28.

Обработка событий между границами зон безопасности

В гл. 19 будет рассмотрено несколько сценариев, в которых ограничения безопасности не позволяют одному SWF-файлу осуществлять *кросс-скриптинг* (управлять программным путем) над другим файлом. Когда два SWF-файла не могут

осуществлять кросс-скриптинг друг над другом из-за ограничений безопасности приложения Flash Player, к ним применяются следующие ограничения, связанные с обработкой событий.

- Приемники событий из одного SWF-файла не могут регистрироваться для событий в объектах другого SWF-файла.
- Если получателем события является объект в иерархии отображения, любые объекты, недоступные в SWF-файле объекта получателя, не включаются в цепочку диспетчеризации событий.

К счастью, описанные ограничения можно полностью обойти с помощью статического метода `allowDomain( )` класса `flash.system.Security`. Рассмотрим два примера, которые демонстрируют применение метода `allowDomain( )` для обхода каждого из этих ограничений.

Информацию по загрузке SWF-файлов можно найти в гл. 28.

Приемник из файла Module.swf регистрируется в объекте файла Main.swf

Предположим, что SWF-файл, находящийся на одном сайте (site-a.com/Main.swf), загружает SWF-файл, расположенный на другом сайте (site-b.com/Module.swf). Предположим также, что в файле `Module.swf` определен приемник, который желает зарегистрироваться в объекте, созданном в файле `Main.swf`. Чтобы разрешить данную регистрацию, перед регистрацией приемника из файла `Module.swf` в файле `Main.swf` должна быть выполнена следующая строка кода:

```
Security.allowDomain("site-b.com");
```

Эта строка позволяет всем SWF-файлам, находящимся на сайте site-b.com (включая файл `Module.swf`), регистрировать приемники в любом объекте, созданном в файле `Main.swf`.

Приемник из файла Main.swf получает уведомление о событии, получателем которого является отображаемый объект в файле Module.swf

Продолжая рассматривать сценарий, в котором файл `Main.swf` загружает файл `Module.swf` из предыдущего раздела, предположим, что экземпляр основного класса файла `Main.swf` добавляет объект класса `Loader`, содержащий файл `Module.swf`, в свою иерархию отображения, как показано в следующем коде:

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;

  public class Main extends Sprite {
    private var loader:Loader;
```

```
    public function Main( ) {
      loader = new Loader( );
      loader.load(new URLRequest("http://site-b.com/Module.swf"));
      // Добавляем объект Loader, содержащий файл Module.swf, в иерархию
      // отображения данного объекта
      addChild(loader);
    }
  }
}
```

Предположим также, что экземпляр основного класса файла `Main.swf` желает получать уведомления всякий раз, когда пользователь щелкает кнопкой мыши на объекте из файла `Module.swf`. Следовательно, экземпляр основного класса файла `Main.swf` регистрирует приемник в объекте `loader` для событий `MouseEvent.CLICK`, как показано в следующем коде:

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;

  public class Main extends Sprite {
    private var loader:Loader;

    public function Main( ) {
      loader = new Loader( );
      loader.load(new URLRequest("http://site-b.com/Module.swf"));
      addChild(loader);
      loader.addEventListener(MouseEvent.CLICK, clickListener);
    }

    private function clickListener (e:MouseEvent):void {
      trace("Module.swf was clicked");
    }
  }
}
```

Тем не менее, поскольку файлы `Main.swf` и `Module.swf` размещены в различных интернет-доменах, ограничения безопасности запрещают вызывать метод `clickListener( )` для возникающих событий `MouseEvent.CLICK`, получателями которых являются отображаемые потомки объекта `loader` (то есть отображаемые объекты из файла `Module.swf`).

Для того чтобы обойти данное ограничение, конструктор основного класса файла `Module.swf` содержит следующую строку кода:

```
Security.allowDomain("site-a.com");
```

После выполнения этой строки файл `Module.swf` начнет доверять файлу `Main.swf` (и всем SWF-файлам с сайта site-a.com), благодаря чему экземпляр основного класса файла `Main.swf` будет включен клиентской средой выполнения Flash в цепочку диспетчеризации события `MouseEvent.CLICK`, получателем которого являются объекты из файла `Module.swf`. В результате метод `clickListener( )`

будет вызываться всякий раз при щелчке кнопкой мыши на объекте из файла `Module.swf`.

Подробную информацию о методе allowDomain( ) и безопасности приложения Flash Player можно найти в разд. «Разрешения создателя (allowDomain( ))» гл. 19.

Стоит отметить, что вызов метода `allowDomain( )` позволяет не только обрабатывать события между границами зон безопасности: все SWF-файлы из разрешенного домена получают возможность осуществлять кросс-скриптинг над SWF-файлом, в котором был вызван метод `allowDomain( )`. Однако существует альтернатива всеобъемлющим разрешениям, выдаваемым методом `allowDomain( )`.

Альтернатива методу allowDomain( ): разделяемые события

В некоторых случаях SWF-файлы из различных доменов могут пожелать совместно использовать события, не предоставляя при этом всех полномочий для кросс-скриптинга. Для решения подобных проблем приложение Flash Player предоставляет переменную экземпляра `sharedEvents` класса `LoaderInfo`. Переменная `sharedEvents` — это простой нейтральный объект, через который два SWF-файла могут отправлять события друг другу, независимо от ограничений, обусловленных требованиями безопасности. Этот подход позволяет осуществлять взаимодействие между SWF-файлами, основанное на событиях, не отменяя требований безопасности, но для его реализации требуется написание большего объема кода, чем при использовании альтернативного подхода с методом `allowDomain( )`.

Рассмотрим применение переменной `sharedEvents` на примере. Предположим, что Томми основал компанию по производству фейерверков и создал рекламный сайт www.blast.ca с использованием технологии Flash. Томми нанял подрядчика Дерека для создания отдельного элемента, реализующего эффект, который заключается в хаотичной генерации анимированных взрывов фейерверков под указателем мыши. Дерек создает SWF-файл `MouseEffect.swf`, в котором реализован этот эффект, и размещает его по адресу www.dereksflasheffects.com/MouseEffect.swf. Дерек говорит Томми загрузить файл `MouseEffect.swf` в его приложение www.blast.ca/BlastSite.swf. Дерек и Томми согласились, что файл `MouseEffect.swf` должен размещаться на сервере www.derekflasheffects.com, что в дальнейшем позволит Дереку легко обновлять данный файл, не внося при этом никаких изменений в сайт Томми.

Томми просит Дерека изменить файл `MouseEffect.swf` таким образом, чтобы генерация взрывов прекращалась в тот момент, когда указатель мыши покидает область отображения приложения Flash Player. Дерек считает эту идею целесообразной и приступает к написанию соответствующего кода. В обычной ситуации, чтобы определить выход указателя мыши за пределы области отображения приложения Flash Player, код в файле `MouseEffect.swf` должен зарегистрировать приемник в экземпляре `Stage` для событий `Event.MOUSE_LEAVE`. Однако, поскольку файлы `MouseEffect.swf` и `BlastSite.swf` размещены в разных доменах, файл `MouseEffect.swf` не имеет доступа к экземпляру `Stage`. Томми решает, что

вместо того, чтобы предоставлять файлу `MouseEffect.swf` полный доступ к файлу `BlastSite.swf`, он просто переадресует все события `Event.MOUSE_LEAVE` файлу `MouseEffect.swf` через переменную `sharedEvents`.

Листинг 12.6 демонстрирует код файла `BlastSite.swf`, относящийся к переадресации событий.

Листинг 12.6. Переадресация события через переменную sharedEvents

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;
  import flash.system.*;

  public class BlastSite extends Sprite {
    private var loader:Loader;

    public function BlastSite ( ) {
      // Загружаем файл MouseEffect.swf
      loader = new Loader( );
      loader.load(
       new URLRequest("http://www.dereksflasheffects.com/MouseEffect.swf"));
      addChild(loader);

      // Регистрируем приемник для событий Event.MOUSE_LEAVE
      stage.addEventListener(Event.MOUSE_LEAVE, mouseLeaveListener);
    }

    // Когда возникает событие Event.MOUSE_LEAVE,...
    private function mouseLeaveListener (e:Event):void {
      // ...переадресуем его файлу MouseEffect.swf
      loader.contentLoaderInfo.sharedEvents.dispatchEvent(e);
    }
  }
}
```

Листинг 12.7 демонстрирует код файла `MouseEffect.swf`, относящийся к обработке событий.

Листинг 12.7. Обработка события, полученного через переменную sharedEvents

```
package {
  import flash.display.Sprite;
  import flash.events.*;

  public class MouseEffect extends Sprite {
    public function MouseEffect ( ) {
      // Регистрируем приемник для событий Event.MOUSE_LEAVE, получаемых
      // через переменную sharedEvents
      loaderInfo.sharedEvents.addEventListener(Event.MOUSE_LEAVE,
                                               mouseLeaveListener);
    }
```

```
    // Обрабатываем события Event.MOUSE_LEAVE, полученные через переменную
    // sharedEvents
    private function mouseLeaveListener (e:Event):void {
      trace("MouseEffect.mouseLeaveListener( ) was invoked...");
      // Здесь прекращаем генерацию взрывов...
    }
  }
}
```

Дерек получил деньги за свою работу и отложил их на предстоящую поездку в Японию. Томми доволен эффектом взрывов, хотя и не уверен, что он способствовал увеличению продаж.

Что дальше?

Мы достигли больших успехов в изучении основ языка ActionScript. Если вы поняли концепцию прочитанных 12 глав, то теперь обладаете достаточными знаниями языка ActionScript, чтобы приступить к рассмотрению большей части API клиентской среды выполнения Flash. Таким образом, пришло время сделать собственный выбор. Если вы желаете продолжить знакомство с базовыми возможностями ActionScript, переходите к чтению гл. 13, в которой будет рассказано, как создавать код, позволяющий выходить из сбойных ситуаций на этапе выполнения программы. Если, с другой стороны, вы предпочитаете узнать, как использовать язык ActionScript для отображения содержимого на экране, сразу переходите к части II.

Обработка исключений и ошибок

В этой главе мы познакомимся с системой языка ActionScript, предназначенной для генерации и реагирования на ошибки этапа выполнения программы, которые называются *исключениями*. В языке ActionScript ошибки могут генерироваться как средой Flash, так и выполняемой программой. Ошибки, генерируемые средой Flash, называются *предопределенными*; ошибки, генерируемые программой, называются *пользовательскими*. В программе можно реагировать на любые ошибки (как предопределенные, так и пользовательские), или, иначе говоря, *обрабатывать* их с помощью инструкции `try/catch/finally`. Генерация ошибок осуществляется с помощью инструкции `throw`.

Чтобы познакомиться с процессом генерации и обработки ошибок в программе, мы вернемся к нашей программе по созданию виртуального зоопарка.

Изменения, которые будут внесены в программу «Зоопарк» на протяжении этой главы, являются последними для этой программы до конца книги. Чтобы завершить программу по созданию виртуального зоопарка, мы должны рассмотреть вопросы, связанные с экранным программированием и работой с мышью, которые освещены в части II. Прочитав часть II, обратитесь к приложению, чтобы узнать, как добавить графику и интерактивность в программу «Зоопарк».

Механизм обработки исключений

Если помните, класс `VirtualPet` определяет метод `setName( )`, который присваивает значение переменной `petName` экземпляров класса `VirtualPet`. Чтобы освежить вашу память, ниже представлен соответствующий код класса `VirtualPet` (те части класса, которые не имеют отношения к присваиванию значения переменной `petName`, не приводятся):

```
public class VirtualPet {
  private var petName:String;

  public function setName (newName:String):void {
    // Если длина заданного нового имени больше maxNameLength символов...
    if (newName.length > VirtualPet.maxNameLength) {
      // ...обрезать имя
      newName = newName.substr(0, VirtualPet.maxNameLength);
    } else if (newName == "") {
      // ...в противном случае, если заданное новое имя является
      // пустой строкой, завершить выполнение метода, не изменяя
      // значения переменной petName
```

```
      return;
    }

    // Присвоить новое проверенное имя переменной petName
    petName = newName;
  }
}
```

Метод `setName( )` перед тем, как изменить значение переменной `petName`, проверяет, является ли допустимым количество символов в новом имени животного. Если новое имя животного не является допустимым, значение переменной не изменяется; в противном случае изменение значения допускается.

Изменим метод `setName( )` таким образом, чтобы он генерировал исключение (оповещал об ошибке) в тех случаях, когда передаваемое значение параметра `newName` содержит недопустимое количество символов. Позднее мы добавим код, восстанавливающий работоспособность программы после ошибки, для обработки нового исключения, генерируемого в методе `setName( )`.

Для генерации исключения в нашем коде мы используем оператор `throw`, который имеет следующий вид:

`throw` *выражение*

В предыдущем коде *выражение* — это значение данных, описывающее некоторую необычную или проблематичную ситуацию. Использование оператора `throw` для оповещения об ошибке иногда называется «генерацией исключения». Язык ActionScript позволяет использовать любое значение в качестве выражения *выражение* в операторе `throw`. Например, значением выражения *выражение* может быть строковый литерал `"Something went wrong!"` (Что-то не получилось!) или числовой код ошибки. Однако корпорация Adobe рекомендует использовать в качестве значения выражения *выражение* экземпляр предопределенного класса `Error` (или одного из его подклассов), считая этот подход хорошей практикой. Класс `Error` является стандартным классом, представляющим исключительные ситуации в программе. Его переменная экземпляра `message` используется для описания ошибки.

Оператор `throw` останавливает выполнение кода и передает значение выражения *выражение* в специальный блок кода, называемый блоком `catch`, который будет реагировать на возникшую проблему, или *обрабатывать* ее. Прежде чем рассмотреть, как работает блок `catch`, изменим метод `setName( )` таким образом, чтобы он генерировал исключение с помощью оператора `throw`, когда получено недопустимое значение параметра `petName`:

```
public function setName (newName:String):void {
  // Если длина заданного нового имени больше maxNameLength символов...
  if (newName.length > VirtualPet.maxNameLength || newName == "") {
    // ...генерируем ошибку
    throw new Error("Invalid pet name specified.");
  }

  // Присвоить новое допустимое имя переменной petName
  petName = newName;
}
```

В нашей новой версии метода `setName( )`, если значение параметра `newName` является недопустимым, мы используем оператор `throw` для прекращения выполнения метода вместо того, чтобы просто обрезать указанное имя, как мы делали раньше. Кроме того, мы указываем описание проблемы — `"Invalid pet name specified"` (Указано недопустимое имя животного) — в качестве аргумента конструктора `Error`. Это описание определяет ситуацию, из-за которой возникла ошибка. Конструктор `Error` присваивает это описание переменной `message` созданного объекта `Error`.

Если метод `setName( )` не обнаружит никаких проблем со значением параметра `newName`, то он завершится нормально, и код, вызвавший его, может быть уверен, что работа, возложенная на этот метод, выполнена успешно. В противном случае блок `catch` должен обработать возникшую проблему. Блок `catch` является частью большой инструкции, называемой инструкцией `try/catch/finally`. Инструкция `try/catch/finally` предусматривает план восстановления для кода, который может сгенерировать исключение. Вот общая структура типовой инструкции `try/catch/finally`:

```
try {
  // Код в этом блоке может генерировать исключения
} catch (e:тип) {
  // Код в этом блоке обрабатывает возникшую проблему
} finally {
  // Код в этом блоке выполняется всегда, независимо от того,
  // сгенерировал блок try исключение или нет
}
```

В приведенном коде ключевое слово `try` сообщает среде Flash, что мы собираемся выполнить код, который может сгенерировать исключение. Блок `catch` обрабатывает исключения, генерируемые блоком `try`. Код в блоке `catch` выполняется в том, и только в том случае, когда код в блоке `try` сгенерировал исключение. Код в блоке `finally` выполняется всегда после завершения выполнения блока `try` или `catch`. Блок `finally` инструкции `try/catch/finally` обычно содержит очищающий код, который должен выполняться независимо от того, было сгенерировано исключение в соответствующем блоке `try` или нет.

Обратите внимание на типовую структуру.

1. Блок `try` выполняет код, который может сгенерировать исключение.
2. Код в блоке `try` использует оператор `throw` для оповещения о любых ошибках.
3. Если в блоке `try` не возникло никаких ошибок, то он выполняется полностью и программа пропускает блок `catch`.
4. Если в блоке `try` была сгенерирована ошибка, то его выполнение прекращается и начинается выполнение блока `catch`. Блок `catch` способен обрабатывать любые ошибки, возникающие в блоке `try`.
5. Выполняется блок `finally`.

В большинстве случаев блок `finally` не требуется и, следовательно, опускается. В последующих примерах мы будем опускать блок `finally`. Далее, в разд. «Блок finally», мы рассмотрим пример использования этого блока.

Когда выполняется блок `catch`, он получает значение выражения *выражение* оператора `throw` в качестве параметра. В блоке `catch` это значение может помочь выявить ошибку, сгенерированную в блоке `try`. Образно говоря, код, в котором возникла проблема, *бросает* (throw) исключение (передает объект `Error`) в блок `catch`, который получает этот объект в качестве параметра (*ловит* (catch) его).

Далее в разд. «Передача исключений вверх по иерархии объектов» мы выясним, что произойдет в том случае, если возникшая ошибка не будет обработана.

Рассмотрим пример инструкции `try/catch/finally`:

```
try {
  somePet.setName("James");
  // Если мы находимся здесь, значит, исключение не возникло;
  // продолжаем выполнение, как планировалось ранее.
  trace("Pet name set successfully.");
} catch (e:Error) {
  // ОШИБКА! Недопустимые данные. Выводим предупреждение.
  trace("An error occurred: " + e.message);
}
```

Если при вызове метода `pet.setName( )` внутри предыдущего блока `try` оператор `throw` метода `setName( )` не будет выполнен (если не произойдет никакой ошибки), то все последующие инструкции в блоке `try` будут выполнены успешно и программа полностью пропустит блок `catch`. Однако если метод `setName( )` сгенерирует исключение, программа немедленно прекратит выполнение инструкций в блоке `try` и перейдет к выполнению блока `catch`. В блоке `catch` значением параметра `e` является объект класса `Error`, переданный в оператор `throw` внутри метода `setName( )`.

В предыдущем примере код в блоке `catch` при отладке просто отображает значение переменной `message` объекта `Error`. Однако в более сложном приложении блок `catch` может попытаться восстановить работоспособность программы после ошибки, возможно отобразив окно, которое позволит пользователю указать допустимое имя.

Обработка нескольких типов исключений

Пример исключения из предыдущего раздела был чрезмерно упрощен. Что произойдет, если наш метод генерирует ошибки нескольких типов? Все ошибки будут отправлены в один и тот же блок `catch`? Что ж, это зависит от разработчика. Конечно, они все могут быть отправлены в один блок `catch`, однако чаще всего обработка различных типов ошибок выполняется отдельными блоками `catch` — и это является хорошей практикой. Рассмотрим почему.

Предположим, что мы хотим получить набор более детальных сообщений об ошибках в нашем методе `setName( )`: одно сообщение для недопустимых данных, одно — для слишком короткого имени, еще одно — для слишком длинного имени.

Тело нашего модифицированного метода `setName( )` могло бы выглядеть следующим образом:

```
if (newName.indexOf(" ") == 0) {
  // Имена не могут начинаться с пробела...
  throw new Error("Invalid pet name specified.");
} else if (newName == "") {
  throw new Error("Pet name too short.");
} else if (newName.length > VirtualPet.maxNameLength) {
  throw new Error("Pet name too long.");
}
```

Чтобы обработать все три возможные сообщения об ошибках, генерируемые в нашем новом методе `setName( )`, мы могли бы записать код нашей инструкции `try/catch/finally` следующим образом:

```
try {
  somePet.setName("некоеИмяЖивотного");
  // Если мы находимся здесь, значит, исключение не возникло; продолжаем
  // выполнение, как планировалось ранее.
  trace("Pet name set successfully.");
} catch (e:Error) {
  switch (e.message) {
    case "Invalid pet name specified.":
    trace("An error occurred: " + e.message);
    trace("Please specify a valid name.");
    break;

    case "Pet name too short.":
    trace("An error occurred: " + e.message);
    trace("Please specify a longer name.");
    break;

    case "Pet name too long.":
    trace("An error occurred: " + e.message);
    trace("Please specify a shorter name.");
    break;
  }
}
```

Надо признаться, что этот код работает, однако он имеет множество недостатков. Самый первый и наиболее серьезный недостаток состоит в том, что ошибки отличаются друг от друга только текстом в строке, которая скрыта внутри класса `VirtualPet`. Всякий раз, когда мы хотим узнать, какие типы ошибок могут возникать в методе `setName( )`, мы вынуждены обращаться к коду класса `VirtualPet` и искать строки сообщений об ошибках. Использование сообщений для идентификации ошибок между различными методами и классами зачастую приводит к появлению ошибок, вызванных человеческим фактором, и затрудняет поддержку нашего кода. Второй недостаток заключается в том, что оператор `switch` сам по себе сложен для чтения. Это ненамного лучше использования, скажем, числовых кодов ошибок вместо формальных исключений.

К счастью, существует официальный (и элегантный) способ обработки нескольких типов исключений. Каждый блок `try` может иметь любое количество вспомогательных блоков `catch`. Когда исключение генерируется в блоке `try`, который имеет несколько блоков `catch`, среда Flash выполняет тот блок `catch`, тип данных параметра которого совпадает с типом данных значения сгенерированного исключения.

Рассмотрим общий синтаксис оператора `try` с несколькими блоками `catch`:

```
try {
  // Код, который может генерировать исключения.
} catch (e:ТипОшибки1) {
  // Код обработки ошибки с типом ТипОшибки1.
} catch (e:ТипОшибки2) {
  // Код обработки ошибки с типом ТипОшибки2.
} catch (e:ТипОшибкиn) {
  // Код обработки ошибки с типом ТипОшибкиn.
}
```

Если бы оператор `throw` в предыдущем блоке `try` сгенерировал исключение, применив в качестве параметра выражение типа *ТипОшибки1*, был бы выполнен первый блок `catch`. Например, следующий код приведет к выполнению первого блока `catch`:

```
throw new ТипОшибки1( );
```

Если бы в оператор `throw` было передано выражение типа *ТипОшибки2*, был бы выполнен второй блок `catch` и т. д. Как уже известно, в языке ActionScript выражение оператора `throw` может принадлежать любому типу данных. Однако помните, что в большинстве программ исключения представляются только экземплярами класса `Error` или одного из его подклассов и это является хорошей практикой.

Если мы хотим генерировать несколько типов исключений в приложении, для каждого типа исключения мы определяем отдельный подкласс класса `Error`. Вам, как разработчику, необходимо определить требуемую степень детализации (то есть указать, до какой степени обособлять различные исключительные ситуации).

Определение степени детализации типов исключений. Следует ли определять подкласс класса `Error` для каждой исключительной ситуации? Обычно ответ на этот вопрос отрицателен — вам не потребуется такая степень детализации, поскольку во многих случаях несколько исключительных ситуаций могут иметь одинаковый смысл. Если вам не нужно задавать различие между несколькими исключительными ситуациями, то можете объединить эти ситуации в одном пользовательском подклассе класса `Error`. Например, вы можете определить один подкласс класса `Error` с именем `InvalidInputException` для решения широкого круга проблем, связанных с вводом данных.

С другой стороны, вы должны определять отдельный подкласс класса `Error` для каждой исключительной ситуации, которая, по вашему мнению, отличается от других возможных ситуаций. Чтобы разобраться, когда следует создавать новый подкласс для конкретной исключительной ситуации, а также продемонстрировать возможность группирования нескольких ситуаций в одном подклассе, вернемся к методу `setName ( )`.

Ранее мы генерировали три исключения в методе `setName( )`. Все три исключения использовали базовый класс `Error`. Приведем этот код снова:

```
if (newName.indexOf(" ") == 0) {
  // Имена не могут начинаться с пробела...
  throw new Error("Invalid pet name specified.");
} else if (newName == "") {
  throw new Error("Pet name too short.");
} else if (newName.length > VirtualPet.maxNameLength) {
  throw new Error("Pet name too long.");
}
```

В данном коде, чтобы провести различие между исключениями класса `VirtualPet` и остальными исключениями в нашем приложении, мы использовали переменную `message` класса `Error`, которая, как уже известно, делает наши исключения неудобными для использования и может привести к появлению ошибок, вызванных человеческим фактором. Для отличия ошибок, относящихся к классу `VirtualPet`, от других ошибок в нашем приложении лучше определить пользовательский подкласс класса `Error` с именем `VirtualPetNameException`, как показано в следующем коде:

```
// Код в файле VirtualPetNameException.as:
package zoo {
  public class VirtualPetNameException extends Error {
    public function VirtualPetNameException ( ) {
      // Передаем сообщение об ошибке в конструктор класса Error, которое
      // будет присвоено переменной message данного объекта
      super("Invalid pet name specified.");
    }
  }
}
```

Теперь, когда у нас появился класс `VirtualPetNameException`, метод `setName( )` может генерировать свой собственный тип ошибки, как показано в следующем коде:

```
public function setName (newName:String):void {
  if (newName.indexOf(" ") == 0) {
    throw new VirtualPetNameException( );
  } else if (newName == "") {
    throw new VirtualPetNameException( );
  } else if (newName.length > VirtualPet.maxNameLength) {
    throw new VirtualPetNameException( );
  }

  petName = newName;
}
```

Обратите внимание, что в предыдущем описании метода для всех трех исключительных ситуаций, относящихся к классу `VirtualPet`, генерируется один и тот же тип ошибки (`VirtualPetNameException`). Как разработчики класса `VirtualPet` мы столкнулись с проблемой определения степени детализации исключительных ситуаций. Мы должны решить не только то, в какой мере сообщения об ошибках класса

VirtualPet будут отличаться от других ошибок приложения, но и то, насколько эти ошибки будут отличаться друг от друга. У нас есть следующие варианты:

Вариант 1. Использовать один класс для исключительных ситуаций класса VirtualPet.

В этом случае мы оставляем предыдущее описание метода setName( ) как есть. Как вскоре станет известно, этот вариант позволяет отличать ошибки класса VirtualPet от других базовых ошибок в программе, однако мы не сможем отличить между собой три внутренние разновидности ошибок класса VirtualPet (недопустимые данные, слишком короткое имя животного и слишком длинное имя животного).

Вариант 2. Упростить код, но по-прежнему использовать один класс для исключительных ситуаций класса VirtualPet.

В данном случае мы изменяем метод setName( ) таким образом, чтобы проверка всех трех исключительных ситуаций выполнялась в одном операторе if. Этот вариант аналогичен предыдущему, но использует более сжатый код.

Вариант 3. Использовать отладочные сообщения для нахождения различий между ошибками.

В данном случае мы добавляем конфигурируемые отладочные сообщения в класс VirtualPetNameException. Этот вариант незначительно увеличивает степень детализации по сравнению с двумя предыдущими, но только для удобства разработчика и только на этапе отладки.

Вариант 4. Создать пользовательский класс исключения для каждой исключительной ситуации.

Используя это вариант, мы создаем два пользовательских подкласса класса VirtualPetNameException: VirtualPetInsufficientDataException и VirtualPetExcessDataException. Этот вариант обеспечивает наибольшую степень детализации. Он позволяет программе независимо реагировать на три разновидности ошибок, относящихся к классу VirtualPet, используя формальную логику ветвлений.

Рассмотрим каждый из описанных вариантов.

Варианты 1 и 2. Использование одного пользовательского типа исключения. Первый вариант состоит в применении предыдущего описания метода setName( ), которое генерирует ошибку одного и того же типа (VirtualPetNameException) для всех трех исключительных ситуаций, относящихся к классу VirtualPet. Поскольку для генерации исключений этот метод использует класс VirtualPetNameException, а не класс Error, исключения класса VirtualPet уже отличаются от других базовых исключений. Пользователи метода setName( ) могут применять код, аналогичный следующему, для отличия ошибок, относящихся к классу VirtualPet, от других базовых ошибок:

```
try {
  // Этот вызов метода setName( ) приведет к возникновению исключения
  // VirtualPetNameException.
  somePet.setName("");
  // Другие инструкции в этом блоке try могут генерировать
  // другие базовые ошибки. Для демонстрационных целей
```

```
  // мы непосредственно сгенерируем
  // базовую ошибку.
  throw new Error("A generic error.");
} catch (e:VirtualPetNameException) {
  // Здесь обрабатываются ошибки, связанные с именем объекта класса
  // VirtualPet.
  trace("An error occurred: " + e.message);
  trace("Please specify a valid name.");
} catch (e:Error) {
  // Здесь обрабатываются все остальные ошибки.
  trace("An error occurred: " + e.message);
}
```

Для большого количества приложений степень детализации, обеспечиваемая классом `VirtualPetNameException`, оказывается достаточной. В этом случае мы должны по крайней мере переписать метод `setName ( )`, чтобы он не содержал избыточный код (трижды генерирующий исключение `VirtualPetNameException`). Рассмотрим переписанный код (представляющий вариант 2 из предыдущего списка):

```
public function setName (newName:String):void {
  if (newName.indexOf(" ") == 0
      || newName == ""
      || newName.length > VirtualPet.maxNameLength) {
    throw new VirtualPetNameException( );
  }

  petName = newName;
}
```

Переписывание кода с целью улучшения его структуры без изменения существующего поведения называется рефакторингом.

Вариант 3. Применение конфигурируемых отладочных сообщений. Вариант 3 заключается в добавлении конфигурируемых отладочных сообщений в класс `VirtualPetNameException`. Варианты 1 и 2 позволяют отличить исключение класса `VirtualPet` от других исключений в приложении, но не позволяют отличить исключение «слишком длинное» от исключения «слишком короткое». Если вы чувствуете, что отладка проблемы, связанной с именем объекта класса `VirtualPet`, затруднена отсутствием знания о том, является имя объекта класса `VirtualPet` слишком длинным или слишком коротким, можно изменить класс `VirtualPetNameException` таким образом, чтобы он принимал дополнительное описание (наподобие общеизвестного крестика на память). Рассмотрим измененный код класса `VirtualPetNameException`:

```
package zoo {
  public class VirtualPetNameException extends Error {
    // Предоставляет конструктор, который позволяет указывать
    // пользовательское сообщение. Если пользовательское сообщение не
    // указано, используется стандартное сообщение об ошибке
    public function VirtualPetNameException (
                          message:String = "Invalid pet name specified.") {
```

```
      super(message);
    }
  }
}
```

Чтобы воспользоваться модифицированным классом `VirtualPetNameException` в методе `setName( )`, вернемся к коду метода `setName( )`, использованному в варианте 1, и добавим отладочные сообщения об ошибке, как показано в следующем коде:

```
public function setName (newName:String):void {
  if (newName.indexOf(" ") == 0) {
    // В данном случае отлично подойдет стандартное сообщение об ошибке,
    // поэтому не стоит утруждать себя указанием пользовательского сообщения
    // об ошибке.
    throw new VirtualPetNameException( );
  } else if (newName == "") {
    // Вот пользовательское сообщение об ошибке «слишком короткое».
    throw new VirtualPetNameException("Pet name too short.");
  } else if (newName.length > VirtualPet.maxNameLength) {
    // Вот пользовательское сообщение об ошибке «слишком длинное».
    throw new VirtualPetNameException("Pet name too long.");
  }

  petName = newName;
}
```

Теперь, когда метод `setName( )` задает пользовательские сообщения об ошибках, упрощается отладка проблем, связанных с именем объекта класса `VirtualPet`, поскольку у нас появляется возможность получить больше информации о возникшей ошибке. Использование метода `setName( )` не изменилось, но теперь, если что-то пойдет не так, мы будем лучше проинформированы, как показано в следующем коде:

```
try {
  // Этот вызов метода setName( ) приведет к возникновению исключения
  // VirtualPetNameException.
  somePet.setName("");
} catch (e:VirtualPetNameException) {
  // Здесь обрабатываются ошибки, связанные с именем объекта класса
  // VirtualPet. В данном случае полезным отладочным сообщением является:
  // An error occurred: Pet name too short
  // (Возникла ошибка: Имя животного слишком короткое).
  trace("An error occurred: " + e.message);
} catch (e:Error) {
  // Здесь обрабатываются все остальные ошибки.
  trace("An error occurred: " + e.message);
}
```

Вариант 4. Пользовательские подклассы класса VirtualPetNameException. В варианте 3 мы добавляли конфигурируемые отладочные сообщения в класс `VirtualPetNameException`. Он помог выявить проблему в нашем коде на этапе разработки, однако этот вариант не позволяет программе выполнить независимые действия по восстановлению работоспособности после возникновения отдельных

ошибок класса VirtualPet. Чтобы программа могла выполнить отдельные ветки кода в зависимости от типа сгенерированной ошибки класса VirtualPet, нам потребуются пользовательские подклассы класса VirtualPetNameException, описанные в варианте 4.

Если вы хотите, чтобы программа могла находить различия между исключительными ситуациями, определяйте отдельный подкласс класса Error для каждой ошибки. Не полагайтесь исключительно на значение переменной message, чтобы реализовать логику ветвлений. Если ваш пользовательский подкласс класса Error определяет конструктор, принимающий сообщение об ошибке в качестве параметра, используйте это сообщение только для отладки, но не для построения логики ветвлений.

Чтобы иметь возможность устанавливать различие между тремя исключительными ситуациями класса VirtualPet, создадим три подкласса класса Error: VirtualPetNameException, VirtualPetInsufficientDataException и VirtualPetExcessDataException. Первый класс непосредственно расширяет класс Error. Оба следующих класса расширяют класс VirtualPetNameException, поскольку мы хотим отличать эти специфические типы ошибок от базового исключения, обозначающего недопустимые данные.

Рассмотрим исходный код наших трех подклассов класса Error, представляющих ошибки класса VirtualPet:

```
// Код в файле VirtualPetNameException.as:
package zoo {
  public class VirtualPetNameException extends Error {
    public function VirtualPetNameException (
                          message:String = "Invalid pet name specified.") {
      super(message);
    }
  }
}

// Код в файле VirtualPetInsufficientDataException.as:
package zoo {
  public class VirtualPetInsufficientDataException
                                          extends VirtualPetNameException {
    public function VirtualPetInsufficientDataException ( ) {
      super("Pet name too short.");
    }
  }
}

// Код в файле VirtualPetExcessDataException.as:
package zoo {
  public class VirtualPetExcessDataException
                                          extends VirtualPetNameException {
    public function VirtualPetExcessDataException ( ) {
      super("Pet name too long.");
    }
  }
```

Каждый класс определяет значение своей переменной `message` и не позволяет изменять его в процессе использования. При обработке любого из описанных исключений класса `VirtualPet` наша программа будет руководствоваться типом данных исключения (а не значением переменной `message`) для нахождения различия между тремя типами исключений.

Теперь, когда у нас появилось три типа исключений, добавим их генерацию в наш метод `setName( )`:

```
public function setName (newName:String):void {
  if (newName.indexOf(" ") == 0) {
    throw new VirtualPetNameException( );
  } else if (newName == "") {
    throw new VirtualPetInsufficientDataException( );
  } else if (newName.length > VirtualPet.maxNameLength) {
    throw new VirtualPetExcessDataException( );
  }

  petName = newName;
}
```

Обратите внимание, что в конструкторы различных исключений класса `VirtualPet` не передаются никакие описания ошибок. Еще раз повторим, что описание каждого исключения задается в соответствующем пользовательском подклассе класса `Error` через его переменную `message`.

Теперь, когда каждое исключение класса `VirtualPet` представлено собственным классом, программистам, работающим с экземплярами класса `VirtualPet`, известны все ошибки, которые могут быть сгенерированы методом `setName( )`. Типы исключений доступны за пределами класса `VirtualPet` и, соответственно, открыты для программистов, работающих над приложением. Разработчику достаточно взглянуть на иерархию классов приложения, чтобы определить исключения, относящиеся к классу `VirtualPet`. Более того, если он случайно укажет неправильное имя для исключения, компилятор сгенерирует ошибку типа.

Рассмотрим, как добавить в код логику ветвления, основанную на типах исключений, которые могут быть сгенерированы методом `setName( )`. Особое внимание обратите на типы данных параметров и размещение каждого блока `catch`.

```
try {
  b.setName("некоеИмяЖивотного");
} catch (e:VirtualPetExcessDataException) {
  // Обработка ситуации «слишком длинное».
  trace("An error occurred: " + e.message);
  trace("Please specify a shorter name.");
} catch (e:VirtualPetInsufficientDataException) {
  // Обработка ситуации «слишком короткое».
  trace("An error occurred: " + e.message);
  trace("Please specify a longer name.");
} catch (e:VirtualPetNameException) {
  // Обработка общих ошибок, связанных с именем.
  trace("An error occurred: " + e.message);
  trace("Please specify a valid name.");
}
```

В приведенном коде, если метод `setName( )` сгенерирует исключение `VirtualPetExcessDataException`, будет выполнен первый блок `catch`. Если метод сгенерирует исключение `VirtualPetInsufficientDataException`, будет выполнен второй блок `catch`. И наконец, если метод сгенерирует исключение `VirtualPetNameException`, будет выполнен третий блок `catch`. Обратите внимание, что в блоках `catch` сначала перечислены специфические типы данных ошибок, а затем — общие. При возникновении исключения выполняется тот блок `catch`, у которого первым совпадет тип данных параметра с типом данных исключения.

Таким образом, если мы изменим тип данных параметра первого блока `catch` на тип `VirtualPetNameException`, первый блок `catch` будет выполняться для всех трех типов исключений!

Вспомните, что класс VirtualPetNameException является суперклассом для обоих классов VirtualPetInsufficientDataException и VirtualPetExcessDataException, поэтому считается, что они соответствуют типу данных VirtualPetNameException.

Фактически мы могли бы предотвратить выполнение всех блоков `catch`, разместив первым новый блок `catch`, типом данных параметра которого является `Error`:

```
try {
  b.setName("некоеИмяЖивотного");
} catch (e:Error) {
  // Обрабатываем все ошибки. Никакие другие блоки catch
  // выполняться не будут.
  trace("An error occurred:" + e.message);
  trace("The first catch block handled the error.");
} catch (e:VirtualPetExcessDataException) {
  // Обработка ситуации «слишком длинное».
  trace("An error occurred: " + e.message);
  trace("Please specify a shorter name.");
} catch (e:VirtualPetInsufficientDataException) {
  // Обработка ситуации «слишком короткое».
  trace("An error occurred: " + e.message);
  trace("Please specify a longer name.");
} catch (e:VirtualPetNameException) {
  // Обработка общих ошибок, связанных с именем.
  trace("An error occurred: " + e.message);
  trace("Please specify a valid name.");
}
```

Очевидно, что попытка добавить первый блок `catch` в предыдущем коде обречена на провал, но этот пример иллюстрирует иерархическую природу обработки ошибок. Поместив базовый блок `catch` в самое начало списка обработчиков, мы можем обработать все ошибки в одном блоке. И наоборот, если поместить базовый блок `catch` в *конец* списка, мы можем создать «страховочную сетку», которая будет обрабатывать любые ошибки, не «пойманные» предыдущими блоками `catch`. Например, в следующем коде последний блок `catch` будет выполнен только в том

случае, если блок `try` сгенерирует исключение, которое не принадлежит типам данных `VirtualPetExcessDataException`, `VirtualPetInsufficientDataException` или `VirtualPetNameException`:

```
try {
  b.setName("некоеИмяЖивотного");
} catch (e:VirtualPetExcessDataException) {
  // Обработка переполнения.
  trace("An error occurred: " + e.message);
  trace("Please specify a smaller value.");
} catch (e:VirtualPetInsufficientDataException) {
  // Обработка нулевой длины.
  trace("An error occurred: " + e.message);
  trace("Please specify a larger value.");
} catch (e:VirtualPetNameException) {
  // Обработка общих ошибок, связанных с размерностью.
  trace("An error occurred: " + e.message);
  trace("Please specify a valid dimension.");
} catch (e:Error) {
  // Обработка любых ошибок, которые не относятся
  // к ошибкам VirtualPetNameException.
}
```

Не забывайте, что выбор степени детализации ошибок зависит от ситуации. Реализуя вариант 4, мы создали пользовательские подклассы класса `Error` для каждого исключения, генерируемого классом `VirtualPet`. Этот подход дает нашей программе максимальную возможность независимо обрабатывать различные типы ошибок. Однако во многих ситуациях подобная гибкость является излишней. Будет лучше, если степень детализации ошибок будет определяться требованиями логики вашей программы.

Передача исключений вверх по иерархии объектов

В ActionScript исключение может быть сгенерировано в любом месте программы, даже в сценарии кадра на временной шкале! В этом случае возникает вопрос: каким образом среда выполнения Flash находит соответствующий блок `catch` для обработки этого исключения? И что произойдет при отсутствии блоков `catch`? Эти загадки решаются с помощью магии *передачи исключений вверх по иерархии объектов*. Проследуем по пути передачи исключений вместе со средой Flash с того момента, как она выполнит в программе оператор `throw`. В ходе последующей инсценировки «размышления» среды выполнения Flash оформлены в виде комментариев.

После исполнения оператора `throw` нормальная работа программы немедленно прекращается и среда Flash пытается найти блок `try`, который содержит этот оператор. Например, рассмотрим оператор `throw`:

```
// Среда выполнения Flash: Хм. Оператор throw.
// Существует ли блок try, который содержит этот оператор?
throw new Error("Something went wrong");
```

Если оператор `throw` включен в блок `try`, среда выполнения Flash пытается найти блок `catch`, тип данных параметра которого совпадает с типом данных значения сгенерированного исключения (в данном случае с типом `Error`):

```
// Среда выполнения Flash: Отлично, я нашла блок try.
// Существует ли соответствующий блок catch?
try {
  throw new Error("Something went wrong");
}
```

Если соответствующий блок `catch` найден, среда выполнения Flash передает управление программой этому блоку:

```
try {
  throw new Error("Something went wrong");
// Среда выполнения Flash: Найден блок catch, типом данных параметра
// которого является Error! Поиск завершен. Сейчас я выполню этот
// блок catch...
} catch (e:Error) {
  // Обработка ошибок...
}
```

Однако если соответствующий блок `catch` не может быть найден или если оператор `throw` изначально не был включен в блок `try`, среда Flash проверяет, размещен ли этот оператор внутри метода или функции. Если да, то среда выполнения ищет блок `try` вокруг кода, вызвавшего этот метод или функцию. Следующий код демонстрирует, как среда Flash реагирует на оператор `throw`, который размещен внутри метода и не включен в блок `try`:

```
public function doSomething ( ):void {
  // Среда выполнения Flash: Хм. Блок try отсутствует.
  // Проверю, кто вызвал этот метод.
  throw new Error("Something went wrong");
}
```

Если код, вызвавший этот метод или функцию, включен в блок `try`, среда Flash попытается найти соответствующий блок `catch` и, если такой блок будет найден, выполнит его. Следующий код демонстрирует исключение, генерируемое в методе и обрабатываемое там, где вызывается этот метод (то есть уровнем выше по *стеку вызовов*):

```
public class ProblemClass {
  public function doSomething ( ):void {
    // Среда выполнения Flash: Хм. Блок try отсутствует.
    // Проверю-ка я, кто вызвал этот метод.
    throw new Error("Something went wrong");
  }
}

public class ErrorDemo extends Sprite {
  public function ErrorDemo ( ) {
```

```
    // Среда выполнения Flash: Ага, вот кто вызвал метод doSomething( ).
    // И вот блок try, включающий этот код, вместе с блоком catch, типом
    // данных параметра которого является Error! Моя работа сделана.
    // Блок catch, пожалуйста, выполняйтесь...
    try {
      var problemObject:ProblemClass = new ProblemClass( );
      problemObject.doSomething( );
    } catch (e:Error) {
      // Обработка ошибок...
      trace("Exception caught in ErrorDemo, thrown by doSomething( ).");
    }
  }
}
```

Стек вызовов — это список функций или методов программы, исполнением которых среда Flash занимается в любой момент времени. Функции и методы размещаются в списке в порядке, обратном порядку их вызова, по направлению сверху вниз. Если функция находится непосредственно под другой функцией в стеке вызовов, значит, нижняя функция была вызвана верхней функцией. Самая нижняя функция в стеке вызовов — это функция, выполняемая в настоящий момент.

В приложении Flex Builder и среде разработки Flash вы можете использовать отладчик для просмотра стека вызовов текущей программы, как описано в документации корпорации Adobe.

В предыдущем коде исключение, сгенерированное методом, было поймано блоком `try/catch`, в который включена инструкция вызова метода. Тем не менее, если вокруг кода, вызывающего функцию или метод, не найден блок `try`, среда выполнения Flash просматривает весь стек вызовов в поисках блока `try` с соответствующим блоком `catch`. Следующий код демонстрирует метод, генерирующий ошибку, которая обрабатывается двумя уровнями выше в стеке вызовов:

```
public class ProblemClass {
  public function doSomething ( ):void {
    // Среда выполнения Flash: Хм. Блок try отсутствует.
    // Проверю-ка я, кто вызвал этот метод.
    throw new Error("Something went wrong");
  }
}

public class NormalClass {
  public function NormalClass ( ) {
    // Среда выполнения Flash: Ага, вот кто вызвал метод doSomething( ).
    //                         Но здесь все равно нет блока try. Проверю,
    //                         кто вызвал этот метод.
    var problemObject:ProblemClass = new ProblemClass( );
    problemObject.doSomething( );
  }
}

public class ErrorDemo extends Sprite {
  public function ErrorDemo ( ) {
```

```
    // Среда выполнения Flash: Ага! Нашла блок try, который имеет блок
    //                         catch, типом данных параметра которого
    //                         является Error! Моя работа сделана.
    //                         Блок catch, пожалуйста, выполняйтесь...
    try {
      var normalObject:NormalClass = new NormalClass( );
    } catch (e:Error) {
      // Обработка ошибок...
      trace("Exception caught in ErrorDemo, thrown by doSomething( ).");
    }
  }
}
```

Обратите внимание, что среда выполнения Flash находит блок `try/catch`, даже несмотря на то, что этот блок не включает ни код, генерирующий ошибку, ни код, который вызывает метод, генерирующий ошибку, а только код, вызывающий метод, который, в свою очередь, вызывает метод, генерирующий ошибку!

Следующий код демонстрирует предыдущий пример с передачей исключения вверх по иерархии объектов в контексте нашей программы по созданию виртуального зоопарка. Для краткости в следующем листинге показан только код, присваивающий объекту имя животного. Комментарии в коде описывают, как происходит передача исключения.

```
package {
  import flash.display.Sprite;
  import zoo.*;

  public class VirtualZoo extends Sprite {
    private var pet:VirtualPet;

    public function VirtualZoo ( ) {
      try {
        // Этот код пытается присвоить животному слишком длинное имя.
        // В результате метод setName( ) генерирует ошибку.
        // Однако возникшее исключение не обрабатывается в конструкторе
        // класса VirtualPet (откуда вызывается метод setName( )). Вместо
        // этого исключение обрабатывается там, где вызывается конструктор
        // класса VirtualPet (то есть двумя уровнями выше в стеке вызовов)
        pet = new VirtualPet("Bartholomew McGillicuddy");
      } catch (e:Error) {
        trace("An error occurred: " + e.message);
        // Если в процессе создания объекта VirtualPet возникает исключение,
        // объект не будет создан. Таким образом, здесь мы создадим новый
        // объект VirtualPet с предопределенным допустимым именем.
        pet = new VirtualPet("Unnamed Pet");
      }
    }
  }
}

package zoo {
  public class VirtualPet {
```

```
    public function VirtualPet (name:String):void {
      // Даже несмотря на то, что метод setName( ) вызывается здесь,
      // исключения, генерируемые методом setName( ), не обрабатываются
      // в данном конструкторе. Они обрабатываются выше в стеке вызовов
      // кодом, который создал данный объект VirtualPet.
      setName(name);
    }

    public function setName (newName:String):void {
      // Исключения, генерируемые в этом методе, не обрабатываются здесь.
      // Они обрабатываются на два уровня выше в стеке вызовов кодом,
      // который создал данный объект VirtualPet.
      if (newName.indexOf(" ") == 0) {
        throw new VirtualPetNameException( );
      } else if (newName == "") {
        throw new VirtualPetInsufficientDataException( );
      } else if (newName.length > VirtualPet.maxNameLength) {
        throw new VirtualPetExcessDataException( );
      }

      petName = newName;
    }
  }
}
```

Необработанные исключения. Мы рассмотрели ряд сценариев, в которых обрабатывались различные ошибки. Что же произойдет в том случае, когда среда выполнения Flash не найдет блок `catch`, способный обработать сгенерированное исключение? Если подходящий блок `catch` не найден во всем стеке вызовов, то Flash завершает выполнение кода, который на текущий момент остается в стеке вызовов. Кроме того, если программа выполняется в отладочной версии среды Flash, информация о возникшей ошибке появится в отдельном окне: в окне Output (Вывод) (среда разработки Flash) или в окне Console (Консоль) (приложение Flex Builder). После этого выполнение программы продолжится в обычном режиме.

Следующий код демонстрирует метод, генерирующий ошибку, которая никогда не будет обработана:

```
public class ProblemClass {
  public function doSomething ( ):void {
    // Среда выполнения Flash: Хм. Блок try отсутствует.
    // Проверю-ка я, кто вызвал этот метод.
    throw new Error("Something went wrong");
  }
}

public class ErrorDemo extends Sprite {
  public function ErrorDemo ( ) {
    // Среда выполнения Flash: Ага, вот кто вызвал метод doSomething( ).
    // Но здесь все равно нет блока try. Хм. Я просмотрела весь стек вызовов
    // до самого верха и не нашла блока try. Если это отладочная версия
    // среды выполнения Flash, я сообщу о проблеме. Возможно, программист
```

```
    // знает, что нужно делать.
    var problemObject:ProblemClass = new ProblemClass( );
    problemObject.doSomething( );
  }
}
```

Как мы только что увидели, метод не обязан обрабатывать свои собственные исключения, поскольку исключения обладают способностью подниматься вверх по стеку вызовов. Исключения метода не обязан обрабатывать даже код, вызывающий этот метод. Обработка исключения допускается на любом уровне в стеке вызовов. Любой метод может делегировать обработку исключений коду, вызывающему данный метод. С другой стороны, генерация исключений, которые никогда не будут обработаны, является дурным тоном и оказывает опасное воздействие на программу. Вы должны всегда обрабатывать исключения или, если столкнетесь с необработанным исключением, в первую очередь изменить свой код, чтобы избежать повторной генерации данного исключения.

Блок finally

Пока мы рассмотрели лишь блоки `try` и `catch` инструкции `try/catch/finally`. Как уже известно, блок `try` содержит код, который может генерировать исключение, а блок `catch` включает в себя код, который выполняется в ответ на сгенерированное исключение. Для сравнения, блок `finally` содержит код, который выполняется всегда, независимо от того, возникло в блоке `try` исключение или нет.

Инструкция `try/catch/finally` содержит один (и только один) блок `finally`, который является ее последним блоком. Например:

```
try {
  // Вложенные инструкции
} catch (e:ТипОшибки1) {
  // Обработка исключений типа ТипОшибки1.
} catch (e:ТипОшибкиn) {
  // Обработка исключений типа ТипОшибкиn.
} finally {
  // Этот код выполняется всегда, независимо от того, как завершается
  // выполнение блока try.
}
```

Неправильное размещение блока `finally` вызовет ошибку на этапе компиляции.

В предыдущем коде блок `finally` будет выполнен сразу после того, как:

- выполнение блока `try` завершится без ошибок;
- блок `catch` обработает исключение, сгенерированное блоком `try`;
- необработанное исключение поднимется вверх по иерархии объектов;
- оператор `return`, `continue` или `break` передаст управление программой за пределы блоков `try` или `catch`.

Блок `finally` инструкции `try/catch/finally` обычно содержит очищающий код, который должен выполняться независимо от того, возникло исключение в соответ-

ствующем блоке try или нет. Предположим, что мы создаем игру в жанре «космический шутер» и определяем класс SpaceShip, представляющий космические корабли. У класса SpaceShip есть метод attackEnemy( ), который выполняет следующее.

- Устанавливает текущую цель для космического корабля.
- Стреляет по выбранной цели.
- Удаляет выбранную цель (присваивая переменной currentTarget объекта SpaceShip значение null).

Предположим, что в нашем гипотетическом приложении при выполнении первых двух из описанных задач может возникнуть исключение. Более того, предположим, что метод attackEnemy( ) не обрабатывает эти исключения самостоятельно; он передает исключения вызывающему методу. Независимо от того, было сгенерировано исключение или нет, метод attackEnemy( ) должен присвоить переменной currentTarget значение null.

Вот так выглядел бы метод attackEnemy( ), если бы мы запрограммировали его с помощью оператора catch (то есть без использования блока finally):

```
public function attackEnemy (enemy:SpaceShip):void {
  try {
    setCurrentTarget(enemy);
    fireOnCurrentTarget( );
  } catch (e:Error) {
    // Удаляем текущую цель, если возникло исключение.
    setCurrentTarget(null);
    // Передаем исключение вызывающему методу.
    throw e;
  }
  // Удаляем текущую цель, если никаких исключений не возникло.
  setCurrentTarget(null);
}
```

Здесь мы вынуждены дублировать инструкцию setCurrentTarget(null). Мы поместили ее и внутрь блока catch, и после инструкции try/catch, гарантируя тем самым, что она будет выполнена независимо от того, возникло исключение в блоке try или нет. Тем не менее дублирование инструкции может привести к ошибке. В предыдущем методе программист мог бы легко забыть удалить текущую цель после блока try/catch.

Если изменить нашу стратегию так, чтобы текущая цель удалялась в блоке finally, мы устраним ненужные команды в предыдущем коде:

```
public function attackEnemy (enemy:SpaceShip):void {
  try {
    setCurrentTarget(enemy);
    fireOnCurrentTarget( );
  } finally {
    setCurrentTarget(null);
  }
}
```

В модифицированной версии блок finally удаляет текущую цель независимо от того, возникло исключение или нет. Поскольку блок finally обрабатывает обе ситуации, у нас нет необходимости в блоке catch; мы можем просто позволить исключению автоматически подняться вверх к вызывающему методу.

Вы можете поинтересоваться, а зачем нам вообще нужен блок finally? Иными словами, почему нельзя просто использовать следующий код?

```
// Этот код выглядит подходящим, однако в нем существует проблема.
// Сможете определить ее?
public function attackEnemy (enemy:SpaceShip):void {
  setCurrentTarget(enemy);
  fireOnCurrentTarget( );
  setCurrentTarget(null);
}
```

Запомните, что, когда генерируется исключение, управление программой передается в ближайший подходящий блок catch в стеке вызовов. Следовательно, если метод fireOnCurrentTarget( ) генерирует исключение, управление передается в метод attackEnemy( ), при этом обратно в метод fireOnCurrentTarget( ) управление возвращено не будет и инструкция setCurrentTarget(null) останется невыполненной. Однако с помощью блока finally мы гарантируем, что инструкция setCurrentTarget(null) будет выполнена до того, как исключение поднимется вверх по иерархии объектов.

Пример метода attackEnemy( ) отражает наиболее распространенное использование блока finally в многопоточных приложениях, в которых одновременно может выполняться сразу несколько фрагментов кода и которые разрабатываются с помощью таких языков программирования, как, например, Java. В языке Java следующая общая структура — обычное явление; она исключает возможность удаления объекта, выполняющего некую задачу, другим объектом в процессе выполнения текущей задачи:

```
// Установить состояние, обозначающее выполнение данным объектом текущей
// задачи. Внешние объекты должны проверять состояние данного объекта перед
// тем, как обратиться к нему или выполнить над ним какие-либо действия.
doingSomething = true;
try {
  // Выполняем задачу.
  doSomething( );
} finally {
  // Сбросить состояние, обозначающее выполнение текущей задачи (независимо
  // от того, возникло в процессе выполнения задачи исключение или нет).
  doingSomething = false;
}
```

В языке ActionScript приведенный код, управляющий состоянием объекта, на самом деле необязателен, поскольку разрабатываемые с помощью этого языка приложения являются однопоточными, следовательно, никакой внешний объект не сможет изменить состояние другого объекта, выполняющего некий метод. Таким образом, в языке ActionScript блок finally используется гораздо реже, чем в языках, применяемых для разработки многопоточных приложений. Тем не менее этот блок

можно использовать для организационных целей — помещать в него код, который выполняет очистку после выполнения другого кода.

Вложенные исключения

До сих пор мы использовали только одноуровневые инструкции `try/catch/finally`, однако логика обработки исключений может быть и вложенной. Инструкция `try/catch/finally` может размещаться внутри блока `try`, `catch` или `finally` другой инструкции `try/catch/finally`. Такое иерархическое вложение позволяет любому блоку инструкции `try/catch/finally` выполнять код, который, в свою очередь, может генерировать исключения.

Предположим, что мы создаем многопользовательское веб-приложение, представляющее доску объявлений. Мы определяем следующие классы: `BulletinBoard` — основной класс приложения, `GUIManager` — класс, управляющий пользовательским интерфейсом, и `User` — класс, который представляет пользователя на доске. В классе `BulletinBoard` мы описываем метод `populateUserList( )`, который отображает список активных пользователей на текущий момент. Выполнение метода `populateUserList( )` состоит из двух этапов: на первом этапе метод получает экземпляр класса `List` из экземпляра класса `GUIManager` нашего приложения. Класс `List` представляет отображаемый на экране список пользователей. Затем метод `populateUserList( )` заполняет экземпляр класса `List` пользователями из переданного массива экземпляров класса `User`. На обоих этапах существует потенциальная возможность возникновения исключения, поэтому в методе `populateUserList( )` используется вложенная структура `try/catch/finally`. Рассмотрим эту вложенную структуру поближе.

Если на первом этапе выполнения метода `populateUserList( )` экземпляр класса `List` окажется недоступным, будет сгенерировано исключение `UserListNotFound` экземпляром класса `GUIManager`. Исключение `UserListNotFound` обрабатывается внешней инструкцией `try/catch/finally`.

Если, с другой стороны, экземпляр класса `List` окажется доступным, метод `populateUserList( )` перейдет к выполнению второго этапа, где с помощью цикла заполнит экземпляр класса `List` пользователями из переданного массива. На каждой итерации цикла, если ID текущего пользователя не может быть найдено, метод `User.geID( )` генерирует исключение `UserIdNotSet`. Оно обрабатывается вложенной инструкцией `try/catch/finally`.

Рассмотрим этот код:

```
public function populateUserList (users:Array):void {
  try {
    // Приступаем к выполнению этапа 1... получаем экземпляр класса List.
    // Если метод getUserList( ) сгенерирует исключение, будет выполнен
    // внешний блок catch.
    var ulist:List = getGUIManager( ).getUserList( );
    // Приступаем к выполнению этапа 2... заполняем экземпляр класса List.
    for (var i:Number = 0; i < users.length; i++) {
      try {
```

```
        var thisUser:User = User(users[i]);
        // Если метод getID( ) сгенерирует исключение, будет выполнен
        // вложенный блок catch. В противном случае пользователь будет
        // добавлен в экземпляр класса List вызовом метода addItem( ).
        ulist.addItem(thisUser.getName( ), thisUser.getID( ));
      } catch (e:UserIdNotSet) {
        trace(e.message);
        continue;  // Пропускаем этого пользователя.
      }
    }
  } catch (e:UserListNotFound) {
    trace(e.message);
  }
}
```

Теперь, когда мы рассмотрели конкретный пример вложенного исключения, познакомимся с общим процессом обработки вложенных исключений.

Если исключение генерируется в блоке `try`, вложенном в другой блок `try`, и внутренний блок `try` содержит блок `catch`, способный обработать сгенерированное исключение, выполняется внутренний блок `catch` и программа продолжает свое выполнение сразу после внутренней инструкции `try/catch/finally`.

```
try {
  try {
    // Здесь генерируется исключение.
    throw new Error("Test error");
  } catch (e:Error) {
    // Здесь обрабатывается исключение.
    trace(e.message);  // Выводит: Test error
  }
  // Здесь продолжается выполнение программы.
} catch (e:Error) {
  // Обработка исключений, генерируемых внешним блоком try.
}
```

Если, с другой стороны, исключение возникло в блоке `try`, вложенном в другой блок `try`, однако внутренний блок `try` не содержит блока `catch`, способного обработать данное исключение, сгенерированное исключение будет передаваться вверх к внешней инструкции `try/catch/finally` (и при необходимости дальше по стеку вызовов) до тех пор, пока не будет найден подходящий блок `catch` или пока не будет достигнута верхняя точка стека вызовов. Если исключение будет обработано в некоторой точке стека вызовов, то выполнение программы продолжится сразу после инструкции `try/catch/finally`, обработавшей это исключение. Обратите внимание, что в следующем примере кода (и последующих примерах) гипотетический тип данных ошибки `SomeSpecificError` является заполнителем, используемым для того, чтобы сгенерированное исключение не было поймано. Чтобы протестировать пример кода в вашем собственном коде, вы должны создать подкласс `SomeSpecificError` класса `Error`.

```
try {
  try {
```

```
    // Здесь генерируется исключение.
    throw new Error("Test error");
  } catch (e:SomeSpecificError) {
    // Здесь исключение не обрабатывается.
    trace(e.message);  // Инструкция никогда не будет выполнена,
                       // поскольку типы не совпадают.
  }
} catch (e:Error) {
  // Исключение обрабатывается здесь.
  trace(e.message);  // Выводит: Test error
}
// Выполнение программы продолжается здесь, сразу после обработки исключения
// внешним блоком catch.
```

Если исключение генерируется в блоке `try`, вложенном в блок `catch`, и внутренний блок `try` не имеет блока `catch`, способного обработать данное исключение, поиск подходящего блока `catch` начинается за пределами внешней инструкции `try/catch/finally`:

```
try {
  // Здесь генерируется внешнее исключение.
  throw new Error("Test error 1");
} catch (e:Error) {
  // Здесь обрабатывается внешнее исключение.
  trace(e.message);  // Выводит: Test error 1
  try {
    // Здесь генерируется внутреннее исключение.
    throw new Error("Test error 2");
  } catch (e:SomeSpecificError) {
    // Внутреннее исключение здесь не обрабатывается.
    trace(e.message);  // Инструкция никогда не будет выполнена,
                       // поскольку типы не совпадают.
  }
}
// Процесс поиска подходящего блока catch для внутреннего исключения
// начинается здесь.
```

Наконец, если исключение генерируется в блоке `try`, вложенном в блок `finally`, но при этом предыдущее исключение уже находится в процессе подъема по стеку вызовов, новое исключение будет обработано до того, как предыдущее исключение продолжит свой подъем по иерархии объектов.

```
// Этот метод генерирует исключение в блоке finally.
public function throwTwoExceptions ( ):void {
  try {
    // Здесь генерируется внешнее исключение. Поскольку этот блок try
    // не имеет соответствующего блока catch, внешнее исключение начинает
    // свой подъем по иерархии объектов.
    throw new Error("Test error 1");
  } finally {
    try {
      // Здесь возникает внутреннее исключение. Внутреннее исключение
      // будет обработано до того, как внешнее исключение фактически начнет
```

```
      // подъем по иерархии объектов.
      throw new Error("Test error 2");
    } catch (e:Error) {
      // Внутреннее исключение обрабатывается здесь.
      trace("Internal catch: " + e.message);
    }
  }
}

// Где-то в другом месте, внутри метода,
// вызывающего предыдущий метод.
try {
  throwTwoExceptions( );
} catch (e:Error) {
  // Здесь обрабатывается внешнее исключение,
  // поднявшееся из метода throwTwoExceptions( ).
  trace("External catch: " + e.message);
}

// Вывод (обратите внимание, что внутреннее исключение обработано первым):
// Internal catch: Test error 2
// External catch: Test error 1
```

Если бы в предыдущем коде исключение, сгенерированное в блоке `finally`, не было обработано, среда Flash сообщила бы об ошибке на этапе отладки и прекратила бы выполнение оставшегося кода в стеке вызовов. В результате никакой код в стеке вызовов не обработал бы сгенерированное первым внешнее исключение. Следующий код демонстрирует описанный сценарий. Этот код генерирует необрабатываемое исключение в блоке `finally`. В итоге исключение, сгенерированное внешним блоком `try`, будет потеряно.

```
try {
  // Здесь генерируется внешнее исключение.
  throw new Error("Test error 1");
} finally {
  try {
    // Здесь генерируется внутреннее исключение.
    throw new Error("Test error 2");
  } catch (e:SomeSpecificError) {
    // Внутреннее исключение здесь не обрабатывается.
    trace("internal catch: " + e.message);  // Инструкция никогда не будет
                                            // выполнена, поскольку типы
                                            // не совпадают.
  }
}
// Процесс поиска подходящего блока catch для внутреннего исключения
// начинается здесь. Если внутреннее исключение не будет обработано, о нем
// сообщит среда выполнения Flash на этапе отладки, при этом передача
// внешнего исключения вверх по иерархии объектов будет прекращена.
```

Предыдущий код демонстрирует влияние необработанного исключения на ход выполнения программы в одном сценарии, однако хочется еще раз повторить, что

не следует допускать появления необрабатываемых исключений. В предыдущем случае мы должны либо обработать исключение, либо модифицировать наш код таким образом, чтобы данное исключение больше не появлялось.

Изменение хода выполнения программы в инструкции try/catch/finally

На протяжении этой главы мы неоднократно убеждались в том, что инструкция `throw` изменяет ход выполнения программы. Когда среда Flash встречает инструкцию `throw`, она немедленно прекращает выполнение текущей задачи и передает управление программой подходящим блокам `catch` и `finally`. Тем не менее блоки `catch` и `finally` также могут влиять на ход выполнения программы с помощью оператора `return` (в случае метода или функции), `break` или `continue` (в случае цикла). Более того, операторы `return`, `break` и `continue` могут также использоваться в блоке `try`.

Чтобы познакомиться с правилами изменения хода выполнения программы в инструкции `try/catch/finally`, рассмотрим, как оператор `return` влияет на ход выполнения программы в блоках `try`, `catch` и `finally`. Следующий пример кода содержит функцию `changeFlow ( )`, которая демонстрирует ход выполнения программы в различных гипотетических ситуациях.

В листинге 13.1 показан оператор `return` в блоке `try`, помещенный перед инструкцией, генерирующей ошибку. В данном случае метод выполняется нормально и никакая ошибка не будет сгенерирована или обработана. Однако перед возвратом из метода будет выполнен блок `finally`. Стоит отметить, что вы вряд ли увидите код, в точности повторяющий код из листинга 13.1, в реальной программе. В большинстве случаев оператор `return` используется в условных операторах и выполняется в ответ на некоторое определенное условие в программе.

Листинг 13.1. Использование оператора return в блоке try перед оператором throw

```
public function changeFlow ( ):void {
  try {
    return;
    throw new Error("Test error.");
  } catch (e:Error) {
    trace("Caught: " + e.message);
  } finally {
    trace("Finally executed.");
  }
  trace("Last line of method.");
}

// Вывод после вызова метода changeFlow( ):
// Finally executed.
```

В листинге 13.2 показан оператор `return` в блоке `try`, помещенный после инструкции, генерирующей ошибку. В данном случае `return` не выполнится, поскольку ошибка будет сгенерирована до того, как программа дойдет до этого оператора. Как только

ошибка будет обработана и выполнение инструкции `try/catch/finally` завершится, выполнение программы продолжится после данной инструкции `try/catch/finally` и выход из метода произойдет в конце его тела. Снова повторим, что листинг 13.2 всего лишь демонстрирует принцип, но не является типичным в реальном сценарии, поскольку ошибка обычно генерируется на основании некоторого условия.

Листинг 13.2. Использование оператора return в блоке try после оператора throw

```
public function changeFlow ( ):void {
  try {
    throw new Error("Test error.");
    return;
  } catch (e:Error) {
    trace("Caught: " + e.message);
  } finally {
    trace("Finally executed.");
  }
  trace("Last line of method.");
}

// Вывод после вызова метода changeFlow( ):
// Caught: Test error.
// Finally executed.
// Last line of method.
```

Листинг 13.3 демонстрирует использование оператора `return` в блоке `catch`. В данном случае `return` будет выполнен после завершения процесса обработки ошибки. При этом код, размещенный после инструкции `try/catch/finally`, выполнен не будет. Однако, как обычно, перед возвратом из метода будет выполнен блок `finally`. В отличие от листингов 13.1 и 13.2 данный код *является* типичным в реальном сценарии, когда выполнение метода прекращается при возникновении ошибки.

Листинг 13.3. Использование оператора return в блоке catch

```
public function changeFlow ( ):void {
  try {
    throw new Error("Test error.");
  } catch (e:Error) {
    trace("Caught: " + e.message);
    return;
  } finally {
    trace("Finally executed.");
  }
  trace("Last line of function.");
}

// Вывод после вызова метода changeFlow( ):
// Caught: Test error.
// Finally executed.
```

Из-за известной ошибки в приложении Flash Player 9 при выполнении кода из листингов 13.2 и 13.3 происходит обращение к несуществующей области стека. Корпорация Adobe планирует исправить эту проблему в следующей версии приложения Flash Player.

В листинге 13.4 показан оператор `return` в блоке `finally`. В данном случае он выполняется тогда, когда выполняется блок `finally` (как уже известно, выполнение блока `finally` происходит после завершения соответствующего блока `try` одним из следующих способов: без ошибок; с обработанной ошибкой; с необработанной ошибкой; в результате вызова операторов `return`, `break` или `continue`). Обратите внимание, что оператор `return` в листинге 13.4 предотвращает выполнение кода, размещенного после инструкции `try/catch/finally`. Вы можете использовать подобную методику для завершения метода после выполнения блока кода независимо от того, было сгенерировано исключение или нет. При этом вся инструкция `try/catch/finally` обычно помещается внутрь условного оператора (иначе оставшаяся часть метода никогда не будет выполнена!).

Листинг 13.4. Использование оператора return в блоке finally

```
public function changeFlow ( ):void {
  try {
    throw new Error("Test error.");
  } catch (e:Error) {
    trace("Caught: " + e.message);
  } finally {
    trace("Finally executed.");
    return;
  }
  trace("Last line of method.");  // Не выполняется.
}

// Вывод после вызова метода changeFlow( ):
Caught: Test error.
Finally executed.
```

Если оператор return в блоке finally вызывается после того, как был вызван оператор return в соответствующем блоке try, вызов return в блоке finally отменяет предыдущий вызов return.

Обработка предопределенного исключения

В самом начале этой главы мы узнали, что с помощью инструкции `try/catch/finally` можно обрабатывать как предопределенные, так и пользовательские ошибки. До настоящего момента все обрабатываемые нами ошибки были пользовательскими. Сейчас мы рассмотрим инструкцию `try/catch/finally`, обрабатывающую предопределенную ошибку.

Предположим, что мы создаем приложение для обмена текстовыми сообщениями, в котором при соединении с сервером данного приложения пользователю предлагается ввести номер порта. Мы присваиваем указанный номер порта переменной с именем `userPort`. После этого мы пытаемся подключиться к указанному порту,

используя класс `Socket`. В некоторых случаях установить соединение не получится из-за ограничений безопасности. Чтобы показать, что причиной неудачной попытки соединения являются ограничения безопасности, среда выполнения Flash генерирует исключение `SecurityError`. Таким образом, при попытке создать соединение мы помещаем соответствующий код в блок `try`. Если установить соединение не получится по причинам безопасности, мы отобразим для пользователя сообщение об ошибке, обозначив проблему, из-за которой возникла данная ошибка.

```
var socket:Socket = new Socket( );
try {
  // Пытаемся подключиться к указанному порту
  socket.connect("example.com", userPort);
} catch (e:SecurityError) {
  // Код, расположенный здесь, отображает сообщение для пользователя
}
```

Список причин, которые могут привести к ошибкам соединения сокета, можно найти в описании метода connect( ) класса Socket в справочнике по языку ActionScript корпорации Adobe.

События об ошибках в случае проблемных ситуаций. В предыдущем разделе мы рассмотрели, как обработать исключение, вызванное недопустимой попыткой установить соединение сокета. Однако не все сбойные ситуации в языке ActionScript приводят к генерации исключений. Информация о проблемах, возникающих асинхронно (то есть спустя некоторое время), передается через события об ошибках, а не через исключения. Например, если мы попытаемся загрузить файл, среда выполнения Flash в асинхронном режиме сначала должна проверить, существует ли запрашиваемый файл. Если этот файл не существует, среда Flash выполнит диспетчеризацию события `IOErrorEvent.IO_ERROR`. Чтобы обработать возникшую проблему, код, инициировавший операцию загрузки, должен зарегистрировать обработчик для события `IOErrorEvent.IO_ERROR`. Если этого не произойдет, возникнет ошибка на этапе выполнения. Пример обработчика события об ошибке можно найти среди примеров в подразд. «Два дополнительных примера регистрации приемников событий» разд. «Основы обработки событий в ActionScript» гл. 12.

Впереди еще одна скучная работа

Исключения никогда не являлись самым эффектным аспектом программирования. Зачастую куда более забавно создавать программы, нежели выяснять, почему эти программы работают неправильно. Тем не менее обработка ошибок — это важная часть в разработке любой программы. В следующей главе мы рассмотрим еще одну похожую скучную тему, посвященную *сборке мусора*. Сборка мусора помогает предотвратить ситуацию, когда программа полностью исчерпывает системную память.

Сборка мусора

Всякий раз, когда программа создает объект, среда выполнения Flash сохраняет его в системной памяти (например, ОЗУ). По мере того как программа создает сотни, тысячи или даже миллионы объектов, объем памяти, занимаемой этой программой, постепенно увеличивается. Чтобы предотвратить полное исчерпывание системной памяти, Flash автоматически удаляет из нее объекты, когда программа перестает в них нуждаться. Процесс автоматического удаления объектов из памяти называется *сборкой мусора*.

Доступность объектов для сборки мусора

В программе на языке ActionScript любой объект становится доступным для процесса сборки мусора сразу после того, как он окажется *недостижимым*. Объект считается недостижимым, когда к нему невозможно обратиться прямо или косвенно хотя бы через один *корневой элемент сборки мусора*. К наиболее значимым корневым элементам сборки мусора в языке ActionScript можно отнести следующие:

- переменные, определенные на уровне пакета;
- локальные переменные в выполняющемся методе или функции;
- статические переменные;
- переменные экземпляра основного класса программы;
- переменные экземпляра объекта, находящегося в списке отображения среды выполнения Flash;
- переменные, находящиеся в цепочке областей видимости выполняющейся функции или метода.

Например, рассмотрим следующую версию класса `VirtualZoo` из нашей программы «Зоопарк». Обратите внимание, что в этой версии объект `VirtualPet`, создаваемый в конструкторе класса `VirtualZoo`, не присваивается переменной.

```
package {
  import flash.display.Sprite;
  import zoo.*;

  public class VirtualZoo extends Sprite {
    public function VirtualZoo ( ) {
      new VirtualPet("Stan");
    }
  }
}
```

При выполнении конструктора класса VirtualZoo из предыдущего кода выражение new VirtualPet("Stan") создает новый объект класса VirtualPet. Однако обратиться к объекту после его создания невозможно, поскольку он не был присвоен никакой переменной. Как результат, созданный объект считается недостижимым и сразу же становится доступным для сборки мусора.

Теперь рассмотрим следующую версию класса VirtualZoo. Особое внимание уделите телу метода-конструктора, выделенного полужирным шрифтом:

```
package {
  import flash.display.Sprite;
  import zoo.*;

  public class VirtualZoo extends Sprite {
    public function VirtualZoo ( ) {
      var pet:VirtualPet = new VirtualPet("Stan");
    }
  }
}
```

Как и раньше, при выполнении конструктора класса VirtualZoo из предыдущего кода выражение new VirtualPet("Stan") создает новый объект класса VirtualPet. Однако на этот раз созданный объект класса VirtualPet присваивается локальной переменной pet. В конструкторе класса VirtualZoo к объекту VirtualPet можно обратиться через эту локальную переменную, поэтому данный объект недоступен для сборки мусора. Тем не менее, как только выполнение конструктора VirtualZoo будет завершено, время жизни переменной pet истечет и обратиться к объекту VirtualPet будет невозможно. Как результат, объект считается недостижимым и становится доступным для сборки мусора.

Далее рассмотрим следующую версию класса VirtualZoo:

```
package {
  import flash.display.Sprite;
  import zoo.*;

  public class VirtualZoo extends Sprite {
    private var pet:VirtualPet;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
    }
  }
}
```

При выполнении конструктора класса VirtualZoo из предыдущего кода выражение new VirtualPet("Stan") снова создает объект класса VirtualPet. Однако на этот раз созданный объект VirtualPet присваивается переменной экземпляра основного класса программы. По существу, этот объект считается достижимым и поэтому недоступен для сборки мусора.

Теперь рассмотрим следующую версию класса VirtualZoo (снова обратите внимание на фрагменты кода, выделенные полужирным шрифтом):

```
package {
  import flash.display.Sprite;
  import zoo.*;

  public class VirtualZoo extends Sprite {
    private var pet:VirtualPet;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      pet = new VirtualPet("Tim");
    }
  }
}
```

Как и раньше, первая строка конструктора из предыдущего кода создает объект класса `VirtualPet` и присваивает его переменной экземпляра `pet`. Однако вторая строка метода-конструктора из предыдущего кода создает *еще один* объект класса `VirtualPet` и тоже присваивает его переменной экземпляра `pet`. Вторая операция присваивания заменяет первое значение переменной `pet` (то есть животное с именем Stan) новым значением (то есть животным с именем Tim). Как результат, объект `VirtualPet` с именем Stan считается недостижимым и становится доступным для сборки мусора. Стоит отметить, что, если бы мы присвоили переменной `pet` значение `null` (или любое другое допустимое значение), как показано в следующем коде, объект `VirtualPet` с именем Stan также оказался бы недостижимым:

```
pet = null;
```

Наконец, рассмотрим следующую версию класса `VirtualZoo`, в которой определены две переменные экземпляра `pet1` и `pet2`:

```
package {
  import flash.display.Sprite;
  import zoo.*;

  public class VirtualZoo extends Sprite {
    private var pet1:VirtualPet;
    private var pet2:VirtualPet;

    public function VirtualZoo ( ) {
      pet1 = new VirtualPet("Stan");
      pet2 = pet1;
      pet1 = null;
      pet2 = null;
    }
  }
}
```

Как и раньше, первая строка конструктора из предыдущего кода создает объект класса `VirtualPet`; для удобства назовем его `"Stan"`. Кроме того, первая строка присваивает объект `"Stan"` переменной `pet1`. Затем вторая строка присваивает тот же объект переменной экземпляра `pet2`. После выполнения второй строки конструктора и перед выполнением третьей строки программа может обратиться

к объекту "Stan" как через переменную pet1, так и через переменную pet2, поэтому данный объект недоступен для сборки мусора. Третья строка заменяет значение переменной pet1 значением null, так что объект "Stan" оказывается недоступен через эту переменную. Тем не менее к объекту "Stan" по-прежнему *можно* обратиться через переменную pet2, поэтому данный объект все еще недоступен для сборки мусора. Наконец, четвертая строка заменяет значение переменной pet2 значением null. Как результат, к объекту "Stan" больше нельзя обратиться через какую-либо переменную. «Бедный» объект "Stan" оказывается недостижимым и официально становится доступным для сборки мусора.

В языке ActionScript существует два особых случая, когда являющийся достижимым объект становится доступным для сборки мусора. Соответствующую информацию можно найти в разд. «Приемники событий и управление памятью» гл. 12 и в описании класса Dictionary справочника по языку ActionScript корпорации Adobe.

Алгоритм поэтапных пометок и вычищений

В предыдущем разделе рассказывалось, что объект становится доступным для сборки мусора (автоматического удаления из памяти) в тот момент, когда он оказывается недостижимым. Однако мы так и не выяснили, когда недостижимые объекты удаляются из памяти. В идеальном мире объекты должны удаляться из памяти сразу после того, как они оказались недостижимыми. Однако на практике незамедлительное удаление недостижимых объектов отнимает очень много времени и приводит к тому, что большинство нетривиальных программ выполняется медленно или вообще не отвечает на запросы. Соответственно среда выполнения Flash не удаляет недостижимые объекты из памяти незамедлительно. Вместо этого, она ищет и удаляет недостижимые объекты периодически, в процессе выполнения *циклов сборки мусора*.

Стратегия удаления недостижимых объектов языка ActionScript называется сборкой мусора с использованием алгоритма *поэтапных пометок и вычищений*. Рассмотрим, как это работает.

После запуска среда Flash просит операционную систему оставить, или *выделить*, произвольный объем памяти, в которой будут храниться объекты выполняемой в настоящий момент программы. По мере выполнения программы объекты накапливаются в памяти. В любой момент времени некоторые объекты программы являются достижимыми, а другие могут оказаться недостижимыми. Если программа создаст достаточное количество объектов, среда выполнения Flash в конечном счете решит выполнить цикл сборки мусора. В процессе выполнения цикла все объекты, находящиеся в памяти, проверяются на «достижимость». В результате все достижимые объекты *помечаются* и продолжают храниться в памяти, а все недостижимые объекты *вычищаются* (удаляются)

из памяти. Однако в большой программе проверка объектов на достижимость может оказаться достаточно длительной. Соответственно среда Flash разбивает циклы сборки мусора на небольшие части, или *приращения*, которые включаются в процесс выполнения программы. Кроме того, среда выполнения Flash использует механизм отложенного подсчета ссылок для повышения производительности процесса сборки мусора. Информацию о механизме отложенного подсчета ссылок можно найти в статье «The Memory Management Reference: Beginner's Guide: Recycling» по адресу http://www.memorymanagement.org/articles/recycle.html#reference.deferred.

В языке ActionScript циклы сборки мусора обычно запускаются тогда, когда объем памяти, необходимый для хранения объектов программы, достигает объема памяти, выделенного среде Flash. Однако ActionScript не дает *никакой гарантии* относительно времени запуска циклов сборки мусора. Более того, программист не может заставить среду Flash выполнить цикл сборки мусора.

Намеренное освобождение объектов

В языке ActionScript не существует прямого способа для удаления объекта из системной памяти. Удаление любых объектов программы осуществляется косвенно, через автоматическую систему сборки мусора.

Тем не менее, хотя программа и не может самостоятельно удалить объект из системной памяти, она по крайней мере может сделать этот объект доступным для удаления, устранив все ссылки на него. Чтобы устранить все ссылки на объект, мы должны вручную удалить этот объект из любых массивов, содержащих его, и присвоить значение `null` (или другое подходящее значение) всем переменным, ссылающимся на него.

Тот факт, что объект является доступным для сборки мусора, вовсе не означает, что он будет незамедлительно удален из памяти. Среда выполнения Flash просто получает разрешение на удаление этого объекта, когда (и если) запустится цикл сборки мусора.

Стоит отметить, однако, что создание и последующее удаление объектов из памяти зачастую является менее эффективным решением, чем повторное использование существующих объектов.

Когда это возможно, вы должны стараться повторно использовать существующие объекты, а не удалять их.

Снова вернемся к программе по созданию виртуального зоопарка и рассмотрим следующий код. В нем описывается метод для кормления животного яблоками.

```
package {
  import flash.display.Sprite;
  import zoo.*;
```

```
  public class VirtualZoo extends Sprite {
    private var pet:VirtualPet;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      pet.eat(new Apple( ));
      pet.eat(new Apple( ));
      pet.eat(new Apple( ));
      pet.eat(new Apple( ));
      pet.eat(new Apple( ));
    }
  }
}
```

Обратите внимание, что при каждом кормлении животного предыдущий код создает новый экземпляр класса `Apple` и передает этот экземпляр в метод `eat ( )`. Всякий раз, когда завершается выполнение метода `eat ( )`, все ссылки на переданный в этот метод экземпляр класса `Apple` теряются, поэтому данный экземпляр становится доступным для сборки мусора. Теперь представьте, что произойдет, если мы скормим животному 1000 объектов класса `Apple`. Затраты нашей программы на обработку данных будут включать не только затраты на создание объектов класса `Apple`, но и затраты на их удаление из памяти в процессе сборки мусора. Чтобы минимизировать подобные затраты, лучше создать один повторно используемый объект класса `Apple` и применять его при каждом кормлении животного яблоком. Следующий код демонстрирует процесс пятикратного кормления животного одним и тем же объектом класса `Apple`:

```
package {
  import flash.display.Sprite;
  import zoo.*;

  public class VirtualZoo extends Sprite {
    private var pet:VirtualPet;
    private var apple:Apple;

    public function VirtualZoo ( ) {
      pet = new VirtualPet("Stan");
      apple = new Apple( );

      pet.eat(apple);
      pet.eat(apple);
      pet.eat(apple);
      pet.eat(apple);
      pet.eat(apple);
    }
  }
}
```

К затратам предыдущего кода можно отнести затраты на создание единственного объекта. При этом не будет вообще *никаких* затрат на сборку мусора. Этот подход гораздо более эффективен по сравнению с предыдущим подходом, заключавшимся в создании нового объекта класса `Apple` и его последующем удалении для каждого вызова метода `eat ( )`!

Деактивация объектов

Как уже известно, удаление всех ссылок на объект делает его доступным для сборки мусора. Тем не менее даже после этого объект продолжает существовать в памяти до тех пор, пока среда выполнения Flash не решит «смести» его в одном из циклов сборки мусора. С того момента, как объект становится доступным для сборки мусора, и до того, как он фактически будет удален из системной памяти, данный объект продолжает получать события и, в случае объектов класса `Function`, может по-прежнему вызываться функцией `setInterval( )`.

Например, представьте приложение для демонстрации изображений в режиме слайд-шоу, которое использует класс `ImageLoader` для загрузки изображений с сервера через определенные интервалы времени. Код класса `ImageLoader` выглядит следующим образом:

```
package {
  import flash.events.*;
  import flash.utils.*;

  public class ImageLoader {
    private var loadInterval:int;

    public function ImageLoader (delay:int = 1000) {
      loadInterval = setInterval(loadImage, delay);
    }

    public function loadImage ( ):void {
      trace("Now loading image...");
      // Код загрузки изображения
      // здесь не приводится
    }
  }
}
```

Теперь представьте, что основной класс приложения `SlideShow` реализует функциональность для запуска и остановки слайд-шоу. Для запуска слайд-шоу класс `SlideShow` создает экземпляр класса `ImageLoader`, управляющего процессом загрузки изображений. Экземпляр класса `ImageLoader` сохраняется в переменной экземпляра `imgLoader`, как показано в следующем коде:

```
imgLoader = new ImageLoader( );
```

Для остановки или приостановки слайд-шоу класс `SlideShow` удаляет ссылку на экземпляр класса `ImageLoader`, как показано в следующем коде:

```
imgLoader = null;
```

Когда переменной `imgLoader` присваивается значение `null`, экземпляр класса `ImageLoader` становится доступным для сборки мусора. Тем не менее, до тех пор пока этот экземпляр не будет фактически удален из системной памяти, операция загрузки в экземпляре `ImageLoader`, реализованная на базе функции `setInterval( )`, будет регулярно выполняться.

Это демонстрирует следующий очень простой класс. Он создает экземпляр класса `ImageLoader` и сразу же удаляет ссылку на созданный экземпляр. Но даже после того, как переменной `imgLoader` присвоено значение `null`, сообщение «Now loading image...» продолжает появляться в окне для отладки с интервалом один раз в секунду.

```
package {
  import flash.display.*;

  public class SlideShow extends Sprite {
    private var imgLoader:ImageLoader;
    public function SlideShow ( ) {
      // Создаем экземпляр класса ImageLoader и сразу же удаляем ссылку
      // на созданный экземпляр
      imgLoader = new ImageLoader( );
      imgLoader = null;
    }
  }
}
```

Если объем памяти, который необходим приложению для демонстрации изображений в режиме слайд-шоу, никогда не достигнет уровня, достаточного для запуска цикла сборки мусора, операция загрузки в экземпляре класса `ImageLoader`, реализованная на базе функции `setInterval ( )`, будет выполняться бесконечно долго. Ненужное выполнение кода в «заброшенном» экземпляре класса `ImageLoader` тратит впустую системные и сетевые ресурсы и может привести к появлению нежелательных побочных эффектов в программе.

Чтобы избежать ненужного выполнения кода в «заброшенных» объектах, программа должна всегда деактивировать объекты перед тем, как избавиться от них. *Деактивация* объекта означает его перевод в нерабочее состояние, когда программа больше не может воспользоваться этим объектом для выполнения кода. Например, чтобы деактивировать объект, мы могли бы выполнить одно из перечисленных далее действий или же все действия сразу:

- отменить регистрацию методов объекта для событий;
- остановить все таймеры и интервалы;
- остановить воспроизводящую головку временных шкал (для экземпляров клипов, созданных в среде разработки Flash);
- деактивировать любые объекты, которые станут недостижимыми после того, как сам объект станет недостижимым.

Чтобы объекты могли быть деактивированы, любой класс, экземпляры которого регистрируются для получения событий или используют таймеры, должен предоставлять открытый метод для деактивации своих экземпляров.

Например, наш предыдущий класс `ImageLoader` должен определить метод, останавливающий внутренний интервал. Сейчас добавим подобный метод и назовем его `dispose ( )`. Имя выбрано произвольно; с таким же успехом мы могли бы присвоить данному методу имя `kill ( )`, `destroy ( )`, `die ( )`, `clean ( )`, `disable ( )`, `deactivate ( )` или любое другое имя. Рассмотрим этот код:

```
package {
  import flash.events.*;
  import flash.utils.*;

  public class ImageLoader {
    private var loadInterval:int;

    public function ImageLoader (delay:int = 1000) {
      loadInterval = setInterval(loadImage, delay);
    }

    public function loadImage ( ):void {
      trace("Now loading image...");
      // Код загрузки изображения не приводится
    }

    public function dispose ( ):void {
      clearInterval(loadInterval);
    }
  }
}
```

Любой код, создающий экземпляр класса `ImageLoader`, должен в дальнейшем вызвать метод `ImageLoader.dispose( )` перед тем, как избавиться от этого экземпляра, как показано в следующем коде:

```
package {
  import flash.display.*;

  public class SlideShow extends Sprite {
    private var imgLoader:ImageLoader;
    public function SlideShow ( ) {
      // Создаем и сразу же избавляемся от экземпляра класса ImageLoader
      imgLoader = new ImageLoader( );
      imgLoader.dispose( );
      imgLoader = null;
    }
  }
}
```

Сборка мусора в действии

В листинге 14.1 показана очень простая программа, демонстрирующая сборку мусора в действии. Программа создает объект класса `Sprite`, многократно отображающий сообщение в консоли для отладочной информации. Поскольку объект `Sprite` достижим только через локальную переменную, он становится доступным для сборки мусора сразу после завершения конструктора основного класса программы. При этом в программе также запущен таймер, который многократно создает объекты, занимая системную память. Когда объем потребленной памяти превысит допустимое значение, запустится сборщик мусора. В процессе сборки

мусора исходный объект `Sprite` будет удален из памяти и его сообщения перестанут появляться в консоли для отладочной информации.

Листинг 14.1. Сборка мусора в действии

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.utils.*;
  import flash.events.*;
  import flash.system.*;

  public class GarbageCollectionDemo extends Sprite {
    public function GarbageCollectionDemo ( ) {
      // Данный объект Sprite будет удален в процессе сборки мусора, когда
      // объем потребленной памяти превысит допустимое значение
      var s:Sprite = new Sprite( );
      s.addEventListener(Event.ENTER_FRAME, enterFrameListener);

      // Многократно создает новые объекты, занимая системную память
      var timer:Timer = new Timer(1, 0);
      timer.addEventListener(TimerEvent.TIMER, timerListener);
      timer.start( );
    }

    private function timerListener (e:TimerEvent):void {
      // Создаем объект, чтобы захватить часть системной памяти. Это может
      // быть любой объект, но объекты класса TextField являются довольно
      // объемными.
      new TextField( );
    }

    // Эта функция выполняется до тех пор, пока объект Sprite не будет
    // удален из памяти в процессе сборки мусора
    private function enterFrameListener (e:Event):void {
      // Отображаем объем памяти, занимаемой программой
      trace("System memory used by this program: " + System.totalMemory);
    }
  }
}
```

К задворкам языка ActionScript

Сборка мусора является чрезвычайно важной частью программирования на языке ActionScript. При создании любой программы на языке ActionScript вы должны принимать во внимание вопросы управления памятью. При создании объекта следует решить, нужен ли он на всем протяжении жизни программы. Если это не так, вы должны включить в программу код, который деактивирует этот объект и впоследствии избавляется от него.

Более полную информацию по сборке мусора в языках программирования можно получить в специальной статье на сайте «Википедии» и на сайте The Memory Management Reference по адресу http://www.memorymanagement.org.

Несколько самостоятельно опубликованных статей, написанных Грантом Скиннером (Grant Skinner) и посвященных сборке мусора в языке ActionScript 3.0, можно найти на сайте автора по адресу http://gskinner.com/talks/resource-management/ и http://www.gskinner.com/blog/archives/2006/06/as3_resource_ma.html.

В следующей главе мы рассмотрим менее распространенные инструменты языка ActionScript, предназначенные для изменения структуры классов и объектов на этапе выполнения программы.

Динамические возможности языка ActionScript

Язык ActionScript изначально задумывался как язык, который позволял бы реализовать простейшее программное поведение для содержимого, создававшегося вручную в среде разработки Flash. В ранних версиях ActionScript большая часть кода представляла собой короткие сценарии, которые реализовывали ограниченную функциональность, по сравнению с кодом, необходимым для создания сложного настольного приложения. По существу, первоначальный набор возможностей языка ActionScript отличался гибкостью и простотой.

Изначально язык ActionScript позволял динамически, на этапе выполнения программы, изменять структуры всех классов и даже любых отдельных объектов. Например, на этапе выполнения программа могла:

- добавлять новые методы или переменные экземпляра в любой класс;
- добавлять новые методы или переменные экземпляра в любой отдельно взятый конкретный объект;
- создавать новый класс с нуля;
- изменять суперкласс выбранного класса.

С появлением языка ActionScript 3.0, приложений Flash Player 9, Adobe AIR и Flex, платформа Flash вышла на новый уровень развития, где сложность программы, разработанной на языке ActionScript, может вполне конкурировать со сложностью полнофункционального настольного приложения. Соответственно, ActionScript, как настоящий язык программирования, взял на вооружение многие формальные структуры, необходимые для разработки крупномасштабных приложений, например формальное ключевое слово `class` и синтаксис наследования, формальные типы данных, систему предопределенных событий, обработку исключений и встроенную поддержку формата XML. Тем не менее динамические возможности ActionScript остаются доступными и по-прежнему составляют важную часть внутренней структуры языка.

В этой главе рассматриваются приемы программирования с использованием динамических возможностей языка ActionScript. Однако стоит отметить, что гибкость, присущая такому программированию, ограничивает или полностью лишает преимущества проверки типов, с которым мы познакомились в гл. 8. Как результат, большинство сложных программ используют описанные в этой главе возможности лишь изредка, если вообще используют. Например, из более чем 700 классов, определенных в среде разработки Flex, только около 10 классов используют возможности динамического программирования. С другой стороны, даже если вы никогда не будете использовать динамическое программирование в своем коде, информа-

ция, представленная в этой главе, поможет получить более полное представление о внутренних процессах языка ActionScript.

В качестве нашего первого приема динамического программирования мы рассмотрим динамические переменные экземпляра — переменные, добавляемые в конкретный объект на этапе выполнения программы.

Динамические переменные экземпляра

В самом начале этой книги написание программы на языке ActionScript сравнивалось с проектированием и разработкой самолета. Чертежи самолета сравнивались с классами ActionScript, а реальные детали конкретного физического самолета — с объектами ActionScript. По этой аналогии структура каждого конкретного самолета будет гарантированно совпадать со структурой всех других самолетов, поскольку все самолеты построены по одному и тому же чертежу. Для сравнения отметим, что все экземпляры конкретного класса будут иметь одинаковую структуру, поскольку в их основе лежит один и тот же класс.

Что же произойдет, если владелец одного из самолетов решит закрепить собственный фонарь на верху своего самолета? Этот самолет будет обладать специфической характеристикой, не присущей всем остальным самолетам. В языке ActionScript «добавление нового фонаря на конкретный самолет» аналогично добавлению новой переменной экземпляра в отдельно взятый конкретный объект, при этом в остальные экземпляры класса данного объекта этот «фонарь» не добавляется. Переменная экземпляра такого рода называется *динамической переменной экземпляра*. В сравнении с динамическими переменными экземпляра «обычные» переменные экземпляра называются *фиксированными*.

Динамические переменные экземпляра могут быть добавлены только в экземпляры классов, объявленных с использованием атрибута `dynamic` (подобные классы называются *динамическими*). Динамические переменные экземпляра не могут быть добавлены в экземпляры классов, которые не объявлены с использованием атрибута `dynamic` (подобные классы называются *закрытыми*). Подкласс динамического класса считается динамическим только в том случае, если его определение тоже включает атрибут `dynamic`.

Следующий код создает новый класс `Person`, который может представлять человека в статистической программе, отслеживающей демографическую ситуацию. Поскольку класс `Person` объявлен с использованием атрибута `dynamic`, в любой отдельный объект класса `Person` на этапе выполнения программы можно добавлять динамические переменные экземпляра.

```
dynamic public class Person {
}
```

После того как класс будет объявлен динамическим, мы сможем добавлять новые динамические переменные в любой экземпляр этого класса с помощью стандартного оператора присваивания. Например, следующий код добавляет динамическую переменную экземпляра `eyeColor` в объект класса `Person`:

```
var person:Person = new Person( );
person.eyeColor = "brown";
```

Как уже известно из гл. 8, если бы класс `Person` не был объявлен с использованием атрибута `dynamic`, приведенный код вызвал бы ошибку обращения, поскольку в классе `Person` не определена переменная экземпляра с именем `eyeColor`. Тем не менее в данном случае класс `Person` объявлен *с использованием* атрибута `dynamic`. В результате, когда среда выполнения Flash попытается присвоить значение несуществующей переменной экземпляра `eyeColor`, вместо того чтобы сообщить об ошибке, она просто создаст динамическую переменную экземпляра с именем `eyeColor` и присвоит ей указанное значение (`"brown"`). Обратите внимание, что определение динамической переменной экземпляра в предыдущем коде не содержит и не должно содержать объявление типа и модификатор управления доступом.

Все динамические переменные экземпляра являются нетипизированными и открытыми.

Следующий код, в котором происходит попытка использовать объявление типа при создании переменной `eyeColor`, вызовет ошибку на этапе компиляции:

```
person.eyeColor:String = "brown";  // Ошибка! Использование :String здесь
                                   // недопустимо
```

Для получения значения динамической переменной экземпляра применяется стандартное выражение доступа к переменной, как показано в следующем коде:

```
trace(person.eyeColor);  // Выводит: brown
```

Если указанная переменная экземпляра (динамическая или нединамическая) не существует, среда выполнения Flash вернет значение `undefined`, как показано в следующем коде:

```
trace(person.age);  // Выводит undefined, поскольку объект person
                    // не имеет переменной экземпляра с именем age
```

Наиболее активное использование динамических переменных экземпляра языка ActionScript присуще среде разработки Flash, где каждая анимированная временная шкала представляется подклассом встроенного класса `MovieClip`. В среде разработки Flash автоматически генерируемые подклассы класса `MovieClip` являются динамическими, чтобы программисты могли определять новые переменные в создаваемых вручную экземплярах клипов. Подробное описание этой методики, а также описания других приемов работы с временными шкалами можно найти в гл. 29.

Динамические переменные экземпляра иногда используются для создания простой «справочной таблицы», рассматриваемой далее, в разд. «Использование динамических переменных экземпляра для создания справочных таблиц».

Следует помнить, что возможность динамического изменения программы может привести к проблемам, которые очень сложно выявить. Например, если класс объявлен динамическим, чтобы поддержать динамические переменные экземпляра, настоящие ошибки обращения, возникающие при использовании экземпляров данного класса, могут запросто остаться незамеченными (поскольку обращение

к несуществующей переменной экземпляра не вызовет ошибки ни на этапе компиляции, ни на этапе выполнения программы). Единственный способ узнать наверняка, работает ли динамически изменяемая программа, — это запустить ее и понаблюдать за поведением. Подобное наблюдение отнимает много времени и может приводить к ошибкам, вызванным человеческим фактором, поэтому большинство программистов избегают использования динамических переменных в сложных программах.

Среде выполнения Flash для обращения к динамической переменной экземпляра требуется больше времени, чем для обращения к фиксированной переменной. Когда производительность является решающим фактором, избегайте использования динамических переменных экземпляра.

Обработка динамических переменных экземпляра с помощью циклов for-each-in и for-in. Цикл `for-each-in` обеспечивает простой способ обработки значений динамических переменных экземпляра объекта (или элементов массива). Он имеет следующий общий вид:

```
for each (переменнаяИлиЗначениеЭлемента in некийОбъект) {
  инструкции
}
```

Инструкции *инструкции* цикла `for-each-in` выполняются один раз для каждой динамической переменной экземпляра или элемента массива в объекте *некийОбъект*. В процессе каждой итерации цикла значение переменной или элемент, над которым выполняется итерация *(перечисление)*, присваивается переменной *переменнаяИлиЗначениеЭлемента*. Код внутри тела цикла имеет возможность применять это значение по своему усмотрению.

Например, рассмотрим описание объекта и связанный с ним цикл `for-each-in`, продемонстрированный в следующем коде. Обратите внимание, что внутренний объект `Object` объявлен с использованием атрибута `dynamic` и, следовательно, поддерживает динамические переменные экземпляра.

```
var info:Object = new Object( );
info.city = "Toronto";
info.country = "Canada";

for each (var detail:* in info) {
  trace(detail);
}
```

Приведенный цикл выполняется дважды, по одному разу для каждой из двух динамических переменных экземпляра объекта, на который ссылается переменная `info`. При выполнении цикла в первый раз переменной `detail` присваивается значение `"Toronto"` (то есть значение переменной `city`). При выполнении цикла во второй раз переменная `detail` содержит значение `"Canada"` (то есть значение переменной `country`). Таким образом, в результате выполнения цикла будет выведена следующая информация:

```
Toronto
Canada
```

В большинстве случаев порядок, в котором циклы for-each-in и for-in перечисляют переменные объекта, не гарантирован. Однако существует два исключения: перечисление переменных экземпляров классов XML и XMLList происходит в возрастающем порядке, в соответствии с числовыми именами их переменных (то есть в соответствии с документированным порядком для объектов XML; дополнительную информацию можно найти в гл. 18). Для всех остальных типов объектов порядок перечисления, используемый циклами for-each-in и for-in, может отличаться в зависимости от версий языка ActionScript и клиентских сред выполнения Flash. Таким образом, не следует писать код, который зависит от порядка перечисления циклов for-each-in или for-in, кроме тех случаев, когда обрабатываются данные в формате XML.

Следующий код демонстрирует цикл `for-each-in`, который используется для обращения к значениям элементов массива:

```
var games:Array = ["Project Gotham Racing",
                   "Shadow of the Colossus",
                   "Legend of Zelda"];

for each (var game:* in games) {
  trace(game);
}
```

Приведенный цикл выполняется трижды, по одному разу для каждого из трех элементов массива `games`. При выполнении цикла в первый раз переменной `game` присваивается значение `"Project Gotham Racing"` (то есть значение первого элемента). При выполнении цикла во второй раз переменная `game` принимает значение `"Shadow of the Colossus"`, а на третий раз — значение `"Legend of Zelda"`. Таким образом, выводимая информация выглядит следующим образом:

```
Project Gotham Racing
Shadow of the Colossus
Legend of Zelda
```

Цикл `for-each-in` является напарником цикла `for-in` языка ActionScript. Тогда как цикл `for-each-in` перечисляет *значения* переменных, цикл `for-in` — *имена* переменных. Например, следующий цикл `for-in` перечисляет имена динамических переменных экземпляра объекта, на который ссылается переменная `info`:

```
for (var detailName:* in info) {
  trace(detailName);
}
// Вывод:
// city
// country
```

Обратите внимание, что предыдущий код выводит *имена* переменных `city` и `country`, а не их значения. Для обращения к значениям этих свойств мы могли бы использовать оператор `[]`, который рассматривается далее, в разд. «Динамические обращения к переменным и методам». Это демонстрирует следующий код:

```
for (var detailName:* in info) {
  trace(info[detailName]);
}
// Вывод:
```

```
// Toronto
// Canada
```

Чтобы исключить перечисление динамической переменной экземпляра в циклах `for-in` и `for-each-in`, используется метод `setPropertyIsEnumerable( )` класса `Object`, показанный в следующем коде:

```
info.setPropertyIsEnumerable("city", false);

for (var detailName:* in info) {
  trace(info[detailName]);
}
// Выводит: Canada
// (переменная "city" не была обработана в цикле for-in)
```

Мы рассмотрим применение цикла `for-each-in` на практическом примере в разд. «Использование динамических переменных экземпляра для создания справочных таблиц».

Динамическое добавление нового поведения в экземпляр

Когда вы познакомитесь с тем, как создавать динамические переменные экземпляра, у вас может возникнуть вопрос, поддерживает ли язык ActionScript и динамические методы экземпляра, то есть добавление нового метода экземпляра в один конкретный объект без добавления этого метода в любые другие экземпляры класса данного объекта. Фактически формальных способов для динамического добавления настоящего метода экземпляра в объект не существует. Тем не менее, присвоив замыкание функции динамической переменной экземпляра, мы можем имитировать эффект добавления нового метода в отдельный объект (чтобы освежить в памяти смысл фразы *«замыкание функции»*, обратитесь к гл. 5). Следующий код демонстрирует общий подход:

```
некийОбъект.некаяДинамическаяПеременная = некаяФункция;
```

В приведенном коде *некийОбъект* — это экземпляр динамического класса, *некаяДинамическаяПеременная* — имя динамической переменной экземпляра (которая может являться либо новой, либо существующей переменной), а *некаяФункция* — ссылка на замыкание функции. Обычно функция *некаяФункция* задается с помощью литерала функции, как показано в следующем коде:

```
некийОбъект.некаяДинамическаяПеременная = function (параметр1, параметр2...
параметрn) {
  // Тело функции
}
```

После того как функция *некаяФункция* будет присвоена динамической переменной экземпляра, она может быть вызвана через этот объект точно так же, как и обычный метод экземпляра, что показано в следующем коде:

```
некийОбъект.некаяДинамическаяПеременная(значение1, значение2... значениеn);
```

Чтобы продемонстрировать использование предыдущего общего синтаксиса, вернемся к примеру с переменной info из предыдущего раздела:

```
var info:Object = new Object( );
info.city = "Toronto";
info.country = "Canada";
```

Предположим, мы хотим добавить в объект, на который ссылается переменная info, новое поведение — возможность форматировать и отображать собственное содержимое в виде строки. Мы создадим новую переменную экземпляра getAddress, которой присвоим желаемую функцию форматирования, как показано в следующем коде:

```
info.getAddress = function ( ):String {
  return this.city + ", " + this.country;
}
```

Для вызова этой функции используется следующий код:

```
trace(info.getAddress( ));
```

Обратите внимание, что внутри тела функции, присвоенной переменной getAddress, мы можем использовать ключевое слово this для обращения к переменным и методам объекта, через который вызывается эта функция. На самом деле в случае с замыканиями функций обратиться без использования ключевого слова this к переменным и методам объекта, через который вызывается эта функция, невозможно. Предположим, мы убрали ключевое слово this из описания функции getAddress( ), как показано в следующем коде:

```
info.getAddress = function ( ):String {
  return city + ", " + country;
}
```

При поиске переменных city и country среда выполнения Flash не рассматривает автоматически переменные экземпляра объекта, на который ссылается переменная info. Следовательно, предыдущий код вызовет ошибку (если выше в цепочке видимости функции getAddress( ) не существует других переменных с именами city и country).

Если функция переменной экземпляра объекта присваивается внутри класса данного объекта, то она сможет обращаться к переменным и методам объекта, объявленным с использованием модификаторов управления доступом private, protected, internal и public. Если присваивание происходит внутри подкласса класса объекта, функция сможет обращаться только к тем переменным и методам объекта, которые объявлены с использованием модификаторов управления доступом protected, internal и public. Если присваивание происходит внутри пакета, в котором объявлен класс объекта, функция сможет обращаться только к переменным и методам объекта, объявленным с использованием модификаторов управления доступом internal и public. Если класс объекта объявлен в одном пакете, а присваивание происходит в другом пакете, функция сможет обращаться только к переменным и методам объекта, объявленным с использованием модификатора управления доступом public.

Например, рассмотрим следующий код, который создает динамический класс Employee:

```
dynamic public class Employee {
  public var startDate:Date;
  private var age:int;
}
```

Следующий код присваивает функцию динамической переменной экземпляра doReport экземпляра класса Employee. Присваивание происходит за пределами класса Employee, но внутри класса, который объявлен в том же пакете, что и класс Employee. Следовательно, функция может обращаться только к тем переменным объекта Employee, которые объявлены с использованием модификаторов управления доступом internal и public. Переменные, объявленные с использованием модификаторов управления доступом protected и private, недоступны для этой функции.

```
public class Report {
  public function Report (employee:Employee) {
    // Присваиваем функцию переменной doReport
    employee.doReport = function ( ):void {
      trace(this.startDate);  // Обращение к public-переменной допустимо
      trace(this.age);        // Обращение к private-переменной запрещено
    }
  }
}
```

Динамические обращения к переменным и методам

Поскольку имена динамических переменных экземпляра зачастую неизвестны вплоть до этапа выполнения программы, язык ActionScript предоставляет возможность указывать имя переменной с помощью обычного строкового выражения. Следующий код демонстрирует общий подход:

некийОбъект[*некоеВыражение*]

В предыдущем коде *некийОбъект* — это ссылка на объект, к переменной которого происходит обращение, а *некоеВыражение* — любое выражение, возвращающее строку (которая обозначает имя переменной). Предыдущий код может применяться как для присваивания значения переменной, так и для получения значения.

Например, следующий код присваивает значение "Toronto" переменной, имя которой указывается с помощью выражения строкового литерала "city":

```
var info:Object = new Object( );
info["city"] = "Toronto";
```

Следующий код присваивает значение "Canada" переменной, имя которой указывается с помощью выражения строкового литерала "country":

```
info["country"] = "Canada";
```

Следующий код получает значение переменной, имя которой указывается с помощью выражения-идентификатора `detail`:

```
var detail:String = "city";
trace(info[detail]);  // Выводит: Toronto
```

Когда среда выполнения Flash встречает код `info[detail]`, она сначала определяет значение переменной `detail` (в нашем случае это `"city"`), после чего ищет переменную с именем `"city"` в объекте, на который ссылается переменная `info`.

Синтаксические правила, применяемые к идентификаторам, не распространяются на переменные, которые создаются с использованием оператора `[]`. Например, следующий код создает динамическую переменную экземпляра, имя которой начинается с цифры:

```
var info:Object = new Object( );
info["411"] = "Information Line";
```

Использование оператора «точка» (`.`) для создания той же переменной вызовет ошибку, поскольку такая запись нарушает синтаксические правила, применяемые к идентификаторам:

```
var info:Object = new Object( );
info.411 = "Information Line";  // ОШИБКА! Идентификаторы не должны
                                // начинаться с цифры
```

Стоит отметить, что описанная методика может использоваться для обращения не только к динамическим переменным экземпляра, но и к любым типам переменных и методов. Однако наиболее часто она применяется при работе именно с динамическими переменными экземпляра. Следующий раздел демонстрирует использование динамических обращений в практической ситуации: для создания справочной таблицы.

Использование динамических переменных экземпляра для создания справочных таблиц

Справочная таблица — это структура данных, которая связывает набор имен с соответствующим набором значений. Например, следующий псевдокод демонстрирует справочную таблицу, которая представляет меню (приводится по-русски):

```
закуска:        маисовые чипсы
основное блюдо: лепешка с начинкой из бобов
десерт:         пирожное
```

Чтобы представить данную справочную таблицу с помощью динамических переменных экземпляра, можно использовать следующий код:

```
var meal:Object = new Object( );
meal.appetizer  = "tortilla chips";
meal.maincourse = "bean burrito";
meal.dessert    = "cake";
```

Теперь рассмотрим более сложный сценарий. Представьте приложение для инвентаризации книг в книжном магазине, которое позволяет пользователю искать книги по номеру ISBN. Информация о каждой книге загружается с внешнего сервера. Чтобы минимизировать количество обращений к серверу, приложение загружает за раз информацию о 500 книгах. Для упрощения будем полагать, что информация о каждой книге представлена в виде отдельной строки, имеющей следующий формат:

```
"Price: $19.99. Title: Path of the Paddle"
```

Для хранения загруженной информации о книге в программе на языке ActionScript создадим экземпляр класса `Object`, который будет служить справочной таблицей для книг:

```
var bookList:Object = new Object( );
```

Загруженную информацию о каждой книге мы присваиваем новой динамической переменной экземпляра созданного объекта `bookList`. Имя каждой переменной соответствует ISBN-номеру книги с предшествующей строкой `"isbn"`. Например, переменная для книги с ISBN-номером 155209328X будет называться `isbn155209328X`. Следующий код демонстрирует создание динамической переменной экземпляра для данной книги в том случае, если бы мы заранее знали ее ISBN-номер:

```
bookList.isbn155209328X = "Price: $19.95. Title: Path of the Paddle";
```

В реальном приложении тем не менее ISBN-номер книги станет известен только после того, как информация об этой книге будет загружена с сервера. Следовательно, имя динамической переменной экземпляра для каждой книги должно формироваться динамически, на основании загружаемых данных. Для демонстрационных целей создадим переменную `bookData`, значение которой будет представлять данные в том виде, в котором они были бы загружены с сервера. В этом упрощенном примере ISBN-номер и подробная информация о каждой книге разделяются одиночным символом «тильда» (~). При этом полная информация о каждой книге отделяется двойным символом «тильда» (~~).

```
var bookData:String = "155209328X~Price: $19.95. Title: Path of the Paddle"
                    + "~~"
                    + "0072231726~Price: $24.95. Title: High Score!";
```

Чтобы преобразовать загруженные данные о книгах из строки в массив книг для обработки, воспользуемся методом `split( )` класса `String`, как показано в следующем коде:

```
var bookDataArray:Array = bookData.split("~~");
```

Для преобразования массива книг в справочную таблицу используем следующий код:

```
// Создаем переменную, которая будет хранить информацию о каждой книге
// в процессе обработки
var book:Array;

// Выполнить цикл один раз для каждого элемента в массиве книг
for (var i:int = 0; i < bookDataArray.length; i++) {
```

```
  // Преобразуем текущий элемент массива из строки в собственный массив.
  // Например, строка:
  // "155209328X~Price: $19.95. Title: Path of the Paddle"
  // становится массивом:
  // ["155209328X", "Price: $19.95. Title: Path of the Paddle"]
  book = bookDataArray[i].split("~");

  // Создаем динамическую переменную экземпляра, имя которой соответствует
  // ISBN-номеру текущего элемента в массиве книг, и присваиваем этой
  // переменной описание текущего элемента в массиве. Обратите внимание, что
  // ISBN-номер представлен выражением book[0], а описание –
  // выражением book[1].
  bookList["isbn" + book[0]] = book[1];
}
```

Когда все 500 книг будут добавлены в объект `bookList` и каждая из них будет храниться в своей собственной динамической переменной экземпляра, пользователь сможет выбирать книгу для просмотра, вводя ее ISBN-номер в текстовое поле `isbnInput`. Вот как бы мы отображали информацию о выбранной пользователем книге в процессе отладки:

```
trace(bookList["isbn" + isbnInput.text]);
```

Следующим же образом мы бы выводили информацию о выбранной пользователем книге на экране в текстовом поле, на которое ссылается переменная `bookDescription`:

```
bookDescription.text = bookList["isbn" + isbnInput.text];
```

Для отображения списка всех книг, хранящихся в объекте `bookList`, можно использовать цикл `for-each-in`, как показано в следующем коде:

```
for each (var bookInfo:* in bookList) {
  // Выводим значение динамической переменной экземпляра,
  // обрабатываемой в текущий момент
  trace(bookInfo);
}
```

В результате выполнения цикла будет выведена следующая отладочная информация:

```
Price: $19.95. Title: Path of the Paddle
Price: $24.95. Title: High Score!
```

Создание справочных таблиц с помощью литералов объекта. Для удобства справочные таблицы, содержимое которых имеет фиксированный размер и известно заранее, можно создавать с помощью литерала объекта. Литерал объекта создает новый экземпляр класса `Object` из набора пар «имя/значение», которые представляют динамические переменные экземпляра, разделены запятыми и заключены в фигурные скобки. Вот общий синтаксис:

```
{имяПеременной1:значениеПеременной1,
 имяПеременной2:значениеПеременной2,
 ...
 имяПеременнойN:значениеПеременнойN}
```

Например, следующий код создает экземпляр класса `Object` с динамической переменной экземпляра `city` (значением которой является `"Toronto"`) и динамической переменной экземпляра `country` (значением является `"Canada"`):

```
var info:Object = {city:"Toronto", country:"Canada"};
```

Данный код идентичен следующему:

```
var info:Object = new Object( );
info.city = "Toronto";
info.country = "Canada";
```

Если бы в приложении для инвентаризации из предыдущего раздела было всего две книги, мы могли бы использовать следующий литерал объекта для создания справочной таблицы `bookList`:

```
var bookList:Object = {
                isbn155209328X:"Price: $19.95. Title: Path of the Paddle",
                isbn0072231726:"Price: $24.95. Title: High Score!"
                };
```

Использование функций для создания объектов

В процессе чтения этой книги вы видели, что большинство объектов в языке ActionScript создается с помощью классов. Тем не менее существует возможность создавать объекты с использованием независимых замыканий функций. Следующий код демонстрирует общий подход. В нем для примера объявлена функция `Employee( )`, которая применяется для создания объекта:

```
// Создаем функцию
function Employee ( ) {
}
// Используем функцию для создания объекта и присваиваем этот объект
// переменной worker
var worker = new Employee( );
```

Обратите внимание, что переменная `worker` является нетипизированной. С точки зрения типов данных объект, на который ссылается переменная `worker`, представляет собой экземпляр класса `Object`. Класс `Employee` не существует, поэтому не существует и тип данных `Employee`. Таким образом, следующий код вызовет ошибку (поскольку тип данных `Employee` не существует):

```
// ОШИБКА!
var worker:Employee = new Employee( );
```

Замыкание функции, используемое для создания объекта, называется *функцией-конструктором* (не путайте с *методом*-конструктором, который является частью определения класса). В языке ActionScript 3.0 независимые функции, объявленные на уровне пакета, не могут применяться в качестве функций-конструкторов, поэтому предыдущий код должен быть размещен внутри метода, в коде за пределами описания пакета или в сценарии кадра на временной шкале в среде разработки Flash. Тем не менее ради краткости в этом разделе все функции-конструкторы

показаны без необходимых методов или сценариев кадров, которые должны содержать эти функции.

Все объекты, создаваемые из функций-конструкторов, неявно являются динамическими. Таким образом, в процессе создания объекта функция-конструктор может использовать ключевое слово `this` для добавления в этот объект новых динамических переменных экземпляра. Динамическим переменным экземпляра, создаваемым в функции-конструкторе, обычно присваиваются значения, которые передаются в функцию в качестве аргументов. Это демонстрирует следующий код:

```
function Employee (age, salary) {
  // Описываем динамические переменные экземпляра
  this.age = age;
  this.salary = salary;
}

// Передаем аргументы, которые будут использованы в качестве значений
// динамических переменных экземпляра данного объекта
var worker = new Employee(25, 27000);
trace(worker.age);  // Выводит: 25
```

Чтобы объекты, создаваемые с помощью определенной функции-конструктора, могли разделять между собой информацию и поведение, язык ActionScript для каждой функции вводит специальную статическую переменную `prototype`. Переменная функции `prototype` ссылается на объект (называемый *объектом-прототипом* функции), к динамическим переменным экземпляра которого можно обращаться через любой объект, созданный с помощью данной функции. Изначально язык ActionScript присваивает переменной `prototype` каждой функции экземпляр базового класса `Object`. Добавляя в этот объект динамические переменные экземпляра, мы можем создать информацию и поведение, разделяемые между всеми объектами, созданными из конкретной функции.

Например, следующий код добавляет динамическую переменную экземпляра `company` в объект `prototype` функции `Employee ( )`:

```
Employee.prototype.company = "AnyCorp";
```

В результате любой объект, созданный из функции `Employee ( )`, может обращаться к переменной `company` так, будто это его собственная динамическая переменная экземпляра:

```
var worker = new Employee(25, 27000);
trace(worker.company);  // Выводит: AnyCorp
```

В предыдущем коде, когда среда выполнения Flash поймет, что объект, созданный из функции `Employee ( )` (`worker`), не имеет переменной экземпляра или метода экземпляра с именем `"company"`, она проверит, определена ли динамическая переменная экземпляра с таким именем в объекте `Employee.prototype`. В этом объекте такая переменная определена, поэтому среда Flash использует ее так, будто это собственная переменная объекта `worker`.

Если, с другой стороны, в объекте `worker` определена собственная переменная `company`, она будет использована вместо переменной объекта `Employee.prototype`. Это демонстрирует следующий код:

```
var worker = new Employee(25, 27000);
worker.company = "CarCompany";
trace(worker.company);  // Выводит: CarCompany (не AnyCorp)
```

Используя методику, рассмотренную в разд. «Динамическое добавление нового поведения в экземпляр», мы можем присвоить функцию динамической переменной экземпляра объекта `prototype` любой функции-конструктора. Эта функция в дальнейшем может быть использована любым объектом, созданным из данной функции-конструктора.

Например, следующий код определяет динамическую переменную экземпляра `getBonus` в объекте `prototype` функции `Employee( )` и присваивает этой переменной функцию, которая подсчитывает и возвращает премию по итогам года:

```
Employee.prototype.getBonus = function (percentage:int):Number {
  // Возвращает размер премии в зависимости от зарплаты сотрудника
  // и указанного процента
  return this.salary * (percentage/100);
}
```

В результате все объекты, созданные из функции `Employee( )`, могут использовать функцию `getBonus( )` так, будто она была присвоена их собственным динамическим переменным экземпляра:

```
var worker = new Employee(25, 27000);
trace(worker.getBonus(10));  // Выводит: 2700
```

Использование объектов-прототипов для дополнения классов

Как уже было сказано, язык ActionScript определяет для каждой функции специальную статическую переменную `prototype`. Используя переменную `prototype` функции, мы можем разделять информацию и поведение между всеми объектами, созданными из этой функции.

Подобно тому, как язык ActionScript определяет переменную `prototype` для каждой функции, он также определяет статическую переменную `prototype` для каждого класса. Используя статическую переменную `prototype`, мы можем добавлять разделяемую между всеми экземплярами данного класса информацию и поведение на этапе выполнения программы.

Например, следующий код определяет новую динамическую переменную экземпляра `isEmpty` в объекте `prototype` внутреннего класса `String` и присваивает ей функцию. Эта функция возвращает значение `true`, когда строка не содержит символов; в противном случае функция возвращает значение `false`:

```
String.prototype.isEmpty = function ( ) {
  return (this == "") ? true : false;
};
```

Для вызова функции `isEmpty( )` над объектом `String` мы используем следующий код:

```
var s1:String = "Hello World";
var s2:String = "";

trace(s1.isEmpty( ));  // Выводит: false
trace(s2.isEmpty( ));  // Выводит: true
```

Тем не менее в предыдущем примере кода — и вообще во всей этой методике — существует проблема: динамическая переменная экземпляра добавляется только на этапе выполнения программы; следовательно, компилятор не имеет ни малейшего представления о том, что эта переменная существует, и сгенерирует ошибку, если компиляция программы будет выполняться в строгом режиме. Например, при компиляции программы в строгом режиме код из предыдущего примера вызовет следующую ошибку:

```
Call to a possibly undefined method isEmpty through a reference with static
type String.
```

По-русски она будет звучать так: **Вызов возможно неопределенного метода isEmpty через ссылку на статический тип String.**

Чтобы обращение к функции `isEmpty( )` при использовании строгого режима не приводило к ошибке на этапе компиляции, мы должны применять динамическое обращение, как показано в следующем коде:

```
s1["isEmpty"]( )
```

С другой стороны, если бы класс `String` не был объявлен с использованием атрибута `dynamic`, первоначальный подход (то есть `s1.isEmpty( )`) не вызывал *бы ошибку.*

Стоит отметить, что фиксированные переменные и методы всегда имеют преимущество перед переменными прототипа. В предыдущем примере, если в классе `String` уже определен метод или переменная экземпляра с именем `isEmpty`, то все обращения к свойству `isEmpty` будут относиться к этой переменной или к методу экземпляра, а не к динамической переменной экземпляра объекта-прототипа класса `String`.

Цепочка прототипов

Из предыдущих разделов мы узнали, что объект-прототип может использоваться для разделения информации и поведения между объектами, созданными из определенной функции-конструктора или класса. Фактически обращаться к данному объекту-прототипу можно и за пределами объектов, созданных из функции или класса, которому принадлежит данный прототип.

В случае класса к динамической переменной экземпляра, определенной в объекте-прототипе данного класса, можно обращаться не только через экземпляры этого класса, но и через экземпляры его потомков. Это демонстрирует следующий простой код:

```
// Создаем простейший класс A
dynamic public class A {
```

```
}

// Создаем другой простейший класс B, который расширяет класс A
dynamic public class B extends A {
}

// Создаем основной класс приложения
public class Main extends Sprite {
  public function Main ( ) {
    // Добавляем динамическую переменную экземпляра в объект-прототип
    // класса A
    A.prototype.day = "Monday";

    // Обращаемся к переменной A.prototype.day через экземпляр класса B
    var b:B = new B( );
    trace(b.day);  // Выводит: "Monday"
  }
}
```

В случае функции обращаться к динамическим переменным экземпляра, определенным в объекте-прототипе функции, можно не только через любой объект, созданный из этой функции, но и через любой объект, *цепочка прототипов* которого включает объект-прототип данной функции.

Рассмотрим, как работают цепочки прототипов. Предположим, что мы создаем функцию `Employee ( )`, как делали это раньше, в объекте-прототипе которой определяем динамическую переменную экземпляра `company`:

```
function Employee ( ) {
}
Employee.prototype.company = "AnyCorp";
```

Любой объект, созданный из функции `Employee ( )`, может обращаться к переменной `company` через объект-прототип функции `Empoyee ( )`. Ничего нового. Теперь предположим, что мы создали еще одну функцию `Manager ( )`:

```
function Manager ( ) {
}
```

Предположим также, что необходимо сделать так, чтобы объекты, созданные из функции `Manager ( )`, могли обращаться к переменной `company` через объект-прототип функции `Employee ( )`. Для этого мы присваиваем объект, созданный из функции `Employee ( )`, переменной `prototype` функции `Manager ( )`.

```
Manager.prototype = new Employee( );
```

Теперь рассмотрим, что произойдет, если мы обратимся к переменной с именем `"company"` через объект, созданный из функции `Manager ( )`, как показано в следующем коде:

```
var worker = new Manager( );
trace(worker.company);
```

При выполнении предыдущего кода среда Flash проверяет, имеет ли объект `worker` переменную или метод экземпляра с именем `"company"`. Объект `worker` не имеет

переменной или метода экземпляра с таким именем, поэтому далее среда выполнения Flash проверяет, определена ли динамическая переменная экземпляра с именем "company" в объекте-прототипе функции Manager( ). Сам по себе объект-прототип функции Manager( ) является объектом, созданным из функции Employee( ). Тем не менее в объектах, созданных из функции Employee( ), динамическая переменная экземпляра с именем "company" не определена. Следовательно, после этого среда Flash проверяет, определена ли динамическая переменная экземпляра с именем "company" в объекте-прототипе функции Employee( ). В нем такая переменная определена, поэтому среда выполнения Flash использует ее так, будто это собственная переменная объекта worker.

Рассмотрим путь, который проходит среда Flash при поиске переменной с именем "company".

1. Ищем переменную company в объекте worker. Не найдена.
2. Ищем переменную company в объекте Manager.prototype. Не найдена.
3. Объект Manager.prototype создан из функции Employee( ), поэтому ищем переменную company в объекте Employee.prototype. Найдена!

Список объектов-прототипов, просматриваемых средой выполнения Flash при попытке определить значение переменной, называется *цепочкой прототипов*. До появления языка ActionScript 3.0 цепочка прототипов была основным механизмом для разделения повторно используемого поведения между различными типами объектов. В языке ActionScript 3.0 эту роль играет наследование классов.

Обратите внимание на следующие ограничения, налагаемые на цепочки прототипов в ActionScript 3.0.

- Объект, присваиваемый переменной prototype функции, сам по себе должен являться объектом, созданным из функции, или экземпляром класса Object (использование экземпляров других классов не допускается).
- Значение переменной prototype класса присваивается средой выполнения Flash автоматически и в дальнейшем не может быть изменено.

Вперед!

В большинстве средних и крупномасштабных проектов приемы динамического программирования, рассмотренные в этой главе, играют лишь второстепенную роль. Тем не менее понимание его возможностей в ActionScript позволит вам чувствовать себя более комфортно при работе с этим языком. Подобным образом знание концепции *области видимости*, которая рассматривается в следующей главе, добавит вам уверенности в себе как программисту на языке ActionScript. Область видимости управляет доступностью и продолжительностью жизни определений в программе.

Область видимости

Область видимости — это физическая область программы, в которой выполняется код. В языке ActionScript существует пять возможных областей видимости:

- тело функции;
- тело метода экземпляра;
- тело статического метода;
- тело класса;
- все остальное (то есть *глобальная область видимости*).

В любой конкретный момент выполнения программы доступность переменных, функций, классов, интерфейсов и пространств имен определяется областью видимости кода, исполняемого в текущий момент. Например, код внутри функции может обращаться к локальным переменным этой функции, поскольку он выполняется внутри ее области видимости. Напротив, код за пределами функции не может обращаться к локальным переменным данной функции, поскольку он выполняется за пределами ее области видимости.

В ActionScript области видимости могут быть вложенными. Например, функция может быть вложена в метод экземпляра, который, в свою очередь, вложен в тело класса:

```
public class SomeClass {
  public function someMethod ( ):void {
    function someNestedFunction ( ):void {
      // Область видимости данной функции вложена внутрь области видимости
      // метода someMethod( ), которая вложена в область видимости класса
      // SomeClass
    }
  }
}
```

Если одна область видимости вложена в другую область, определения (то есть переменные, функции, классы, интерфейсы и пространства имен), доступные во внешней области видимости, становятся доступными во вложенной области. Например, функция, вложенная внутрь метода экземпляра, может обращаться к локальным переменным этого метода. Полный список вложенных областей видимости, окружающих код, выполняемый в текущий момент, называется *цепочкой областей видимости*.

В этой главе рассматривается доступность переменных, функций, классов, интерфейсов и пространств имен внутри различных областей видимости языка ActionScript.

Обратите внимание, что помимо «доступных» определений, перечисленных в каждом из последующих разделов, определения, объявленные во внешнем пакете с использованием модификатора управления доступом `public`, можно сделать видимыми и в данной области видимости с помощью директивы `import`. Подробную информацию можно найти в подразд. «Пример создания объекта: добавление животного в зоопарк» разд. «Создание объектов» гл. 1.

Местоположение определения и используемые при его объявлении модификаторы управления доступом совместно определяют доступность данного определения в программе. Для информации в табл. 16.1 приведена доступность определений в соответствии с их местоположением и применяемым модификатором управления доступом.

Таблица 16.1. Доступность определения в зависимости от его местоположения и используемого модификатора управления доступом

Определение	Доступно
Определение за пределами всех пакетов	Только внутри исходного файла, содержащего данное определение
Определение в безымянном пакете	В любой точке программы
Определение с использованием модификатора управления доступом public в именованном пакете	Внутри пакета, содержащего данное определение, и в любом месте, где оно импортируется
Определение с использованием модификатора управления доступом internal в именованном пакете	Только внутри пакета, содержащего данное определение
Метод или переменная, объявленные с использованием модификатора управления доступом public	В любом месте, где доступен класс, содержащий данное определение
Метод или переменная, объявленные с использованием модификатора управления доступом internal	Внутри пакета, содержащего класс с данным определением
Метод или переменная, объявленные с использованием модификатора управления доступом protected	Внутри класса, содержащего данное определение, и его классов-потомков
Метод или переменная, объявленные с использованием модификатора управления доступом private	Только внутри класса, содержащего данное определение
Определение в методе экземпляра, статическом методе или функции	Доступно внутри метода или функции, содержащих данное определение, а также во всех вложенных в них функциях

Глобальная область видимости

Код, помещенный непосредственно в тело пакета или на верхний уровень тела пакета, находится в глобальной области видимости. Иначе говоря:

```
package {
  // Этот код находится в глобальной области видимости
}
// Этот код также находится в глобальной области видимости
```

Код, находящийся в глобальной области видимости, может обращаться к функциям, переменным, классам, интерфейсам и пространствам имен, определенным:

- на верхнем уровне безымянного пакета;
- за пределами всех пакетов, но в том же исходном файле (AS).

Другими словами:

```
package {
  // Размещенные здесь определения доступны всему коду
  // в глобальной области видимости
}
// Размещенные здесь определения доступны всему коду в том же исходном файле
```

Стоит отметить, что код, размещенный на верхнем уровне тела именованного пакета, может также обращаться к определениям, размещенным на верхнем уровне того же пакета. Эти определения доступны потому, что внутри именованного пакета язык ActionScript автоматически открывает пространство имен, связанное с данным пакетом (более подробную информацию можно найти в гл. 17). Иными словами:

```
package somePackage {
  // Размещенные здесь определения становятся автоматически доступны
  // всему коду в пакете somePackage
}
```

Область видимости класса

Код, помещенный на верхний уровень тела класса, находится в области видимости этого класса. Например:

```
package {
  public class SomeClass {
    // Размещенный здесь код находится в области видимости класса someClass
  }
}
```

Помните, что код, размещенный на верхнем уровне тела класса, включается в автоматически создаваемый статический метод (инициализатор класса), который выполняется всякий раз, когда среда Flash определяет класс на этапе выполнения программы. Подробную информацию можно найти в разд. «Статические методы» гл. 4.

Через цепочку областей видимости код в области видимости класса может обращаться к следующим определениям:

- ко всем определениям, доступным коду в глобальной области видимости;
- статическим методам и статическим переменным, определенным в этом классе;
- статическим методам и статическим переменным, определенным в предках данного класса, если они существуют (то есть в суперклассе, суперклассе суперкласса и т. д.).

Иначе говоря:

```
package {
  public class SomeClass extends SomeParentClass {
    // Определенные здесь статические переменные
    // и статические методы доступны
    // в любом месте класса SomeClass
  }
}

package {
  public class SomeParentClass {
    // Определенные здесь статические переменные
    // и статические методы доступны
    // в любом месте класса SomeClass
  }
}
```

Хотя класс может обращаться к статическим переменным и методам своих предков, следует помнить, что статические переменные и методы не наследуются. Дополнительную информацию можно найти в разд. «Пример наследования» гл. 6.

Область видимости статического метода

Код, помещенный в тело статического метода, находится в области видимости данного метода. Это демонстрирует следующий код:

```
package {
  public class SomeClass {
    public static function staticMeth ( ) {
      // Размещенный здесь код находится в области видимости метода
      // staticMeth
    }
  }
}
```

Через цепочку областей видимости код в области видимости статического метода может обращаться к этим определениям:

- ко всем определениям, доступным коду в глобальной области видимости;
- ко всем определениям, доступным коду в области видимости класса, который содержит определение данного статического метода.

Кроме того, код в области видимости статического метода может обращаться ко всем локальным переменным, вложенным функциям и пространствам имен, определенным внутри данного статического метода.

Другими словами:

```
package {
  public class SomeClass extends SomeParentClass {
    public static function staticMeth ( ) {
      // Определенные здесь локальные переменные, вложенные функции
```

```
      // и пространства имен доступны в методе staticMeth
    }
  }
}
```

Область видимости метода экземпляра

Код, помещенный в тело метода экземпляра, находится в области видимости данного метода:

```
package {
  public class SomeClass {
    public function instanceMeth ( ) {
      // Размещенный здесь код находится в области видимости
      // метода instanceMeth
    }
  }
}
```

Через цепочку областей видимости код в области видимости метода экземпляра может обращаться к следующим определениям:

- ко всем определениям, доступным коду в глобальной области видимости;
- ко всем определениям, доступным коду в области видимости класса, который содержит определение данного метода экземпляра;
- ко всем методам и переменным экземпляра объекта, через который был вызван данный метод экземпляра (с учетом ограничений, налагаемых используемыми модификаторами управления доступом);
- ко всем локальным переменным, вложенным функциям и пространствам имен, определенным внутри данного метода экземпляра.

Это демонстрирует следующий код:

```
package {
  public class SomeClass extends SomeParentClass {
    public function instanceMeth ( ) {
      // Все методы и переменные экземпляра текущего объекта (то есть this)
      // доступны в методе instanceMeth( ) (с учетом ограничений, налагаемых
      // используемыми модификаторами управления доступом).
      // Определенные здесь локальные переменные, вложенные функции
      // и пространства имен доступны в методе instanceMeth( )
    }
  }
}
```

Область видимости функции

Код, добавленный в тело функции, находится в области видимости данной функции. Конкретный список определений, доступных коду, находящемуся в области видимости функции, зависит от местоположения этой функции в программе.

Код в функции, которая определена на уровне пакета или за пределами всех пакетов, может обращаться к следующим определениям:

- ко всем определениям, доступным коду в глобальной области видимости;
- ко всем локальным переменным, вложенным функциям и пространствам имен, определенным внутри этой функции.

Код в функции, которая определена внутри статического метода, может обращаться к таким определениям:

- ко всем определениям, доступным коду в глобальной области видимости;
- ко всем определениям, доступным коду в области видимости данного статического метода;
- ко всем локальным переменным, вложенным функциям и пространствам имен, определенным внутри этой функции.

Код в функции, определенной внутри метода экземпляра, может обращаться к следующим определениям:

- ко всем определениям, доступным коду в глобальной области видимости;
- ко всем определениям, доступным коду в области видимости данного метода экземпляра;
- ко всем локальным переменным, вложенным функциям и пространствам имен, определенным внутри этой функции.

Код в функции, которая определена внутри другой функции, может обращаться к таким определениям:

- ко всем определениям, доступным коду в глобальной области видимости;
- ко всем определениям, доступным коду в области видимости внешней функции;
- ко всем локальным переменным, вложенным функциям и пространствам имен, определенным внутри этой функции.

Обзор областей видимости

Приведенный далее код демонстрирует все возможные области видимости языка ActionScript:

```
package {
  // Размещенный здесь код находится в глобальной области видимости

  public class SomeClass {
    // Этот код находится в области видимости класса SomeClass

    public static function staticMeth ( ):void {
      // Размещенный здесь код находится в области видимости метода
      // staticMeth
    }

    public function instanceMeth ( ):void {
      // Этот код находится в области видимости метода instanceMeth
```

```
      function nestedFunc ( ):void {
        // Размещенный здесь код находится в области видимости функции
        // nestedFunc
      }
    }
  }
}
// Этот код находится в глобальной области видимости
```

Детали реализации

Для осуществления хранения информации об определениях в цепочке областей видимости среда Flash использует список объектов. Информацию об определениях для каждой области видимости хранят следующие объекты.

Глобальная область видимости — глобальный объект (объект, автоматически создаваемый средой выполнения Flash для хранения глобальных определений).

Область видимости класса — объект `Class` данного класса (и объекты `Class` его предков).

Область видимости статического метода — объект `Class` данного класса (и объекты `Class` его предков).

Область видимости метода экземпляра — текущий объект (`this`) и объект активации (*объект активации* — это объект, создаваемый средой выполнения Flash и хранящийся в ней, который включает локальные переменные и параметры функции или метода).

Область видимости функции — объект активации.

Когда среда выполнения Flash встречает в программе выражение-идентификатор, она выполняет поиск этого идентификатора среди объектов в цепочке областей видимости. Например, рассмотрим следующий код:

```
package {
  public class SomeClass {
    public function instanceMeth ( ):void {
      function nestedFunc ( ):void {
        trace(a);
      }
    }
  }
}
var a:int = 15;
```

Когда среда Flash встречает идентификатор `a`, она выполняет поиск его значения в объекте активации функции `nestedFunc ( )`. Однако в функции `nestedFunc ( )` не определено никаких локальных переменных или параметров с именем `a`, поэтому далее среда Flash продолжает поиск идентификатора `a` в текущем объекте (то есть в объекте, через который был вызван метод `instanceMeth ( )`). Но класс `SomeClass` не определяет и не наследует метод или переменную экземпляра

с именем a, поэтому после этого Flash продолжает поиск идентификатора a в классе объекта SomeClass. В классе SomeClass отсутствует определение статического метода или статической переменной с именем a, поэтому далее среда выполнения Flash продолжает поиск идентификатора a в объекте суперкласса класса SomeClass, которым является класс Object. Однако в классе Object отсутствует определение статического метода или статической переменной с именем a, поэтому после этого среда Flash продолжает поиск идентификатора a в глобальном объекте. В нем Flash находит идентификатор a и определяет, что значением этого идентификатора является 15. Получив значение идентификатора a, среда выполнения выводит число 15 в процессе отладки. Довольно много действий, чтобы найти маленькую a!

Вот объекты, среди которых среда выполнения Flash пыталась найти идентификатор a, перечисленные в порядке поиска:

- объект активации функции nestedFunc( );
- объект, через который был вызван метод instanceMeth( );
- объект класса SomeClass;
- объект класса Object;
- глобальный объект.

Если бы идентификатор a не был найден в глобальном объекте, среда выполнения Flash сообщила бы об ошибке обращения.

Переменные, объявленные за пределами описания пакета, доступны только внутри исходного файла, содержащего эти переменные.

Теперь, когда все известно про цепочку областей видимости, завершим эту главу, рассмотрев единственный инструмент языка ActionScript, который позволяет непосредственно управлять цепочкой областей видимости, — оператор with.

Расширение цепочки областей видимости с помощью оператора with

Оператор with предоставляет сокращенный способ обращения к переменным и методам объекта, который исключает необходимость повторно указывать имя объекта. Этот оператор имеет следующий обобщенный вид:

```
with (объект) {
  вложенныеИнструкции
}
```

Когда обращение к идентификатору происходит внутри блока оператора with, на наличие указанного имени проверяется объект *объект* до того, как будет проверена оставшаяся часть цепочки областей видимости. Другими словами, оператор with временно добавляет объект *объект* в конец внутреннего списка объектов, составляющих цепочку областей видимости среды выполнения Flash.

Например, чтобы обратиться к переменной `PI` внутреннего класса `Math`, мы обычно используем следующий код:

```
Math.PI;
```

Однако с помощью оператора `with` мы можем обратиться к переменной `PI` внутреннего класса `Math` без предварительного обращения к этому классу:

```
with (Math) {  // Выполняем инструкции в контексте класса Math
  trace(PI);   // Выводит: 3.1459... (поскольку переменная PI определена
               // в классе Math)
}
```

Некоторые разработчики находят оператор `with` полезным, когда создается код, в котором приходится часто обращаться к переменным и методам определенного объекта.

К пространствам имен

В этой главе мы узнали, как язык ActionScript управляет доступностью определений в различных областях видимости. В следующей главе мы научимся использовать пространства имен для управления видимостью определений. Стоит отметить, что пространства имен являются важной частью внутренней архитектуры языка ActionScript, но в пользовательском коде они обычно применяются только в сложных ситуациях.

Пространства имен

В самых общих чертах, *пространство имен* — это набор имен, не содержащий дубликатов. Иными словами, внутри этого набора каждое имя является уникальным. Например, в английском языке названия фруктов могут считаться пространством имен, поскольку каждый фрукт имеет уникальное имя: apple (яблоко), pear (груша), orange (апельсин) и т. д. Подобным образом пространством имен могут считаться названия цветов, поскольку каждый цвет обладает своим уникальным именем: blue (синий), green (зеленый), orange (оранжевый) и т. д.

Обратите внимание, что имя orange встречается в обеих группах имен. Имя orange само по себе не является уникальным — оно уникально только внутри каждой группы. В зависимости от того, идет разговор о фрукте или цвете, одно и то же имя orange может обозначать два различных предмета. В этом и заключается смысл пространств имен. Они позволяют одному и тому же имени (идентификатору) иметь различные значения в зависимости от контекста, в котором оно используется.

Что касается программирования, данная особенность пространств имен — «то же имя — другое значение» — обладает двумя общими преимуществами:

- позволяет программистам избежать конфликтов имен;
- дает возможность программе адаптировать свое поведение в соответствии с текущим контекстом.

В процессе изучения этой главы мы рассмотрим множество тонкостей, связанных с пространствами имен в языке ActionScript. Попытайтесь разобраться в теме так, чтобы различные подробности не уводили вас в сторону от относительной простоты пространств имен. По существу, пространства имен — это не более чем система именований, состоящая из двух частей. Они используются для того, чтобы отличать одну группу имен от другой, подобно тому как код города позволяет отличать один номер телефона от других телефонных номеров по всему миру.

Словарь пространств имен

В этой главе мы встретимся с несколькими новыми терминами, относящимися к пространствам имен. Некоторые наиболее важные из них приведены далее в виде краткого справочника. Сейчас бегло ознакомьтесь с этим списком, чтобы получить общее представление, а в дальнейшем возвращайтесь к нему всякий раз, когда вам понадобится памятка при изучении последующих разделов. В оставшейся части этой главы каждый из приведенных терминов рассматривается гораздо более подробно.

Локальное имя — локальная часть уточненного идентификатора, то есть имя, которое уточняется пространством имен. Например, `orange` в `fruit::orange`.

Название пространства имен — уникальное идентифицирующее название пространства имен, представленное в виде унифицированного идентификатора ресурса (URI — Uniform Resource Identifier). В языке ActionScript название пространства имен доступно через переменную экземпляра `uri` класса `Namespace`. В XML название пространства имен доступно через атрибут `xmlns`.

Префикс пространства имен — псевдоним названия пространства имен. Префиксы пространств имен используются только в языке XML, однако в ActionScript для упрощения операций расширения E4X к ним можно обращаться через переменную `prefix` класса `Namespace`.

Идентификатор пространства имен — идентификатор, используемый при определении пространства имен в языке ActionScript. Например, в следующем определении пространства имен `fruit` является идентификатором:

```
namespace fruit = "http://www.example.com/games/kidsgame/fruit";
```

Открытое пространство имен — пространство имен, которое было добавлено в набор открытых пространств имен с помощью директивы `use namespace`.

Открытые пространства имен — набор пространств имен, просматриваемый компилятором при попытке разрешить неуточненные ссылки.

Уточняющее пространство имен — пространство имен, которое квалифицирует определение переменной или метода либо идентификатор пространства имен как уточненный идентификатор.

Уточненный идентификатор — идентификатор языка ActionScript, состоящий из двух частей и включающий идентификатор пространства имен и локальное имя, разделенные двумя двоеточиями. Например, `fruit::orange`.

Пространства имен в языке ActionScript

В языке ActionScript пространство имен — это описатель для имени переменной, метода, XML-тега или XML-атрибута. *Описатель* ограничивает, или уточняет, значение идентификатора, позволяя нам сказать в коде, что «переменная `orange` является фруктом, а не цветом», или «метод `search( )` применяется для поиска в японском языке, а не в английском», или «тег `<TABLE>` описывает разметку HTML-страницы, а не часть мебели».

Используя пространства имен в языке ActionScript для уточнения имен переменных и методов, мы можем выполнить следующее.

- Предотвратить конфликты именования (подробную информацию можно найти в разд. «Пространства имен для модификаторов управления доступом»).
- Реализовать пользовательские уровни видимости методов и переменных во всей программе, независимо от структуры пакетов программы (обратитесь к примеру

пространства имен `mx_internal`, рассмотренному в подразд. «Пример: управление видимостью на уровне прикладной среды» разд. «Практические примеры использования пространств имен»).

- Реализовать контроль доступа, основанный на разрешениях, когда классы должны запрашивать доступ к переменным или методам (обратитесь к листингу 17.5 в подразд. «Пример: управление доступом на основании разрешений» разд. «Практические примеры использования пространств имен»).
- Реализовать различные «режимы» в программе (обратитесь к листингу 17.7 в подразд. «Пример: реализация режимов работы программы» разд. «Практические примеры использования пространств имен»).

Кроме того, пространства имен в языке ActionScript предоставляют непосредственный доступ к пространствам имен языка XML в XML-документах. Эта тема рассматривается в разд. «Использование пространств имен XML» гл. 18.

Пространства имен в языке C++ в сравнении с пространствами имен в языке ActionScript. Хотя отчасти синтаксис пространств имен в ActionScript похож на синтаксис пространств имен в языке C++, в ActionScript пространства имен используются иначе, нежели в C++.

В языке C++ пространство имен является синтаксическим контейнером, подобно пакетам в языках ActionScript и Java. Здесь идентификатор считается находящимся *в* конкретном пространстве имен только в том случае, если он физически находится в операторе блока этого пространства имен. Например, в следующем коде переменная `a` находится в пространстве имен `n`, поскольку физическое размещение объявления переменной находится внутри блока пространства имен:

```
namespace n {
  int a;
}
```

Таким образом, пространства имен в языке C++ в основном используются для того, чтобы предотвратить конфликты именования между различными частями кода и запретить одной части кода обращаться к другой части кода.

В отличие от этого в языке ActionScript пространство имен может включать любую переменную или метод, независимо от физической структуры кода. Пространства имен в языке ActionScript определяют правила видимости для методов и переменных, которые выходят за структурные границы (за границы классов и пакетов) программы.

Программисты на C++, пытающиеся отыскать в ActionScript эквивалент пространств имен языка C++, должны рассмотреть возможность использования пакетов ActionScript, которые описаны в гл. 1. В C++ не существует непосредственных аналогов пространств имен языка ActionScript.

Перед тем как перейти к рассмотрению примеров применения пространств имен, познакомимся с основными концепциями и синтаксисом, необходимыми при использовании пространств имен в ActionScript. В следующих нескольких вводных разделах мы создадим два пространства имен — `fruit` и `color`, после чего ис-

пользуем их для уточнения определений двух переменных и, наконец, обратимся к этим переменным с помощью так называемых уточненных идентификаторов. Изложенный материал мы будем рассматривать на примере простого приложения: детской игры «Учусь читать». Начнем.

Создание пространств имен

Чтобы создать пространство имен, мы должны присвоить ему название. Название каждого пространства имен, формально именуемое *названием пространства имен*, — это строка, которая по соглашению представляет унифицированный идентификатор ресурса, или URI. Идентификатор URI однозначно идентифицирует пространство имен среди остальных пространств имен в программе и потенциально даже в любой другой программе по всему миру.

Термин URI относится к обобщенному стандарту идентификации ресурсов, подвидом которого является известный стандарт адресации в Интернете — URL. Спецификацию стандарта можно найти по адресу http://www.ietf.org/rfc/rfc2396.txt.

Использование идентификаторов URI в качестве названий пространств имен в языке ActionScript основано на стандарте, установленном консорциумом World Wide Web (W3C) в их рекомендации «Namespaces in XML». Текст документа доступен по адресу http://www.w3.org/TR/xml-names11.

Первым шагом на пути создания пространства имен является выбор идентификатора URI для названия будущего пространства.

Выбор идентификатора URI для пространства имен

Обычно идентификатором URI, используемым в качестве названия пространства имен, является указатель URL, который находится в управлении организации, создающей данный код. Например, моим сайтом является www.moock.org, поэтому для нового имени пространства имен я мог бы использовать идентификатор URI, имеющий следующую структуру:

`http://www.moock.org/`*`приложение/пространство_имен`*

Мы воспользуемся следующими идентификаторами URI для пространств имен `fruit` и `color`, которые будут применены в создаваемой нами детской игре:

```
http://www.example.com/games/kidsgame/fruit
http://www.example.com/games/kidsgame/color
```

Стоит отметить, что идентификатор URI не обязан — а зачастую так и происходит — существовать в действительности. Он применяется лишь для идентификации пространства имен. Это не адрес веб-страницы, XML-документа или любого другого сетевого ресурса. Можно использовать любой идентификатор URI, однако применение в качестве названия пространства имен указателя URL со своего собственного сайта позволяет минимизировать вероятность того, что другие организации будут использовать такое же название.

Определение пространства имен

Теперь, когда мы выбрали идентификатор URI, который будет использован в качестве названия нашего пространства имен, мы создаем пространство имен с помощью ключевого слова `namespace`, за которым следует идентификатор пространства имен, знак = и, наконец, само название пространства имен (идентификатор URI):

```
namespace идентификатор = URI;
```

Идентификатор пространства имен — это имя константы языка ActionScript, которой присваивается значение пространства имен (значением пространства имен является экземпляр класса `Namespace`, генерируемый в результате выполнения предыдущей инструкции). *URI* — пространство имен.

Например, чтобы создать пространство имен с идентификатором `fruit` и URI `"http://www.example.com/games/kidsgame/fruit"`, мы используем:

```
namespace fruit = "http://www.example.com/games/kidsgame/fruit";
```

Чтобы создать пространство имен с идентификатором `color` и URI `"http://www.example.com/games/kidsgame/color"`, мы применяем:

```
namespace color = "http://www.example.com/games/kidsgame/color";
```

Обратите внимание на отсутствие объявления типа — в данном случае его использование не допускается. Неявным типом данных идентификатора пространства имен всегда является `Namespace`. Пространства имен могут определяться везде, где могут определяться переменные, а именно:

- на верхнем уровне определения пакета;
- на верхнем уровне определения класса;
- в функции или методе;
- на временной шкале клипа в FLA-файле.

Фактически пространства имен почти всегда определяются на верхнем уровне определения пакета или класса (если они не используются для XML-документов); дополнительную информацию можно найти в гл. 18. Пока мы будем определять все наши пространства имен на уровне пакета.

Чтобы создать пространство имен на уровне пакета, которое можно использовать в любом месте программы, поместите его определение в отдельный файл с расширением AS, имя которого полностью совпадает с именем идентификатора пространства имен, как показано в следующем примере для пространства имен `fruit`:

```
// Файл fruit.as
package kidsgame {
  namespace fruit = "http://www.example.com/games/kidsgame/fruit";
}
```

и для пространства имен `color`:

```
// Файл color.as
package kidsgame {
  namespace color = "http://www.example.com/games/kidsgame/color";
}
```

Далее, в разд. «Доступность пространств имен», мы рассмотрим примеры пространств имен, определенных на уровне класса или функции. Пространства имен, определенные на временной шкале клипа, рассматриваются так, будто они были определены на уровне класса в классе, который представляет клип, содержащий данные пространства имен (более подробную информацию об определениях на уровне временной шкалы можно найти в разд. «Класс документа» гл. 29).

Явные идентификаторы URI в сравнении с неявными

До сих пор все наши определения пространств имен включали явный идентификатор URI, аналогичный тому, который выделен полужирным шрифтом в следующем определении пространства имен:

```
namespace fruit = "http://www.example.com/games/kidsgame/fruit";
```

Но когда объявление пространства имен явно не включает название пространства (URI), это название генерируется средой выполнения автоматически. Например, следующее определение пространства имен не включает идентификатор URI, поэтому среда автоматически создает его:

```
package {
  namespace ns1;
}
```

Чтобы доказать это, мы можем отобразить автоматически сгенерированный идентификатор URI для пространства имен `ns1`, как показано ниже:

```
namespace ns1;
trace(ns1.uri);  // Выводит: ns1
```

Метод экземпляра `toString( )` класса `Namespace` также возвращает значение переменной экземпляра `uri`, так что вызов функции `trace( )` можно сократить до:

```
trace(ns1);  // Также выводит: ns1
```

Однако при определении пространств имен вообще целесообразно указывать идентификатор URI, поскольку явные URI можно легко идентифицировать в любом контексте, даже между несколькими SWF-файлами, в то время как автоматически генерируемые URI таким свойством не обладают. В данном разделе мы будем указывать явные идентификаторы URI для всех пространств имен.

Хорошей практикой является включение идентификатора URI в определения любых пространств имен.

Обзор терминологии по пространствам имен

Всего на нескольких коротких страницах текста мы встретились с множеством новых терминов. Рассмотрим их.

Название пространства имен — это идентификатор URI, который идентифицирует пространство имен.

Класс Namespace представляет пространство имен в объектно-ориентированном виде.

Значение пространства имен — это экземпляр класса `Namespace`.

Идентификатор пространства имен — это имя константы языка ActionScript, которая ссылается на значение пространства имен.

Вообще говоря, сложность этих терминов можно охватить одним простым названием — «пространство имен». Например, в этой книге мы часто используем простую фразу «пространство имен `fruit`» вместо более точной с технической точки зрения фразы «пространство имен `"http://www.example.com/games/kidsgame/fruit"`, представленное объектом `Namespace`, на который ссылается идентификатор `fruit`».

Чтобы упростить чтение этой книги, мы обычно будем использовать более простую и менее точную фразу «пространство имен `fruit`». Тем не менее отличие между названием пространства имен и идентификатором пространства имен является важным для некоторых последующих обсуждений, поэтому вы должны познакомиться с приведенной выше терминологией.

Использование пространств имен для уточнения определений переменных и методов

Теперь, когда мы определили пространства имен `fruit` и `color`, мы можем использовать их для указания так называемого *уточняющего пространства имен* для новых методов и переменных. Уточняющее пространство имен — это пространство имен, внутри которого имя переменной или метода является уникальным.

Чтобы указать уточняющее пространство имен для новой переменной или метода, используем идентификатор этого пространства имен в качестве атрибута определения переменной или метода, как показано в следующем примере:

```
// Указываем уточняющее пространство имен для переменной
идентификаторПространстваИмен var имяСвойства:типДанных =
необязательноеИсходноеЗначение;

// Указываем уточняющее пространство имен для метода
идентификаторПространстваИмен function имяМетода (параметры):ВозвращаемыйТип {
  инструкции
}
```

Следующие правила применяются к идентификатору *идентификаторПространстваИмен*.

- Он должен быть доступен в области видимости, в которой определены переменная или метод, как рассматривается далее, в разд. «Доступность пространств имен».
- Идентификатор не может быть литеральным значением; в частности, строковым литералом, содержащим название пространства имен (URI).
- Он должен быть константой на этапе компиляции.

Эти три правила в действительности означают, что идентификатор *идентификаторПространстваИмен* может являться только идентификатором пространства имен, созданного с помощью ключевого слова `namespace`, и, в частности, не может быть переменной, которая ссылается на значение пространства имен (такие переменные будут рассмотрены далее, в разд. «Присваивание и передача значений пространств имен»).

Вот как можно указать уточняющее пространство имен `fruit` для переменной экземпляра с именем `orange`:

```
fruit var orange:String = "Round citrus fruit";
```

Следующим образом можно указать уточняющее пространство имен `color` для переменной экземпляра с таким же именем `orange`:

```
color var orange:String = "Color obtained by mixing red and yellow";
```

Допустимо и распространено использование одного уточняющего пространства имен для уточнения множества различных переменных, при котором подчеркивается тот факт, что имя каждой переменной уникально внутри данного пространства имен (в этом и заключается вся идея!). Например, рассмотрим другую переменную — `purple`, которая также уточняется пространством имен `color`:

```
color var purple:String = "Color obtained by mixing red and blue";
```

Когда несколько переменных и методов уточняются одним и тем же пространством имен `n`, можно считать, что они формируют логическую группу. С теоретической точки зрения естественно полагать, что переменная или метод «принадлежит» или «находится внутри» пространства имен, в котором они объявлены. Однако на техническом уровне «пространство имен» языка ActionScript не является структурой данных, которая физически хранит переменные или методы. Пространства имен *не* являются ни контейнерами данных, ни массивами. Они служат лишь для уточнения имен. Чтобы избежать путаницы, с этого момента мы будем использовать фразу «пространство имен `n` уточняет имя переменной `p`» вместо фразы «переменная с именем `p` принадлежит пространству имен `n`» или «переменная с именем `p` находится внутри пространства имен `n`».

Пространства имен не содержат имена; они просто уточняют их.

Обратите внимание, что для одного определения переменной или метода нельзя указать несколько пространств имен. Каждое определение может включать только одно уточняющее пространство имен. Например, следующий код недопустим:

```
// Попытка указать два пространства имен для одного определения.
fruit color var orange:String;  // Вызывает следующую ошибку:
                                // Only one namespace attribute
                                // may be used per definition
                                // (В определении может указываться только
                                // одно пространство имен)
```

Атрибуты пространств имен, определенных пользователем, допустимы только на верхнем уровне описания класса. В предыдущем разделе было рассказано, как использовать собственные пространства имен в качестве атрибутов при определении методов и переменных. На самом деле это единственное место, где допустимо применять пространства имен, заданные пользователем, в качестве атрибута определения.

Пространства имен, определенные пользователем, можно применять в качестве атрибутов только на верхнем уровне описания класса.

Если вы попытаетесь использовать пространство имен, заданное пользователем, в качестве атрибута определения в любом другом месте, возникнет следующая ошибка:

```
A user-defined namespace attribute can only be used at the top level of a
class definition.
```

По-русски она будет выглядеть следующим образом: **Атрибут пространства имен, определенного пользователем, может быть использован только на верхнем уровне описания класса.**

В частности, это значит, что вы не можете указать пространство имен, определенное пользователем, при определении класса, переменной на уровне пакета, функции на уровне пакета, локальной переменной или вложенной функции. Следующие определения являются недопустимыми:

```
// Недопустимое определение класса. Здесь не допускается использовать
// пространство имен color!
color class Picker {
}

public function doSomething ( ):void {
  // Недопустимое определение локальной переменной. Здесь не допускается
  // использовать пространство имен color!
  color var tempSwatch;
}

package p {
  // Недопустимое определение переменной на уровне пакета. Здесь
  // не допускается использовать пространство имен color!
  color var favorites:Array;
}
```

Напротив, внутренние пространства имен языка ActionScript могут быть использованы в качестве атрибутов определения в любом месте, где это допускается языком ActionScript. Например, как будет рассказано далее, в разд. «Пространства имен для модификаторов управления доступом», модификаторы управления доступом (`public`, `internal`, `protected` и `private`) являются внутренними пространствами имен, а два из них (`public` и `internal`) могут быть использованы на уровне пакета.

Итак, вернемся к нашему коду. Мы уже знаем, как создать два пространства имен и три переменные, уточненные этими пространствами, — это показано в следующем коде:

```
// Создаем два пространства имен
package kidsgame {
  namespace fruit = "http://www.example.com/games/kidsgame/fruit";
}

package kidsgame {
  namespace color = "http://www.example.com/games/kidsgame/color";
}

// В любом месте внутри класса создаем три переменные
fruit var orange:String = "Round citrus fruit";
color var orange:String = "Color obtained by mixing red and yellow";
color var purple:String = "Color obtained by mixing red and blue";
```

Далее мы рассмотрим, как обращаться к этим переменным с помощью *уточненных идентификаторов*.

Уточненные идентификаторы

До сих пор все идентификаторы, с которыми мы встречались в этой книге, были так называемыми *простыми идентификаторами* — имеющими «простые» однокомпонентные имена, например `box`, `height` и `border`. Но для того, чтобы работать с пространствами имен, мы должны использовать *уточненные идентификаторы*. Уточненный идентификатор — это особый вид идентификатора, который включает не только имя, но и пространство имен, уточняющее это имя. Соответственно уточненные идентификаторы состоят из двух частей, а не из одной.

- *Локальное имя* — имя, которое является уникальным внутри указанного пространства имен.
- *Уточняющее пространство имен* — пространство имен, внутри которого данное локальное имя является уникальным.

В коде на языке ActionScript уточненные идентификаторы записываются следующим образом:

уточняющееПространствоИмен::*локальноеИмя*

Здесь локальное имя и уточняющее пространство имен объединяются вместе с помощью *оператора уточнителя имени*, записываемого в виде двух двоеточий (`::`). Первая часть выражения — *уточняющееПространствоИмен* — должна быть представлена либо идентификатором пространства имен, либо переменной, значением которой является пространство имен. О том, как присваивать пространства имен переменным, будет рассказано далее, в разд. «Присваивание и передача значений пространств имен». Тем не менее часть *уточняющееПространствоИмен* не может быть представлена строковым литералом, являющимся названием пространства имен (URI).

Рассмотрим реальный пример использования уточненного идентификатора. Во-первых, вспомните наше предыдущее определение переменной `orange`, уточненное пространством имен `fruit`:

```
fruit var orange:String = "Round citrus fruit";
```

Вот как мы обращаемся к этой переменной с помощью уточненного идентификатора:

```
fruit::orange
```

Аналогично этому рассмотрим уточненный идентификатор для переменной с локальным именем `orange`, которое уточнено пространством имен `color`:

```
color::orange
```

Обратите внимание, что в предыдущих примерах локальные имена являются одинаковыми (`orange`), но отличаются уточняющие их пространства имен. Язык ActionScript использует уточняющее пространство имен для проведения различия между двумя локальными именами. В языке ActionScript уточненные идентификаторы используются точно так же, как и простые; они просто содержат уточняющее пространство имен. Формат *уточняющееПространствоИмен*`::`*локальноеИмя* применяется одинаково как к именам методов, так и к именам переменных:

```
someNamespace::p    // Обращение к переменной p
someNamespace::m( ) // Вызов метода m( )
```

Для обращения к уточненному идентификатору через объект применяется знакомый нам оператор «точка», как показано в следующем коде:

некийОбъект`.`*уточняющееПространствоИмен*`::`*локальноеИмя*

Например, этот код обращается к переменной `p` объекта `someObj`, которая уточняется пространством имен `someNamespace`:

```
someObj.someNamespace::p
```

Следующий код вызывает метод `m( )` объекта `someObj`, который уточняется пространством имен `someNamespace`:

```
someObj.someNamespace::m( )
```

Расширенные имена. Как уже было сказано, уточненный идентификатор является двухкомпонентным именем, состоящим из уточняющего пространства имен и локального имени:

уточняющееПространствоИмен`::`*локальноеИмя*

Часть *уточняющееПространствоИмен* в уточненном идентификаторе — это ссылка на объект `Namespace`, переменная `uri` которого определяет название пространства имен.

Для сравнения отметим, что расширенное имя представляет собой двухкомпонентное имя, состоящее из литерала названия пространства имен (URI) и локального имени. Расширенные имена используются только в документации — но не в коде — и обычно записываются в следующем виде:

`{`*названиеПространстваИмен*`}`*локальноеИмя*

Например, снова рассмотрим определение пространства имен `fruit`:

```
namespace fruit = "http://www.example.com/games/kidsgame/fruit";
```

Теперь рассмотрим определение переменной `orange`, уточняемой пространством имен `fruit`:

```
fruit var orange:String = "Round citrus fruit";
```

В коде мы обращаемся к этой переменной с помощью уточненного идентификатора `fruit::orange`. Однако в документации мы могли бы обсуждать эту переменную, ссылаясь на действительное название соответствующего пространства имен, а не на ее идентификатор. Это можно сделать с помощью следующего *расширенного имени*:

```
{http://www.example.com/games/kidsgame/fruit}orange
```

В этой книге расширенные имена применяются редко, однако они достаточно распространены в документации по пространствам имен XML. Если вы не используете пространства имен XML, просто знайте, что синтаксис *{названиеПространстваИмен}локальноеИмя* является только соглашением по документированию, но не поддерживаемым вариантом записи кода.

Функциональный пример использования пространств имен

Воспользуемся нашими новыми знаниями о пространствах имен для создания простейшей программы. В листинге 17.1 представлены базовые элементы приложения, которое мы будем разрабатывать на протяжении следующих разделов, — детской игры на распознавание слов. В ней игроку демонстрируется картинка с изображением цвета или фрукта и предлагается выбрать его название из списка вариантов. На рис. 17.1 показаны две различные картинки игры.

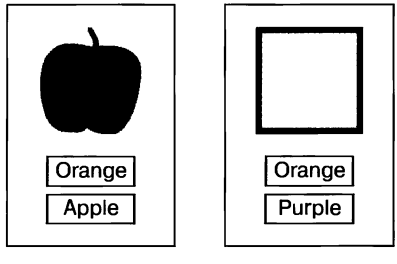

Рис. 17.1. Детская игра «Учусь читать»

Пока каждый предмет в игре будет представлен переменной, значением которой является строковое описание. Мы определим набор всех переменных-предметов в классе с именем `Items`. С помощью пространств имен мы отделим переменные-«фрукты» от переменных-«цветов»; названия переменных-«фруктов» будут уточняться пространством имен `fruit`, а названия переменных-«цветов» — пространством имен `color`.

Взглянем на код в листинге 17.1, после чего рассмотрим его более подробно.

Листинг 17.1. Детская игра: функциональный пример использования пространств имен

```
// Файл fruit.as
package {
  namespace fruit = "http://www.example.com/games/kidsgame/fruit";
```

```
}
// Файл color.as
package {
  namespace color = "http://www.example.com/games/kidsgame/color";
}

// Файл Items.as
package {
  public class Items {
    fruit var orange:String = "Round citrus fruit";
    color var orange:String = "Color obtained by mixing red and yellow";

    public function Items ( ) {
      trace(fruit::orange);
      trace(color::orange);
    }
  }
}
```

Предыдущий код начинается с определения пространств имен нашей игры. Пространство `fruit` определяется в файле `fruit.as`, как показано в следующем коде:

```
package {
  namespace fruit = "http://www.example.com/games/kidsgame/fruit";
}
```

После этого мы определяем пространство имен `color` в файле `color.as`, как показано в следующем коде:

```
package {
  namespace color = "http://www.example.com/games/kidsgame/color";
}
```

Оба пространства имен определены на уровне пакета, поэтому обращаться к ним может любой класс нашего приложения.

Затем мы определяем класс `Items` в файле `Items.as`. В классе `Items` определены две переменные, каждая из которых имеет локальное имя `orange`. Для первой переменной мы указываем уточняющее пространство имен `fruit`; для второй — уточняющее пространство имен `color`.

```
package {
  public class Items {
    fruit var orange:String = "Round citrus fruit";
    color var orange:String = "Color obtained by mixing red and yellow";
  }
}
```

Наконец, чтобы убедиться, что наш код работает, в методе-конструкторе класса `Items` мы используем функцию `trace( )` для отображения значений обеих переменных `orange`. Чтобы отличать одну переменную `orange` от другой, используются уточненные идентификаторы `fruit::orange` и `color::orange`.

```
package {
  public class Items {
```

```
    fruit var orange:String = "Round citrus fruit";
    color var orange:String = "Color obtained by mixing red and yellow";

    public function Items ( ) {
      trace(fruit::orange);  // Выводит: Round citrus fruit
      trace(color::orange);  // Выводит: Color obtained by
                             // mixing red and yellow
    }
  }
}
```

Попробуйте догадаться, что бы произошло, если бы мы изменили предыдущий конструктор класса `Items` и обращались к простому идентификатору `orange`, без применения уточняющего пространства имен, как показано в следующем коде:

```
public function Items ( ) {
  trace(orange); // Что произойдет здесь?
}
```

Ответ: возникнет следующая ошибка на этапе компиляции программы:

```
Access of undefined property orange.
```

На русском языке она будет звучать так: **Обращение к несуществующему свойству orange.**

Компилятор не может найти переменную или метод (то есть «свойство») по имени `orange`, поскольку в области видимости метода-конструктора `Items` не существует переменной или метода с простым идентификатором `orange`. Переменные `fruit::orange` и `color::orange` уточняются с помощью пространств имен, поэтому они оказываются недоступны при попытке обратиться к ним с применением неуточненного идентификатора. И все-таки далее, в разд. «Открытые пространства имен и директива use namespace», мы познакомимся с упрощенным способом обращения к уточненным идентификаторам без использования уточняющего пространства имен.

Очевидно, что в листинге 17.1 не показана полнофункциональная игра, но этот код призван дать вам представление о базовом синтаксисе и использовании пространств имен. Мы завершим создание нашей игры далее в этой главе.

На этом начальном этапе разработки нашей игры вы могли бы поинтересоваться, почему вместо использования пространств имен мы не можем просто определить переменные с длинными именами, например `orangeFruit` и `orangeColor`. Или почему нельзя разделить оба вида oranges, присвоив их двум отдельным массивам, как показано в следующем коде:

```
var fruitList:Array = ["orange", "apple", "banana"];
var colorList:Array = ["orange", "red", "blue"];
```

Описанные варианты имеют право на жизнь. Фактически, учитывая выбранный уровень упрощений, нам действительно было бы лучше использовать массивы или более длинные имена переменных. Но не стоит пока терять веру в пространства имен; мы собираемся строить системы с более сложными сценариями использования.

Доступность пространств имен

Как и в случае с определениями переменных и методов, определения пространств имен могут быть изменены с помощью модификаторов управления доступом `public`, `internal`, `protected` и `private`. Местоположение определения пространства имен в сочетании с модификатором управления доступом этого определения указывают, где может быть использован результирующий идентификатор пространства имен.

Вот несколько основных правил, которые помогут решить, где разместить ваши пространства имен.

- Если пространство имен должно быть доступно в любом месте программы или в группе классов, определяйте его на уровне пакета.
- Если вам требуется пространство имен, которое задает видимость переменных и методов внутри одного класса, определяйте его на уровне класса.
- Если пространство имен будет применяться лишь иногда внутри функции и вы знаете идентификатор URI этого пространства имен, но не можете обращаться к нему напрямую, определяйте пространство имен на уровне функции.

Рассмотрим несколько примеров, начав с пространств имен, задаваемых на уровне пакета.

Доступность определений пространств имен на уровне пакета

В следующем коде мы определяем идентификатор пространства имен `fruit` в пакете `kidsgame`:

```
package kidsgame {
  public namespace fruit = "http://www.example.com/games/kidsgame/fruit";
}
```

Поскольку идентификатор `fruit` объявлен на уровне пакета с использованием модификатора управления доступом `public`, он может быть применен для уточнения любой переменной или метода в данной программе. Безусловно, код за пределами пакета `kidsgame` должен импортировать пространство имен `fruit` перед его использованием, как показано в следующем примере:

```
package anyPackage {
  // Импортируем пространство имен fruit
  import kidsgame.fruit;

  public class AnyClass {
    // Здесь можно применять пространство имен fruit, поскольку оно уже
    // было импортировано
    fruit var banana:String = "Long yellow fruit";
  }
}
```

Теперь сделаем так, чтобы доступность пространства имен `color` была ограничена одним пакетом, используя модификатор управления доступом `internal`:

```
package kidsgame {
  internal namespace color = "http://www.example.com/games/kidsgame/color";
}
```

Когда идентификатор пространства имен определяется с применением модификатора управления доступом `internal` на уровне пакета, он может быть использован только внутри пакета, в котором определен. Это демонстрирует следующий код.

```
package kidsgame {
  public class Items {
    // Здесь можно применять пространство имен color. Использование
    // пространства имен color допустимо, поскольку оно происходит внутри
    // пакета kidsgame
    color var green:String = "Color obtained by mixing blue and yellow";
  }
}

package cardgame {
  import kidsgame.color;
  public class CardGame {
    // Недопустимо.
    // Пространство имен color может быть использовано
    // только внутри пакета kidsgame.
    color var purple:String = "Color obtained by mixing blue and red";
  }
}
```

При определении пространств имен на уровне пакета могут применяться модификаторы управления доступом `public` и `internal`, при этом использование модификаторов управления доступом `private` и `protected` не допускается. Более того, если в определении пространства имен на уровне пакета модификатор управления доступом опускается, применяется модификатор управления доступом `internal`. Например, следующий код:

```
package kidsgame {
  // internal указывается явно
  internal namespace fruit;
}
```

аналогичен данному коду:

```
package kidsgame {
  // internal подразумевается неявно
  namespace fruit;
}
```

Теперь рассмотрим определения пространств имен на уровне класса.

Доступность определений пространств имен на уровне класса

Идентификатор пространства имен, определенный в классе с использованием модификатора управления доступом `private`, доступен только внутри данного класса и недоступен в его подклассах и в любом другом внешнем коде:

```
public class A {
  private namespace n = "http://www.example.com/n";
  // Отлично. Идентификатор пространства имен n доступен в этом месте.
  n var someVariable:int;
}

public class B extends A {
  // Ошибка. Идентификатор пространства имен n недоступен в этом месте.
  n var someOtherVariable:int;
}
```

Мы можем использовать пространство имен, объявленное с применением модификатора управления доступом `private`, для реализации системы управления доступом на основании разрешений. Эта система рассматривается далее, в подразд. «Пример: управление доступом на основании разрешений» разд. «Практические примеры использования пространств имен».

Идентификатор пространства имен, объявленный в классе с использованием модификатора управления доступом `protected`, `internal` или `public`, доступен напрямую в любом месте данного класса и его подклассов, но недоступен в любом другом внешнем коде. Такое определение пространства имен является противоположностью определения на уровне пакета с применением модификатора управления доступом `public`, создающего идентификатор пространства имен, к которому можно обращаться напрямую из любого места программы. Это демонстрирует следующий код:

```
public class A {
  public namespace n = "http://www.example.com/n";

  // Отлично. Идентификатор пространства имен n доступен напрямую
  // в этом месте.
  n var someVariable:int;
}

public class B extends A {
  // Отлично. Идентификатор пространства имен n доступен напрямую
  // в этом месте.
  n var someOtherVariable:int;
}

public class C {
  // Ошибка. Идентификатор пространства имен n напрямую не доступен
  // в этом месте.
  // (Но идентификатор n мог бы быть доступен, если бы он был определен
  // на уровне пакета.)
  n var yetAnotherVariable:int;
}
```

Стоит отметить, что, хотя идентификатор пространства имен, объявленный в классе с использованием модификатора управления доступом `internal` или `public`, доступен напрямую только в этом классе и его подклассах, к объекту `Namespace`, на который ссылается идентификатор пространства имен, можно обратиться с применением синтаксиса обращения к статическим переменным.

Например, чтобы обратиться к объекту Namespace, на который ссылается идентификатор пространства имен n в предыдущем коде, мы могли бы использовать выражение: A.n. Доступ к объектам Namespace с помощью синтаксиса обращения к статическим переменным регламентируется обычными ограничениями, налагаемыми на статические переменные, объявленные с применением модификаторов управления доступом protected, internal и public. В предыдущем коде, поскольку идентификатор n объявлен с использованием модификатора управления доступом public, выражение A.n является допустимым в любом коде, который имеет доступ к классу A. Если бы идентификатор n был объявлен с использованием модификатора управления доступом internal, выражение A.n было бы допустимым только внутри пакета, содержащего определение данного пространства имен. Если бы идентификатор n был объявлен с использованием модификатора управления доступом protected, выражение A.n было бы допустимым только внутри класса A и его подклассов.

Однако ссылки на пространства имен, устанавливаемые через класс (например, A.n), не могут быть использованы в качестве атрибута в определении переменной или метода. Следующий синтаксис недопустим, поскольку атрибут определения переменной или метода должен быть константным значением на этапе компиляции:

```
A.n var p:int;  // Недопустимо. A.n не является константой
                // на этапе компиляции.
```

Итак, если мы не можем использовать выражение A.n для уточнения определений, для чего оно *может* использоваться вообще? Оставайтесь с нами, мы скоро узнаем ответ на этот вопрос в разд. «Присваивание и передача значений пространств имен».

Пока рассмотрим последнюю тему, связанную с доступностью пространств имен: определение пространств имен в методах и функциях.

Доступность определений пространств имен на уровне функции

Как и в случае с другими определениями на уровне функции, идентификатор пространства имен, объявленный на уровне функции, не может включать никакие модификаторы управления доступом (то есть он не может быть объявлен с использованием модификаторов управления доступом public, private и т. д.) и доступен только в области видимости данной функции:

```
public function doSomething ( ):void {
  // Это недопустимо
  private namespace n = "http://www.example.com/n";
}
```

Более того, определения локальных переменных и вложенных функций не могут использовать пространства имен в качестве атрибутов:

```
public function doSomething ( ):void {
  // Это тоже недопустимо
  namespace n = "http://www.example.com/n";
  n var someLocalVariable:int = 10;
}
```

Таким образом, идентификаторы пространств имен, определенные в функции, могут быть использованы только для формирования уточненных идентификаторов (в следующем коде предполагается, что пространство имен n было определено где-то в программе и применяется для уточнения переменной экземпляра someVariable).

```
public function doSomething ( ):void {
  // Это допустимо
  namespace n = "http://www.example.com/n";
  trace(n::someVariable);
}
```

Определения пространств имен на уровне функции используются только в редких случаях, когда функция не может обратиться к пространству имен, которое временно требуется для этой функции, напрямую, при этом идентификатор URI пространства имен известен. Например, функция, которая обрабатывает фрагмент XML-документа, содержащего уточненные имена элементов, может использовать код, похожий на следующий:

```
public function getPrice ( ):void {
  namespace htmlNS = "http://www.w3.org/1999/xhtml";
  output.text = htmlNS::table.htmlNS::tr[1].htmlNS::td[1].price;
}
```

Пространства имен XML будут рассматриваться в гл. 18.

Видимость уточненных идентификаторов

Наверняка вы заметили, что в этой книге определения уточненных идентификаторов не включают модификаторы управления доступом (public, internal, protected или private). Мы видели достаточно много таких определений:

```
fruit var orange:String = "Round citrus fruit";
```

Однако ни одного такого (обратите внимание на присутствие модификатора управления доступом private):

```
private fruit var orange:String = "Round citrus fruit";
```

По понятной причине *нельзя* использовать модификаторы управления доступом в определениях, которые включают уточняющее пространство имен. Например, следующий код:

```
private fruit var orange:String;
```

вызовет ошибку:

```
Access specifiers not allowed with namespace attributes
```

На русском языке она будет звучать так: Использование спецификаторов вместе с атрибутами пространства имен недопустимо.

Однако если нельзя использовать модификаторы управления доступом, с помощью чего можно управлять доступностью уточненного идентификатора? Ответ: с помощью доступности идентификатора уточняющего пространства имен.

Доступность уточняющего пространства имен в уточненном идентификаторе определяет доступность этого идентификатора. Если уточняющее пространство имен доступно в указанной области видимости, значит, доступен и уточненный идентификатор.

Например, в выражении `gameitems.fruit::orange` переменная `fruit::orange` доступна тогда, и только тогда, когда пространство имен `fruit` доступно в области видимости этого выражения. Доступность переменной `fruit::orange` целиком и полностью определяется доступностью пространства имен `fruit`.

В листинге 17.2 демонстрируется видимость уточненного идентификатора на примере обобщенного кода.

Листинг 17.2. Демонстрация видимости уточненного идентификатора

```
// Создаем пространство имен n в пакете one, видимое только в этом пакете
package one {
  internal namespace n = "http://www.example.com/n";
}

// Создаем переменную n::p в классе A, пакет one
package one {
  public class A {
    n var p:int = 1;
  }
}

// Поскольку пространство имен n объявлено с использованием модификатора
// управления доступом internal, переменная n::p доступна в любом месте
// внутри пакета one
package one {
  public class B {
    public function B ( ) {
      var a:A = new A( );
      trace(a.n::p); // OK
    }
  }
}

// Однако переменная n::p недоступна для кода за пределами пакета one
package two {
  import one.*;

  public class C {
    public function C ( ) {
      var a:A = new A( );
      trace(a.n::p);  // Недопустимо, поскольку пространство имен n
                      // объявлено с использованием модификатора управления
                      // доступом internal в пакете one, и поэтому
                      // недоступно в пакете two
    }
  }
```

Сравнение уточненных идентификаторов

Два пространства имен считаются одинаковыми тогда, и только тогда, когда совпадают их названия (URI). Например, чтобы определить, являются ли одинаковыми пространства имен в уточненных идентификаторах `fruit::orange` и `color::orange`, среда выполнения Flash не проверяет, соответствуют ли буквы слова «fruit» первого идентификатора буквам слова color второго идентификатора. Вместо этого среда Flash проверяет, совпадают ли значения переменной `uri` экземпляра класса `Namespace`, на который ссылается идентификатор `fruit`, и экземпляра класса `Namespace`, на который ссылается идентификатор `color`. Если значение переменной `fruit.uri` равняется значению переменной `color.uri`, то пространства имен считаются одинаковыми.

Таким образом, когда мы записываем следующее выражение:

```
trace(fruit::orange == color::orange);
```

среда Flash выполняет данное сравнение (обратите внимание на использование расширенных имен, которые рассматривались в разд. «Уточненные идентификаторы»):

```
{http://www.example.com/games/kidsgame/fruit}orange
   == {http://www.example.com/games/kidsgame/color}orange
```

Даже если два уточненных идентификатора внешне кажутся различными, на деле они могут оказаться одинаковыми и привести к неожиданным конфликтам имен. Например, в следующем коде попытка определить переменную `ns2::p` вызовет ошибку на этапе компиляции, поскольку переменная с расширенным именем `{http://www.example.com/general}p` уже существует:

```
namespace ns1 = "http://www.example.com/general"
namespace ns2 = "http://www.example.com/general"
ns1 var p:int = 1;
ns2 var p:int = 2;  // Ошибка! Повторное определение переменной!
```

Даже несмотря на то, что идентификаторы `ns1` и `ns2` являются различными, переменные `ns1::p` и `ns2::p` считаются одинаковыми, поскольку они имеют одинаковые расширенные имена (`{http://www.example.com/general}p`).

Стоит отметить, что названия пространств имен (идентификаторы URI) сравниваются как строки, с учетом регистра символов. Поэтому, несмотря на то, что для браузера два идентификатора URI, которые отличаются только регистром символов, будут считаться одинаковыми, в языке ActionScript они будут считаться различными. В ActionScript следующие два идентификатора URI считаются различными, поскольку слово example в первом случае начинается со строчной буквы, а во втором — с прописной:

```
namespace ns1 = "http://www.example.com"
namespace ns2 = "http://www.Example.com"
trace(ns1 == ns2);  // Отображает: false
```

Присваивание и передача значений пространств имен

Поскольку каждое пространство представлено экземпляром класса `Namespace`, пространства имен могут присваиваться переменным или элементам массива, передаваться в методы и возвращаться из методов и вообще могут быть использованы, как любой другой объект. Эта гибкость позволяет:

- передавать пространство имен из одной области видимости в другую;
- динамически выбирать одно пространство имен из нескольких на этапе выполнения программы.

При использовании пространств имен в ActionScript эти действия являются крайне важными. Узнаем почему.

Присваивание значения пространства имен переменной

Чтобы присвоить значение пространства имен переменной, мы используем точно такой же синтаксис присваивания, как и для любого другого значения. Например, следующий код присваивает значение пространства имен `fruit` переменной `currentItemType` (напомним, что *значение пространства имен* — это объект класса `Namespace`):

```
// Файл fruit.as
package {
  namespace fruit = "http://www.example.com/games/kidsgame/fruit";
}

// Файл Items.as
package {
  public class Items {
    // Присваиваем значение пространства имен fruit переменной
    // currentItemType
    private var currentItemType:Namespace = fruit;
  }
}
```

Переменная, которая ссылается на объект класса `Namespace`, может быть использована для формирования уточненного идентификатора. Например, рассмотрим следующее определение переменной:

```
fruit var orange:String = "Round citrus fruit";
```

Для того чтобы обратиться к этой переменной, мы можем использовать выражение `fruit::orange` либо `currentItemType::orange`. Присвоив переменной `currentItemType` некоторое другое пространство имен, можно динамически изменить смысл идентификатора `currentItemType::orange`, а также всех остальных методов и переменных, уточняемых переменной `currentItemType` по всей программе. Если организовать группы методов и переменных с помощью пространств имен, мы можем воспользоваться возможностью динамического

выбора пространства имен для переключения между различными режимами работы программы.

Например, предположим, что мы создаем приложение для мгновенного обмена сообщениями, которое может функционировать в двух режимах, представляемых соответствующими пространствами имен `offline` и `online`. В приложении определены две версии метода с именем `sendMessage( )`: одна версия предназначена для работы в режиме онлайн, а другая — для работы в автономном режиме.

```
online sendMessage (msg:String):void {
  // Отправить сообщение прямо сейчас...
}

offline sendMessage (msg:String):void {
  // Поставить сообщение в очередь и отправить его позднее...
}
```

Наше приложение управляет текущим режимом работы с помощью переменной `currentMode`. Всякий раз, когда устанавливается или теряется соединение с сервером, обновляется значение переменной `currentMode`.

```
private function connectListener (e:Event):void {
  currentMode = online;
}

private function closeListener (e:Event):void {
  currentMode = offline;
}
```

Во всех обращениях к методу `sendMessage( )` в качестве уточняющего пространства имен применяется переменная `currentMode`, как показано в следующем коде:

```
currentMode::sendMessage("yo dude");
```

Изменяя значение переменной `currentMode`, приложение динамически переключается между двумя версиями метода `sendMessage( )` в зависимости от состояния соединения.

Далее, в подразд. «Пример: реализация режимов работы программы» разд. «Практические примеры использования пространств имен», мы вернемся к концепции использования пространств имен в качестве режимов работы программы на примере японско-английского словаря, в котором происходит переключение между различными режимами поиска.

Просто запомните, что, хотя переменная может применяться для указания пространства имен уточненного идентификатора, переменные *не могут* быть использованы для указания пространства имен в определении переменной или метода. Третья строка следующего кода:

```
namespace fruit;
var currentItemType:Namespace = fruit;
currentItemType var orange:String = "Round citrus fruit";
```

вызовет такую ошибку:

```
Namespace was not found or is not a compile-time constant.
```

По-русски это будет звучать так: Пространство имен не найдено, или оно не является константой на этапе компиляции.

Аналогичным образом переменные не могут быть использованы для указания пространства имен в директиве `use namespace`. Мы познакомимся с этой директивой далее, в разд. «Открытые пространства имен и директива use namespace».

Пространства имен в качестве аргументов и возвращаемых значений методов

Вдобавок к тому, что значения пространств имен могут присваиваться переменным и элементам массива, они могут передаваться в методы и возвращаться из них. Например, следующий код определяет метод `doSomething( )`, который принимает значение пространства имен в качестве аргумента:

```
public function doSomething (n:Namespace):void {
  trace(n);
}
```

Этот код передает пространство имен `fruit` в метод `doSomething( )`:

```
doSomething(fruit);
```

Пространство имен может передаваться в метод для того, чтобы перенести одну часть контекста программы в другую. Например, приложение, реализующее корзину в интернет-магазине, может передавать пространство имен `currentLocale` в класс `Checkout`, который затем динамически выберет подходящую валюту и зависящее от времени приветствие, взяв за основу текущее местоположение пользователя.

Этот код определяет метод `getNamespace( )`, который возвращает пространство имен `fruit`:

```
public function getNamespace ( ):Namespace {
  return fruit;
}
```

Пространство имен может возвращаться из метода для того, чтобы предоставить вызывающему коду привилегированный доступ к закрытым переменным и методам. Полный пример, демонстрирующий возврат пространства имен из метода в качестве части системы управления доступом на основе разрешений, можно найти далее, в подразд. «Пример: управление доступом на основании разрешений» разд. «Практические примеры использования пространств имен».

Пример использования значения пространства имен

Теперь, когда мы познакомились с тем, как работают значения пространств имен в теории, вернемся к нашему предыдущему примеру детской игры и посмотрим, как значения пространств имен могут быть использованы в реальном приложении. Напомним, что указанная игра представляет собой упражнение на чтение, в котором игрок пытается распознать выбранный случайным образом цвет или фрукт.

Первая версия кода игры (см. листинг 17.1) демонстрировала единственный, чрезвычайно упрощенный фрагмент игры. В этой измененной версии мы сделаем игру полностью функциональной, подробно рассмотрев возможность применения пространств имен для управления несколькими наборами данных.

Бегло просмотрите код в листинге 17.3, чтобы получить общее представление о программе. В данном примере пространства имен используются только в классах `Items` и `KidsGame`, поэтому вы должны сосредоточить свое внимание на этих классах. Информацию о методиках, применяемых для создания пользовательского интерфейса в этом примере, можно найти в части II. Подробный анализ кода представлен сразу после листинга.

Листинг 17.3. Детская игра: пример использования значения пространства имен

```
// Файл fruit.as
package {
  public namespace fruit = "http://www.example.com/games/kidsgame/fruit";
}

// Файл color.as
package {
  public namespace color = "http://www.example.com/games/kidsgame/color";
}

// Файл Items.as
package {
  // Простой класс для хранения данных, содержащий объекты Item для игры.
  public class Items {
    // Фрукты
    fruit var orange:Item = new Item("Orange", "fruit-orange.jpg", 1);
    fruit var apple:Item = new Item("Apple", "fruit-apple.jpg", 2);

    // Цвета
    color var orange:Item = new Item("Orange", "color-orange.jpg", 3);
    color var purple:Item = new Item("Purple", "color-purple.jpg", 4);

    // Массивы, хранящие полные наборы элементов (то есть все фрукты
    // или все цвета)
    fruit var itemSet:Array = [fruit::orange, fruit::apple];
    color var itemSet:Array = [color::orange, color::purple];

    // Массив пространств имен, представляющих
    // типы наборов элементов в игре
    private var itemTypes:Array = [color, fruit];

    // Возвращает все элементы-фрукты, используемые в игре
    fruit function getItems ( ):Array {
      return fruit::itemSet.slice(0);
    }

    // Возвращает все элементы-цвета, используемые в игре
    color function getItems ( ):Array {
```

```
      return color::itemSet.slice(0);
    }

    // Возвращает список доступных типов элементов, используемых в игре
    public function getItemTypes ( ):Array {
      return itemTypes.slice(0);
    }
  }
}

// Файл KidsGame.as
package {
  import flash.display.*;
  import flash.events.*;
  import flash.utils.*;

  // Основной класс приложения для детской игры «Учусь читать», которое
  // демонстрирует основы использования пространств имен в языке ActionScript.
  // Игроку показывается картинка с цветом или фруктом и предлагается выбрать
  // название этого цвета или фрукта из списка вариантов.
  public class KidsGame extends Sprite {
    private var gameItems:Items;  // Список всех элементов, используемых
                                  // в игре
    private var thisQuestionItem:Item; // Элемент для каждого вопроса
    private var questionScreen:QuestionScreen; // Пользовательский интерфейс

    // Конструктор
    public function KidsGame( ) {
      // Получаем элементы, используемые в игре (фрукты и цвета,
      // названия которых должен указать игрок)
      gameItems = new Items( );
      // Отображаем первый вопрос
      newQuestion( );
    }

    // Создаем и отображаем новый случайный вопрос
    public function newQuestion ( ):void {
      // Получаем полный список типов элементов (массив пространств имен)
      var itemTypes:Array = gameItems.getItemTypes( );
      // Случайным образом выбираем тип элемента (одно из пространств имен,
      // на которые ссылается переменная itemTypes)
      var randomItemType:Namespace = itemTypes[Math.floor(
                                     Math.random( )*itemTypes.length)];

      // Получаем элементы набора, выбранного случайным образом
      var items:Array = gameItems.randomItemType::getItems( );

      // Случайным образом выбираем элемент для данного вопроса
      // из набора элементов
      thisQuestionItem = items[Math.floor(Math.random( )*items.length)];
```

```
      // Удаляем предыдущий вопрос, если он существует
      if (questionScreen != null) {
        removeChild(questionScreen);
      }

      // Отображаем новый вопрос
      questionScreen = new QuestionScreen(this, items, thisQuestionItem);
      addChild(questionScreen);
    }

    // Обрабатываем ответ игрока
    public function submitGuess (guess:int):void {
      trace("Guess: " + guess + ", Correct: " + thisQuestionItem.id);
      if (guess == thisQuestionItem.id) {
        questionScreen.displayResult("Correct!");
        // Отключить кнопки ответа до тех пор, пока игрок не дождется
        // следующего вопроса.
        questionScreen.disable( );
        // Подождать 3 секунды перед отображением следующего вопроса.
        var timer:Timer = new Timer(3000, 1);
        timer.addEventListener(TimerEvent.TIMER, doneResultDelay);
        timer.start( );
      } else {
        questionScreen.displayResult("Incorrect. Please try again.");
      }
    }

    // Создает новый вопрос после того, как был получен
    // ответ на предыдущий.
    private function doneResultDelay (e:TimerEvent):void {
      newQuestion( );
    }
  }
}

// Файл Item.as
package {
  // Простой контейнер данных, который хранит информацию об элементе.
  public class Item {
    // Название элемента (например, "apple")
    public var name:String;
    // Адрес URL, с которого загружается изображение,
    // представляющее элемент
    public var src:String;
    // Уникальный идентификатор элемента, который используется
    // для обработки ответов игрока
    public var id:int;

    // Конструктор
    public function Item (name:String, src:String, id:int) {
      this.name = name;
```

```
      this.src = src;
      this.id = id;
    }
  }
}

// Файл QuestionScreen.as
package {
  import flash.events.*;
  import flash.display.*;
  import flash.text.*;
  import flash.net.*;

  // Создает пользовательский интерфейс для вопроса
  public class QuestionScreen extends Sprite {
    private var status:TextField;
    private var game:KidsGame;
    private var items:Array;
    private var thisQuestionItem:Item;

    // Конструктор
    public function QuestionScreen (game:KidsGame,
                                    items:Array,
                                    thisQuestionItem:Item) {
      // Сохраняем ссылку на основной объект игры
      this.game = game;

      // Собираем данные о вопросе
      this.items = items;
      this.thisQuestionItem = thisQuestionItem;

      // Помещаем вопрос на экран
      makeQuestion( );
    }

    // Создаем и отображаем интерфейс для вопроса
    public function makeQuestion ( ):void {
      // Отображаем картинку для элемента
      var imgLoader:Loader = new Loader( );
      addChild(imgLoader);
      imgLoader.load(new URLRequest(thisQuestionItem.src));

      // Добавляем набор слов, которые может выбирать игрок,
      // щелкая кнопкой мыши. Для упрощения будем отображать
      // название каждого элемента в наборе.
      var wordButton:WordButton;
      for (var i:int = 0; i < items.length; i++) {
        wordButton = new WordButton( );
        wordButton.setButtonText(items[i].name);
        wordButton.setID(items[i].id);
        wordButton.y = 110 + i*(wordButton.height + 5);
```

```
        wordButton.addEventListener(MouseEvent.CLICK, clickListener);
        addChild(wordButton);
      }

      // Создаем текстовое поле, в котором будет отображаться статус вопроса
      status = new TextField( );
      status.autoSize = TextFieldAutoSize.LEFT;
      status.y = wordButton.y + wordButton.height + 10;
      status.selectable = false;
      addChild(status);
    }

    // Отображает сообщение в поле статуса
    public function displayResult (msg:String):void {
      status.text = msg;
    }

    // Выводит выбранное пользователем слово для данного вопроса
    public function disable ( ):void {
      // Отключаем события мыши для всех потомков данного объекта Sprite.
      mouseChildren = false;
    }

    // Реагируем на событие, вызванное щелчком кнопкой мыши на кнопке-слове
    private function clickListener (e:MouseEvent):void {
      // Выбранный игроком вариант имеет идентификатор элемента, который был
      // присвоен объекту WordButton в методе makeQuestion( ).
      game.submitGuess(e.target.getID( ));
    }
  }
}

// Файл WordButton.as
package {
  import flash.text.*;
  import flash.display.*;

  // Представляет на экране слово, на котором можно щелкнуть кнопкой мыши
  // (то есть доступный вариант ответа на вопрос). ID обозначает
  // идентификатор элемента, выбранного игроком (переменная Item.id).
  public class WordButton extends Sprite {
    private var id:int; // Идентификатор элемента, представляемого
                        // данной кнопкой
    private var t:TextField;

    // Конструктор
    public function WordButton ( ) {
      t = new TextField( );
      t.autoSize   = TextFieldAutoSize.LEFT;
      t.border     = true;
      t.background = true;
```

```
      t.selectable = false;
      addChild(t);

      buttonMode    = true;
      mouseChildren = false;
    }

    // Присваивает текст, отображаемый на кнопке
    public function setButtonText (text:String):void {
      t.text = text;
    }

    // Присваивает идентификатор элемента, представляемого данной кнопкой
    public function setID (newID:int):void {
      id = newID;
    }

    // Возвращает идентификатор элемента, представляемого данной кнопкой
    public function getID ( ):int {
      return id;
    }
  }
}
```

Просмотрели код? Отлично, рассмотрим его более детально. Возможно, вы заметили, что определения пространств имен в игре не изменялись вообще с того момента, как они были представлены в листинге 17.1. Тем не менее существенно изменился класс `Items`, а также появилось несколько новых классов:

- `KidsGame` — основной класс приложения;
- `Item` — предоставляет информацию о конкретном элементе игры;
- `QuestionScreen` — формирует пользовательский интерфейс для каждого вопроса;
- `WordButton` — представляет кнопку-слово на экране.

Поскольку сейчас наше внимание сосредоточено на пространствах имен, мы рассмотрим только классы `Items` и `KidsGame`. В качестве упражнения рассмотрите оставшиеся классы самостоятельно.

Для начала посмотрим, как изменился класс `Items` со времени его предыдущего представления в листинге 17.1. Во-первых, мы добавили две новые переменные `fruit::apple` и `color::purple`, представляющие элементы. Благодаря новым переменным каждая категория элементов для фруктов и цветов теперь состоит из двух элементов: апельсина и яблока для фруктов и оранжевого и фиолетового для цветов. Мы также заменили простые описания элементов из листинга 17.1 (например, Round citrus fruit) экземплярами класса `Item`. Экземпляры класса `Item` хранят название элемента, URL-адрес изображения этого элемента и его идентификатор. Следующий код демонстрирует измененные переменные, представляющие элементы. Как и в листинге 17.1, каждая переменная уточняется пространством имен, соответствующим множеству, которому принадлежит данный элемент.

```
fruit var orange:Item = new Item("Orange", "fruit-orange.jpg", 1);
fruit var apple:Item = new Item("Apple", "fruit-apple.jpg", 2);

color var orange:Item = new Item("Orange", "color-orange.jpg", 3);
color var purple:Item = new Item("Purple", "color-purple.jpg", 4);
```

Кроме того, в класс `Items` добавлены два массива, предназначенные для управления элементами в виде групп. Каждый массив содержит полный список элементов своей группы (либо фруктов, либо цветов). Массивы присваиваются переменным с одним и тем же именем (`itemSet`), но уточняемым различными пространствами имен (`fruit` и `color`).

```
fruit var itemSet:Array = [fruit::orange, fruit::apple];
color var itemSet:Array = [color::orange, color::purple];
```

Чтобы предоставить доступ другим классам к различным наборам элементов в игре, в `Items` определено два метода с одним и тем же локальным именем `getItems ( )`, которое уточняется различными пространствами имен `fruit` и `color`. Каждый метод `getItems ( )` возвращает копию набора элементов, соответствующего пространству имен данного метода. Таким образом, обратиться к подходящему набору элементов можно динамически, в зависимости от текущего типа вопроса (либо цвет, либо фрукт).

```
fruit function getItems ( ):Array {
  // Возвращает фрукты.
  return fruit::itemSet.slice(0);
}

color function getItems ( ):Array {
  // Возвращает цвета.
  return color::itemSet.slice(0);
}
```

Наконец, в классе `Items` определены переменная `itemTypes` и соответствующий метод-аксессор `getItemTypes( )`. Переменная `itemTypes` хранит список всех различных множеств элементов в игре. В нашей игре определено только два множества: фрукты и цвета, но в дальнейшем можно будет легко добавить новые множества. Каждое множество элементов соответствует пространству имен, поэтому переменная `itemTypes` представляет собой массив пространств имен. Метод `getItemTypes( )` возвращает копию этого массива, предоставляя внешнему коду возможность централизованно получать официальный список типов элементов в игре.

```
// Переменная itemTypes
private var itemTypes:Array = [color, fruit];

// Метод getItemTypes( )
public function getItemTypes ( ):Array {
  return itemTypes.slice(0);
}
```

Это все, что касается изменений в классе `Items`. Теперь рассмотрим новый основной класс приложения `KidsGame`. В отличие от `Items`, класс `KidsGame` никогда не

использует идентификаторы пространств имен `fruit` и `color` напрямую. Вместо этого он обращается к указанным пространствам имен через метод экземпляра `getItemTypes( )` класса `Items`.

Переменная `gameItems` класса `KidsGame` позволяет ему обращаться к игровым данным посредством объекта класса `Items`. При этом метод `newQuestion( )` класса `KidsGame` генерирует новый вопрос на основании данных, хранящихся в переменной `gameItems`. Метод `newQuestion( )` включает основную часть кода, связанного с использованием пространств имен. Именно эта часть интересует нас больше всего, поэтому рассмотрим данный код детально.

Напомним, что каждый вопрос отображает элемент одного из предопределенных наборов элементов, хранящихся в классе `Items` (`fruit::itemSet` или `color::itemSet`). Соответственно первая задача, которая стоит перед методом `newQuestion( )`, — случайным образом выбрать набор элементов для генерируемого вопроса. Сначала мы получаем весь массив возможных наборов элементов (то есть пространств имен) из класса `Items`, используя метод `gameItems.getItemTypes( )`:

```
var itemTypes:Array = gameItems.getItemTypes( );
```

Затем мы случайным образом выбираем пространство имен из результирующего массива. Для удобства мы присваиваем выбранное пространство имен локальной переменной `randomItemType`.

```
var randomItemType:Namespace = itemTypes[Math.floor(
                        Math.random( )*itemTypes.length)];
```

Обратите внимание, что типом данных переменной `randomItemType` является тип `Namespace`, поскольку эта переменная ссылается на значение пространства имен. Как только будет выбран набор элементов (пространство имен) для вопроса, мы должны получить список существующих элементов из этого набора. Чтобы получить соответствующий массив элементов (либо фруктов, либо цветов), мы вызываем метод класса `Items`, который соответствует нашему выбранному пространству имен, — либо метод `fruit::getItems( )`, либо метод `color::getItems( )`. Однако вместо того, чтобы обращаться к желаемому методу напрямую, мы динамически генерируем уточненный идентификатор метода, используя переменную `randomItemType` для определения пространства имен, как показано в следующем коде:

```
gameItems.randomItemType::getItems( )
```

Массив, возвращаемый методом, присваивается локальной переменной `items`:

```
var items:Array = gameItems.randomItemType::getItems( );
```

В предыдущем вызове метода обратите внимание, что поведение программы определяется контекстом программы. Эту особенность можно рассматривать как разновидность полиморфизма, которая основывается не на наследовании классов, а на произвольных группах методов и переменных, определяемых пространствами имен.

Теперь, когда у нас на руках есть массив элементов, мы можем продолжить текущую работу, которая заключается в отображении вопроса на экране. Сначала

мы случайным образом выбираем элемент для отображения из массива элементов:

```
thisQuestionItem = items[Math.floor(Math.random( )*items.length)];
```

Затем мы размещаем изображение и варианты ответов для выбранного элемента на экране, используя класс `QuestionScreen`:

```
// Удаляем предыдущий вопрос, если он существует
if (questionScreen != null) {
  removeChild(questionScreen);
}

// Отображаем новый вопрос
questionScreen = new QuestionScreen(this, items, thisQuestionItem);
addChild(questionScreen);
```

Приведем код метода `newQuestion( )` еще раз. Обратите особое внимание на использование значений пространств имен в этом коде, поскольку мы не будем больше возвращаться к нему.

```
public function newQuestion ( ):void {
  // Получаем полный список типов элементов (массив пространств имен)
  var itemTypes:Array = gameItems.getItemTypes( );
  // Случайным образом выбираем тип элемента (одно из пространств имен,
  // на которые ссылается переменная itemTypes)
  var randomItemType:Namespace = itemTypes[Math.floor(
                                 Math.random( )*itemTypes.length)];

  // Получаем элементы набора, выбранного случайным образом
  var items:Array = gameItems.randomItemType::getItems( );

  // Случайным образом выбираем элемент для данного вопроса
  // из набора элементов
  thisQuestionItem = items[Math.floor(Math.random( )*items.length)];

  // Удаляем предыдущий вопрос, если он существует
  if (questionScreen != null) {
    removeChild(questionScreen);
  }

  // Отображаем новый вопрос
  questionScreen = new QuestionScreen(this, items, thisQuestionItem);
  addChild(questionScreen);
}
```

Оставшаяся часть кода из листинга 17.3 относится к игровой логике и созданию пользовательского интерфейса, что в настоящее время не является нашей основной задачей. Как уже отмечалось ранее, вы должны самостоятельно изучить оставшийся код. Информацию о методиках создания пользовательского интерфейса можно найти в части II этой книги.

Что ж, это был хороший практический пример. Впереди нас ждет еще несколько примеров, однако сначала мы должны рассмотреть две фундаментальные концеп-

ции, относящиеся к пространствам имен: открытые пространства имен и пространства имен для модификаторов управления доступом.

Открытые пространства имен и директива use namespace

Помните простой класс `Items` из листинга 17.1?

```
package {
  public class Items {
    fruit var orange:String = "Round citrus fruit";
    color var orange:String = "Color obtained by mixing red and yellow";

    public function Items ( ) {
      trace(fruit::orange);
      trace(color::orange);
    }
  }
}
```

Как уже говорилось, один из способов обращения к переменным `orange` в предыдущем коде заключается в применении уточненных идентификаторов, как показано ниже:

```
trace(fruit::orange); // Выводит: Round citrus fruit
trace(color::orange); // Выводит: Color obtained by
                      //          mixing red and yellow
```

Однако язык ActionScript предлагает еще один удобный инструмент для обращения к переменным, уточняемым пространствами имен: директиву `use namespace`. Директива `use namespace` добавляет указанное пространство имен в набор так называемых *открытых пространств имен* для определенной области видимости программы. *Открытые пространства имен* — это набор пространств имен, к которому обращается компилятор при попытке разрешить неуточненные ссылки. Например, если пространство имен `n` находится в наборе открытых пространств имен и компилятор встретит неуточненную ссылку на переменную `p`, то он автоматически проверит существование переменной `n::p`.

Рассмотрим общий вид директивы `use namespace`:

`use namespace` *идентификаторПространстваИмен*

Здесь *идентификаторПространстваИмен* — это идентификатор пространства имен, которое должно быть добавлено в набор открытых пространств имен. Стоит отметить, что данный идентификатор должен быть константой на этапе компиляции, поэтому не может быть переменной, которая ссылается на значение пространства имен.

Посмотрим на примере предыдущего конструктора класса `Items`, как работает директива `use namespace`, обратившись напрямую к локальной переменной `orange` после того, как пространство имен `fruit` будет добавлено в набор

открытых пространств имен (эта операция также называется открытием пространства имен `fruit`).

```
public function Items ( ) {
  use namespace fruit;
  trace(orange);
}
```

Мы добавили пространство имен `fruit` в набор открытых пространств имен, поэтому, когда компилятор встретит следующий код:

```
trace(orange);
```

он автоматически проверит, существует ли уточненный идентификатор `fruit::orange`. В нашем примере данный идентификатор существует, поэтому он будет использован вместо локального имени `orange`. Другими словами, в конструкторе класса `Items` этот код:

```
trace(fruit::orange); // Выводит: Round citrus fruit
```

выполняет то же самое, что и следующий:

```
use namespace fruit;
trace(orange);        // Выводит: Round citrus fruit
```

Открытые пространства имен и область видимости

Каждая область видимости в программе на языке ActionScript имеет отдельный список открытых пространств имен. Пространство имен, открытое в определенной области видимости, будет открыто для нее, включая вложенные области, но при этом оно не будет открыто для остальных областей видимости. Открытое пространство имен будет доступно даже до инструкции `use namespace` (однако лучше всего помещать директиву `use namespace` в самом верху содержащего ее блока кода).

Напомним, что «область видимости» обозначает «область программы». В ActionScript для каждого пакета, класса и метода определена уникальная область видимости. Условные операторы и операторы циклов не имеют собственных областей видимости.

В листинге 17.4 демонстрируется общий код, чтобы показать две различные области видимости со своими отдельными списками открытых пространств имен. Пояснения к коду приводятся в виде комментариев.

Листинг 17.4. Демонстрация открытых пространств имен

```
public class ScopeDemo {
  // Создаем пространство имен.
  private namespace n1 = "http://www.example.com/n1";

  // Создаем две переменные, уточняемые пространством имен n1.
  n1 var a:String = "a";
  n1 var b:String = "b";

  // Конструктор
  public function ScopeDemo ( ) {
```

```
    // Вызываем метод, который обращается к переменной n1::a.
    showA( );
  }

  public function showA ( ):void {
    // Эта неуточненная ссылка на переменную a полностью соответствует
    // уточненному идентификатору n1::a, поскольку следующая строка кода
    // открывает пространство имен n1.
    trace(a); // OK!

    // Открываем пространство имен n1.
    use namespace n1;

    // Неуточненная ссылка на переменную a
    // снова соответствует уточненному
    // идентификатору n1::a.
    trace(a); // OK!

    // Создаем вложенную функцию.
    function f ( ):void {
      // Пространство имен n1 остается открытым во вложенных областях
      // видимости...
      trace(a); // OK! Соответствует n1::a.
    }

    // Вызываем вложенную функцию.
    f( );
  }

  public function showB ( ):void {
    // В следующем коде происходит неправильное обращение к переменной
    // n1::b. Пространство имен n1 открыто только в области видимости метода
    // showA( ), но не в области видимости метода showB( ), поэтому попытка
    // обращения окажется неудачной. Более того, в области видимости метода
    // showB( ) не существует ни одной переменной с простым
    // идентификатором b, поэтому компилятор сгенерирует следующую ошибку:
    //   Attempted access to inaccessible property b through a reference
    //   with static type ScopeDemo.
    //   (Предпринята попытка обращения к недоступному свойству b через
    //   ссылку на статический тип ScopeDemo.)
    trace(b); // ОШИБКА!
  }
}
```

Поскольку открытое пространство имен остается открытым во вложенных областях видимости, мы можем открывать пространство имен на уровне класса или пакета с тем, чтобы использовать его в любом месте блока инструкции `class` или `package`. Однако стоит отметить, что после того, как пространство имен было открыто, закрыть его будет невозможно. Не существует директивы `unuse namespace`, равно как не существует способа удалить пространство имен из списка открытых пространств имен определенной области видимости.

Открытие нескольких пространств имен

Вполне допустимо открывать несколько пространств имен в одной и той же области видимости. Например, рассмотрим четыре переменные, относящиеся к двум пространствам имен (переменные взяты из класса `Items`, представленного в листинге 17.3):

```
fruit var orange:Item = new Item("Orange", "fruit-orange.jpg", 1);
fruit var apple:Item = new Item("Apple", "fruit-apple.jpg", 2);
color var orange:Item = new Item("Orange", "color-orange.jpg", 3);
color var purple:Item = new Item("Purple", "color-purple.jpg", 4);
```

Предположим, что мы добавили метод `showItems ( )` в класс `Items` для отображения всех элементов игры. В этом методе мы можем открыть оба пространства имен `fruit` и `color`, а затем обращаться к переменным `fruit::apple` и `color::purple`, не указывая уточняющее пространство имен:

```
public function showItems ( ):void {
  use namespace fruit;
  use namespace color;
  // Вот это да! Никаких пространств имен!
  trace(apple.name);  // Выводит: Apple
  trace(purple.name); // Выводит: Purple
}
```

Рассмотрим, как это работает. Как уже известно, термин *«открытые пространства имен»* означает «набор пространств имен, к которому обращается компилятор при попытке разрешить неуточненные ссылки». Если в указанной области видимости открыто несколько пространств имен, компилятор проверяет каждое пространство абсолютно для всех неуточненных ссылок в данной области видимости. Например, в методе `showItems ( )` открыты оба пространства имен `fruit` и `color`. Следовательно, когда компилятор встречает неуточненный идентификатор `apple`, он проверяет, существуют ли идентификаторы `fruit::apple` и `color::apple`. В случае с идентификатором `apple` неуточненная ссылка соответствует идентификатору `fruit::apple`, но не соответствует идентификатору `color::apple`. Поскольку идентификатор `apple` соответствует только одному уточненному идентификатору (а именно, `fruit::apple`), этот уточненный идентификатор и используется вместо неуточненной ссылки `apple`.

Что же произойдет в том случае, если используется неуточненная ссылка, как, например, `orange`, которая соответствует двум уточненным идентификаторам:

```
public function showItems ( ):void {
  use namespace fruit;
  use namespace color;
  // Соответствует fruit::orange и color::orange -
  // что произойдет в этом случае?
  trace(orange);
}
```

Если неуточненная ссылка соответствует имени в более чем одном пространстве имен, возникает ошибка на этапе выполнения. Предыдущий код вызовет следующую ошибку:

```
Ambiguous reference to orange.
```

По-русски ошибка будет выглядеть следующим образом: Неоднозначная ссылка на orange.

Из-за ошибки в некоторых компиляторах компании Adobe предыдущая ошибка может остаться незамеченной.

Если открыты оба пространства имен `fruit` и `color`, мы должны использовать уточненные идентификаторы `fruit::orange` или `color::orange` для обращения к нашим переменным `orange`, исключая неоднозначность, как показано в следующем коде:

```
public function showItems ( ):void {
  use namespace fruit;
  use namespace color;

  trace(apple);  // Выводит: Apple
  trace(purple); // Выводит: Purple

  // Открыты оба пространства имен fruit и color, поэтому ссылки
  // на переменную orange должны быть полностью уточнены.
  trace(fruit::orange);
  trace(color::orange);
}
```

Пространства имен для модификаторов управления доступом

Точно так же, как мы используем пространства имен для управления видимостью переменных и методов в наших собственных программах, язык ActionScript использует пространства имен для управления видимостью каждой переменной и каждого метода в любой программе! Помните четыре модификатора управления доступом в ActionScript — `public`, `internal`, `protected`, `private`? Сам язык ActionScript реализует приведенные правила видимости с помощью пространств имен. Например, с точки зрения ActionScript определение переменной:

```
class A {
  private var p:int;
}
```

означает «создать новую переменную `p`, уточняемую пространством имен `private` класса `A`».

В каждой области видимости ActionScript неявно открывает подходящее пространство имен для различных модификаторов управления доступом. Например, в каждой области видимости всегда добавляется глобальное пространство имен `public` в набор открытых пространств имен. На верхнем уровне пакета также добавляются пространства имен `internal` и `public` данного пакета. В коде класса, который находится внутри пакета, также добавляются пространства имен `private` и `protected` данного класса. Таким образом, набор открытых пространств имен включает не

только пространства имен, открытые пользователем, но и пространства имен для управления доступом, которые неявно открываются в каждой области видимости.

Открыть пространства имен для управления доступом явно с помощью директивы use namespace невозможно. Среда выполнения открывает пространства имен для управления доступом автоматически, в соответствии с текущей областью видимости.

Пространства имен для модификаторов управления доступом определяют доступность идентификаторов и предотвращают конфликты именования. Например, в следующем коде суперкласс `Parent` и подкласс `Child` определяют переменную с одним и тем же именем, используя модификатор управления доступом `private`: `description`. Переменная `description` класса `Parent` недоступна для кода в классе `Child`, поскольку она уточнена пространством имен `private` класса `Parent`, которое не открывается в области видимости класса `Child`. Благодаря этому названия переменных не конфликтуют между собой.

```
package p {
  public class Parent {
    private var description:String = "A Parent object";
    public function Parent ( ) {
      trace(description);
    }
  }
}

package p {
  public class Child extends Parent {
    private var description:String = "A Child object";
    public function Child ( ) {
      trace(description); // Конфликта не происходит
    }
  }
}
```

Однако если переменную `description` класса `Parent` объявить с использованием модификатора управления доступом `protected`, возникнет конфликт. Рассмотрим почему. Во-первых, изменим модификатор управления доступом для переменной `description` на `protected`:

```
public class Parent {
  protected var description:String = "A Parent object";
}
```

Теперь представим себя на месте среды Flash, пытающейся выполнить код в конструкторе класса `Child`. Мы входим в конструктор и встречаем ссылку на идентификатор `description`. Чтобы разрешить этот идентификатор, мы должны проверить его существование в открытых пространствах имен. И какие же пространства имен открыты в конструкторе класса `Child`? Как мы уже знаем, в коде класса, который находится внутри пакета, среда Flash открывает пространства имен `private` и `protected` данного класса, пространства имен `internal` и `public` пакета и глобальное пространство имен `public`. Итак, открытыми пространствами имен являются:

- пространство имен `private` класса `Child`;
- пространство имен `protected` класса `Child` (которое уточняет все члены, унаследованные от непосредственного суперкласса);
- пространство имен `internal` пакета `p`;
- пространство имен `public` пакета `p`;
- глобальное пространство имен `public`;
- все пользовательские пространства имен, открытые явным образом.

Когда среда выполнения проверяет существование переменной `description` в открытых пространствах имен, она находит два соответствия: `private::description` и `protected::description` класса `Child`. Как мы уже знаем из предыдущего раздела, когда неуточненная ссылка соответствует имени в более чем одном пространстве имен, возникает ошибка неоднозначного обращения. Более того, если несколько имен уточняются различными неявно открытыми пространствами имен, возникает ошибка, связанная с конфликтом определений. В случае с переменной `description` возникнет следующая ошибка:

```
A conflict exists with inherited definition Parent.description in namespace
protected.
```

На русском языке она будет выглядеть так: **Существует конфликт с унаследованным определением Parent.description в пространстве имен protected.**

Если в вашем коде существуют конфликтующие имена методов и переменных, компилятор опишет суть конфликта, указав пространство имен, в котором этот конфликт произошел. Например, следующий код:

```
package {
  import flash.display.*;
  public class SomeClass extends Sprite {
    private var prop:int;
    private var prop:int; // Недопустимое повторное определение свойства
  }
}
```

вызовет следующую ошибку:

```
A conflict exists with definition prop in namespace private.
```

По-русски это будет звучать так: **Существует конфликт с определением prop в пространстве имен private.**

На самом деле из-за ошибки компилятора в приложениях Flex Builder 2 и Flash CS3 предыдущее сообщение будет содержать неправильную фразу `namespace internal`, хотя должно быть `namespace private`.

Подобным образом, данный код:

```
package {
  import flash.display.*;
  public class SomeClass extends Sprite {
    private var x;
  }
}
```

вызовет следующую ошибку (поскольку — об этом вы можете почитать в справочнике по языку ActionScript компании Adobe — в классе `DisplayObject` уже определена переменная `x` с использованием модификатора управления доступом `public`):

```
A conflict exists with inherited definition flash.display:DisplayObject.x in
namespace public.
```

На русском языке это будет выглядеть следующим образом: Существует конфликт с унаследованным определением flash.display:DisplayObject.x в пространстве имен public.

Директива import открывает пространства имен public. Стоит отметить, что с технической точки зрения импортирование пакета, как показано в следующем коде:

```
import somePackage.*;
```

открывает пространство имен `public` импортированного пакета. Тем не менее оно не открывает пространство имен `internal` импортированного пакета. Даже если пакет импортируется, его идентификаторы, объявленные с использованием модификатора управления доступом `internal`, остаются недоступными для внешнего кода.

Практические примеры использования пространств имен

В самом начале этой главы упоминалось четыре практических сценария использования пространств имен:

- предотвращение конфликтов именования;
- управление видимостью членов на уровне прикладной среды;
- управление доступом на основании разрешений;
- реализация различных режимов работы программы.

В предыдущем разделе рассказывалось, как пространства имен предотвращают конфликты именования. В этом разделе мы рассмотрим каждый из трех оставшихся сценариев на примерах из реальной жизни.

Пример: управление видимостью на уровне прикладной среды

Наш первый пример прикладного использования пространств имен взят из прикладной среды Flex компании Adobe — это библиотека компонентов пользовательского интерфейса и утилит для разработки интернет-приложений с широкими функциональными возможностями.

Прикладная среда Flex включает большое количество кода — сотни классов, размещаемых в дюжинах пакетов. Некоторые методы и переменные этих классов должны быть доступны в различных пакетах, но при этом они должны считаться внутренними по отношению ко всей прикладной среде. Возникает дилемма: если методы и переменные объявить с использованием модификатора управления до-

ступом public, код, находящийся за пределами прикладной среды, будет иметь к ним нежелательный доступ, но если их объявить с использованием модификатора управления доступом internal, эти методы и переменные нельзя будет использовать в других пакетах.

Для разрешения этой ситуации в прикладной среде Flex определяется пространство имен mx_internal, используемое для уточнения методов и переменных, которые не должны быть видны за пределами прикладной среды, но при этом должны быть доступны в различных пакетах внутри ее.

Вот объявление пространства имен mx_internal:

```
package mx.core {
  public namespace mx_internal =
      "http://www.adobe.com/2006/flex/mx/internal";
}
```

Рассмотрим конкретный пример использования пространства имен mx_internal из прикладной среды Flex.

Для работы с табличными данными наподобие тех, которые используются в программах электронных таблиц, прикладная среда Flex предоставляет компонент DataGrid. Класс DataGrid находится в пакете mx.controls. Вспомогательные классы для компонента DataGrid размещаются в отдельном пакете: mx.controls.gridclasses. Чтобы взаимодействие между классом DataGrid и его вспомогательными классами осуществлялось максимально эффективно, DataGrid обращается к некоторым внутренним переменным его вспомогательных классов напрямую, а не с помощью доступных всем методов-получателей. Однако эти внутренние переменные не должны использоваться классами за пределами прикладной среды Flex, поэтому они уточняются пространством имен mx_internal. Например, вспомогательный класс mx.controls.gridclasses.DataGridColumn хранит индекс столбца в переменной mx_internal::colNum.

```
// Файл DataGridColumn.as
mx_internal var colNum:Number;
```

Чтобы получить индекс столбца, класс DataGrid сначала открывает пространство имен mx_internal:

```
use namespace mx_internal;
```

а затем обращается к переменной mx_internal::colNum напрямую, как показано в следующем фрагменте кода, взятого из определения метода-писателя:

```
// Файл DataGrid.as
public function set columns(value:Array):void {
  // Инициализируем "colNum" для всех столбцов
  var n:int = value.length;
  for (var i:int = 0; i < n; i++) {
    var column:DataGridColumn = _columns[i];
    column.owner = this;

    // Обращаемся к переменной mx_internal::colNum напрямую. (Напомним, что
    // пространство имен mx_internal открыто, поэтому выражение
    // column.colNum эквивалентно выражению column.mx_internal::colNum.)
```

```
    column.colNum = i;
  }
  // Оставшаяся часть метода не приводится
}
```

Классы за пределами прикладной среды Flex для получения индекса столбца используют общедоступный метод `getColumnIndex( )` вместо обращения к переменной `mx_internal::colNum` напрямую.

Проблема решена на 9/10. Помещение переменных и методов в пространство имен `mx_internal`, безусловно, уменьшает их непосредственную видимость, однако технически это не запрещает коду за пределами прикладной среды Flex обращаться к ним. Любой разработчик, которому известен идентификатор URI пространства имен `mx_internal`, может применять этот идентификатор для обращения к любой переменной или методу, уточняемому с использованием пространства имен `mx_internal`.

Однако целью пространства имен `mx_internal` является не техническое запрещение использования переменных и методов разработчиком. Скорее оно является большим знаком предупреждения, сообщающим о том, что переменные и методы не предназначены для использования внешним кодом и могут быть изменены без предупреждения или привести к ошибочному поведению, если обращение к ним осуществляется из кода за пределами прикладной среды Flex.

Пример: управление доступом на основании разрешений

Второй пример использования пространств имен демонстрирует вариант механизма управления доступом, когда класс определяет группу методов и переменных, к которым могут обращаться только определенные классы. В этом примере задействованы следующие участники.

Защищаемый класс — класс, который предоставляет доступ к своим защищенным методам и переменным.

Методы и переменные с ограниченным доступом — группа методов и переменных, доступ к которым ограничен.

Авторизованные классы — классы, которым разрешен доступ к методам и переменным с ограниченным доступом.

Рассмотрим базовый код защищаемого класса:

```
package {
  // Это защищаемый класс.
  public class ShelteredClass {
    // Пространство имен restricted уточняет переменные
    // и методы, доступ к которым ограничен.
    private namespace restricted;

    // Это массив авторизованных классов. В данном примере
    // определен только один авторизованный класс: Caller.
    private var authorizedClasses:Array = [ Caller ];
```

```
    // Это переменная с ограниченным доступом.
    // К ней могут обращаться только авторизованные
    // классы.
    restricted var secretData:String = "No peeking";

    // Это метод с ограниченным доступом.
    // К нему могут обращаться только авторизованные
    // классы.
    restricted function secretMethod ( ):void {
      trace("Restricted method secretMethod( ) called");
    }
  }
}
```

Защищаемый класс хранит массив авторизованных классов. Кроме того, в нем объявлено пространство имен с использованием модификатора управления доступом `private`, которое применяется для уточнения методов и переменных с ограниченным доступом. Более того, идентификатор URI для этого пространства имен генерируется автоматически, поэтому его невозможно узнать и использовать за пределами данного класса. Наконец, защищаемый класс определяет сами переменные и методы с ограниченным доступом.

Для обращения к методу или переменной с ограниченным доступом (например, `secretData` или `secretMethod( )`) потенциальный класс должен получить общеизвестные «ключи от парадной двери». Другими словами, он должен получить ссылку на пространство имен, которое уточняет методы и переменные с ограниченным доступом. Однако защищаемый класс предоставит эту ссылку только в том случае, если потенциальный класс — будем называть его «вызывающим классом» — является одним из элементов массива `authorizedClasses`.

В нашем примере вызывающий класс будет просить у класса `ShelteredClass` ссылку на пространство имен `restricted`, используя метод `getRestricted Namespace( )` класса `ShelteredClass`. Метод `getRestrictedAccess( )` принимает экземпляр вызывающего класса в качестве аргумента. Если экземпляр вызывающего класса оказывается авторизованным, метод `getRestrictedNames pace( )` вернет ссылку на пространство имен `restricted`. В противном случае метод вернет значение `null`, которое сообщает о том, что вызывающий класс не имеет права обращаться к методам и переменным с ограниченным доступом. Рассмотрим код метода `getRestrictedNamespace( )`:

```
public function getRestrictedNamespace
                              (callerObject:Object):Namespace {
  // Проверяем, есть ли объект callerObject в массиве authorizedClasses.
  for each (var authorizedClass:Class in authorizedClasses) {
   // Если вызывающий объект является экземпляром авторизованного класса...
    if (callerObject is authorizedClass) {
      // ...возвращаем обратно ссылку на пространство имен restricted
      //  («ключи от парадной двери»)
      return restricted;
    }
  }
```

```
  // Вызывающий объект не является экземпляром
  // авторизованного класса, поэтому
  // запрещаем дальнейшее обращение к переменной
  // и методу с ограниченным доступом.
  return null;
}
```

В листинге 17.5 продемонстрирован весь код класса `ShelteredClass`, включая метод `getRestrictedNamespace( )`.

Листинг 17.5. Класс ShelteredClass

```
package {
  // Это защищаемый класс
  public class ShelteredClass {
    // Пространство имен restricted уточняет переменные
    // и методы, доступ к которым ограничен.
    private namespace restricted;

    // Это массив авторизованных классов. В данном примере
    // определен только один авторизованный класс: Caller.
    private var authorizedClasses:Array = [ Caller ];

    // Это переменная с ограниченным доступом.
    // К ней могут обращаться только авторизованные классы.
    restricted var secretData:String = "No peeking";

    // Это метод с ограниченным доступом.
    // К нему могут обращаться только авторизованные классы.
    restricted function secretMethod ( ):void {
      trace("Restricted method secretMethod( ) called");
    }

    public function getRestrictedNamespace
                                  (callerObject:Object):Namespace {
      // Проверяем, есть ли объект callerObject в массиве authorizedClasses.
      for each (var authorizedClass:Class in authorizedClasses) {
        // Если вызывающий объект является экземпляром
        // авторизованного класса...
        if (callerObject is authorizedClass) {
          // ...возвращаем обратно ссылку на пространство имен restricted
          // («ключи от парадной двери»)
          return restricted;
        }
      }
      // Вызывающий объект не является экземпляром
      // авторизованного класса, поэтому
      // запрещаем дальнейшее обращение к переменной
      // и методу с ограниченным доступом.
      return null;
    }
  }
```

Теперь рассмотрим класс `Caller` — класс, который желает получить доступ к методам и переменным с ограниченным доступом класса `ShelteredClass`. Принимая во внимание значения элементов массива `authorizedClasses` класса `ShelteredClass`, мы знаем, что класс `Caller` является допустимым. В нашем примере `Caller` также является основным классом приложения, поэтому он расширяет класс `Sprite`. Класс `Caller` создает экземпляр класса `ShelteredClass` в своем методе конструктора и присваивает этот экземпляр переменной `shelteredObject`.

```
package {
  import flash.display.*;

  public class Caller extends Sprite {
    private var shelteredObject:ShelteredClass;

    public function Caller ( ) {
      shelteredObject = new ShelteredClass( );
    }
  }
}
```

Чтобы вызвать метод `secretMethod ( )` класса `ShelteredClass`, объект `Caller` должен сначала получить ссылку на пространство имен `restricted`. Для этого объект `Caller` передает себя в метод `getRestrictedNamespace ( )` и присваивает результат (либо пространство имен `restricted`, либо значение `null`) переменной `key` для дальнейшего использования.

```
var key:Namespace = shelteredObject.getRestrictedNamespace(this);
```

Далее, перед тем как вызвать метод `secretMethod ( )`, объект `Caller` проверяет, ссылается ли переменная `key` на допустимое пространство имен. Если это так, объект `Caller` использует переменную `key` в качестве пространства имен для вызова метода `secureMethod ( )`:

```
if (key != null) {
  shelteredObject.key::secureMethod( );
}
```

Для удобства метод с именем `callSecretMethod ( )` нашего класса `Caller` включает код, который вызывает метод `secretMethod ( )`:

```
public function callSecretMethod ( ):void {
  var key:Namespace = shelteredObject.getRestrictedNamespace(this);
  if (key != null) {
    shelteredObject.key:: secretMethod( );
  }
}
```

Листинг 17.6 демонстрирует весь код рассматриваемого класса `Caller`, включая метод `callSecretMethod ( )` и другой удобный метод `displaySecret ( )`, который обращается к переменной `secretData` с ограниченным доступом, используя тот же основной принцип.

Листинг 17.6. Класс Caller

```
package {
  import flash.display.*;
```

```
  public class Caller extends Sprite {

    private var shelteredObject:ShelteredClass;

    public function Caller ( ) {
      shelteredObject = new ShelteredClass( );
      callSecretMethod( );
      displaySecret( );
    }

    public function callSecretMethod ( ):void {
      var key:Namespace = shelteredObject.getRestrictedNamespace(this);
      if (key != null) {
        shelteredObject.key::secretMethod( );
      }
    }

    public function displaySecret ( ):void {
      var key:Namespace = shelteredObject.getRestrictedNamespace(this);
      if (key != null) {
        trace(shelteredObject.key::secretData);
      }
    }
  }
}
```

Пример: реализация режимов работы программы

Последним рассматриваемым примером будет электронный словарь, который позволяет переводить с японского языка на английский и наоборот. Словарь демонстрирует использование режимов программы — область программирования на языке ActionScript, где применяются пространства имен, с самым большим потенциалом. Находясь в «режиме японского языка», словарь возвращает английский перевод для японских слов; находясь в «режиме английского языка», словарь возвращает японский перевод для английских слов. Каждый режим представляется пространством имен: `japanese` для японско-английского режима и `english` для англо-японского режима.

В этом примере задействованы следующие участники:

- `Japanese` — пространство имен для переменных и методов, относящихся к японскому языку;
- `English` — пространство имен для переменных и методов, относящихся к английскому языку;
- `QueryManager` — класс, осуществляющий поиск слов;
- `SearchOptions` — этот класс содержит базовые настройки для операции поиска;
- `JapaneseSearchOptions` — класс, содержащий настройки, характерные для операции поиска на японском языке;

- `EnglishSearchOptions` — данный класс хранит настройки, характерные для операции поиска на английском языке;
- `JEDictionary` — основной класс приложения.

Рассмотрим всех перечисленных участников по отдельности, принимая во внимание, что данный пример не является полнофункциональным, и в тех местах, где должен осуществляться реальный поиск по базе данных, в нем используется код-заполнитель.

Начнем с рассмотрения определений пространств имен `japanese` и `english`, чей код уже должен быть вам знаком:

```
package {
  public namespace english = "http://www.example.com/jedict/english";
}

package {
  public namespace japanese = "http://www.example.com/jedict/japanese";
}
```

Далее идет класс `QueryManager`, который определяет два метода для поиска слова, — `japanese::search( )` и `english::search( )`. Вызов подходящего метода происходит в зависимости от текущего режима программы. Каждый метод `search( )` принимает аргумент `options`, который определяет настройки поиска в виде либо объекта класса `JapaneseSearchOptions`, либо объекта класса `EnglishSearchOptions` соответственно. Далее при рассмотрении класса `JEDictionary` мы увидим, что настройки поиска выбираются в соответствии с текущим режимом программы. Вот код класса `QueryManager`:

```
package {
  public class QueryManager {

    japanese function search (word:String,
                              options:JapaneseSearchOptions):Array {
      trace("Now searching for '" + word + "'.\n"
            + " Match type: " + options.getMatchType( ) + "\n"
            + " English language variant: " + options.getEnglishVariant( ));

      // Расположенный здесь код (не показан) должен выполнять поиск
      // в японско-английском словаре и возвращать результаты,
      // но для эксперимента мы будем просто возвращать предопределенный
      // список результатов:
      return ["English Word 1", "English Word 2", "etc"];
    }

    english function search (word:String,
                             options:EnglishSearchOptions):Array {
      trace("Now searching for '" + word + "'.\n"
            + " Match type: " + options.getMatchType( ) + "\n"
            + " Use kanji in results: " + options.getKanjiInResults( ));

      // Расположенный здесь код (не показан) должен выполнять поиск
      // в англо-японском словаре и возвращать результаты,
```

```
      // но для эксперимента мы будем просто возвращать предопределенный
      // список результатов:
      return ["Japanese Word 1", "Japanese Word 2", "etc"];
    }
  }
}
```

Теперь рассмотрим три класса, предоставляющие настройки поиска: `SearchOptions` и два его подкласса `JapaneseSearchOptions` и `EnglishSearchOptions`. Класс `SearchOptions` задает, с использованием какого режима программа должна выполнять поиск указанной строки: «точного совпадения» (искомое слово должно полностью совпадать со строкой поиска), «совпадает начало» (все искомые слова должны начинаться со строки поиска) или «содержит совпадение» (все искомые слова должны включать строку поиска).

Различные типы совпадений представлены такими константами, как `MATCH_EXACT`, `MATCH_STARTSWITH` и `MATCH_CONTAINS`. Тип совпадений для текущего поиска может быть установлен и получен с помощью методов `setMatchType( )` и `getMatchType( )`. Рассмотрим класс `SearchOptions`:

```
package {
  public class SearchOptions {
    public static const MATCH_EXACT:String = "Exact";
    public static const MATCH_STARTSWITH:String = "StartsWith";
    public static const MATCH_CONTAINS:String = "Contains";

    private var matchType:String;

    public function SearchOptions ( ) {
      // По умолчанию используется режим «точного совпадения».
      setMatchType(SearchOptions.MATCH_EXACT);
    }

    public function getMatchType ( ):String {
      return matchType;
    }

    public function setMatchType (newMatchType:String):void {
      matchType = newMatchType;
    }
  }
}
```

Класс `JapaneseSearchOptions` расширяет класс `SearchOptions`, добавляя настройки, характерные только для японско-английского режима поиска, — определение того, какой вариант английского языка должен использоваться для возвращаемых результатов: U.S. English (американский) или U.K. English (британский). Эти два варианта английского языка представляются константами `ENGLISH_UK` и `ENGLISH_US`. Вариант английского языка для текущего поиска может быть установлен и получен с помощью методов `setEnglishVariant( )` и `getEnglishVariant( )`.

```
package {
  public class JapaneseSearchOptions extends SearchOptions {
    public static const ENGLISH_UK:String = "EnglishUK";
    public static const ENGLISH_US:String = "EnglishUS";

    private var englishVariant:String;

    public function JapaneseSearchOptions ( ) {
      setEnglishVariant(JapaneseSearchOptions.ENGLISH_UK);
    }
    public function getEnglishVariant ( ):String {
      return englishVariant;
    }

    public function setEnglishVariant (newEnglishVariant:String):void {
      englishVariant = newEnglishVariant;
    }
  }
}
```

Как и `JapaneseSearchOptions`, класс `EnglishSearchOptions` расширяет класс `SearchOptions`, добавляя настройки, характерные только для англо-японского режима поиска, — определение того, набор каких символов должен использоваться для возвращаемых результатов: кандзи (набор идеографических символов) или хирагана (набор фонетических символов). Набор символов для текущего поиска может быть установлен или получен с помощью методов `setKanjiInResults( )` и `getKanjiInResults( )`:

```
package {
  public class EnglishSearchOptions extends SearchOptions {
    private var kanjiInResults:Boolean = false;

    public function getKanjiInResults ( ):Boolean {
      return kanjiInResults;
    }

    public function setKanjiInResults (newKanjiInResults:Boolean):void {
      kanjiInResults = newKanjiInResults;
    }
  }
}
```

Наконец, рассмотрим основной класс приложения `JEDictionary`, в котором происходит основная «магия», связанная с использованием пространств имен. Бегло просмотрите код класса, представленный в листинге 17.7, после чего мы рассмотрим его более подробно.

Листинг 17.7. Класс JEDictionary

```
package {
  import flash.display.Sprite;

  public class JEDictionary extends Sprite {
    private var queryMan:QueryManager;
```

```
    japanese var options:JapaneseSearchOptions;
    english  var options:EnglishSearchOptions;

    private var lang:Namespace;

    public function JEDictionary( ) {
      queryMan = new QueryManager( );

      japanese::options = new JapaneseSearchOptions( );
      japanese::options.setMatchType(SearchOptions.MATCH_STARTSWITH);
      japanese::options.setEnglishVariant(JapaneseSearchOptions.ENGLISH_US);

      english::options = new EnglishSearchOptions( );
      english::options.setMatchType(SearchOptions.MATCH_CONTAINS);
      english::options.setKanjiInResults(true);

      // Найти перевод японского слова...
      setModeJapaneseToEnglish( );
      findWord("sakana");

      // Найти перевод английского слова...
      setModeEnglishToJapanese( );
      findWord("fish");
    }

    public function findWord (word:String):void {
      var words:Array = queryMan.lang::search(word, lang::options);
      trace(" Words found: " + words);
    }

    public function setModeEnglishToJapanese ( ):void {
      lang = english;
    }

    public function setModeJapaneseToEnglish ( ):void {
      lang = japanese;
    }
  }
}
```

Итак, основной класс приложения `JEDictionary` расширяет класс `Sprite`:

```
public class JEDictionary extends Sprite {
```

Для поиска класс `JEDictionary` создает экземпляр класса `QueryManager`, который присваивается переменной `queryMan`:

```
private var queryMan:QueryManager;
```

Затем класс `JEDictionary` создает две переменные, каждая из которых имеет локальное имя `options`, уточняемое пространствами имен `japanese` и `english`. Эти переменные хранят настройки поиска, которые будут передаваться в метод экземпляра `search( )` класса `QueryManager`. Обратите внимание, что их типы данных соответствуют типу выполняемого поиска:

```
japanese var options:JapaneseSearchOptions;
english var options:EnglishSearchOptions;
```

После этого идет определение важной переменной `lang`, которая ссылается на пространство имен, соответствующее текущему режиму словаря (японскому или английскому):

```
private var lang:Namespace;
```

Это все, что касается переменных класса `JEDictionary`; теперь рассмотрим его методы: `setModeEnglishToJapanese( )`, `setModeJapaneseToEnglish( )` и `findWord( )`. Они активизируют различные режимы словаря, присваивая переменной `lang` пространство имен `english` или `japanese` соответственно:

```
public function setModeEnglishToJapanese ( ):void {
  lang = english;
}

public function setModeJapaneseToEnglish ( ):void {
  lang = japanese;
}
```

Метод `findWord( )` применяет объект класса `QueryManager` для выполнения поиска в словаре, используя подходящий метод `search( )`. Вызов метода `search( )` является самой важной строкой кода в нашем примере со словарем:

```
queryMan.lang::search(word, lang::options)
```

Обратите внимание, что пространство имен (режим программы) определяет не только тип выполняемого поиска (поведение), но и тип вариантов, используемых для этого поиска (данные). Когда переменной `lang` присвоено пространство имен `japanese`, вызывается метод `japanese::search( )`, в который передается объект `JapaneseSearchOptions`. Когда переменной `lang` присвоено пространство имен `english`, вызывается метод `english::search( )`, в который передается объект `EnglishSearchOptions`.

Результат вызова метода `search( )` присваивается локальной переменной `words`, после чего полученные результаты отображаются в виде отладочного сообщения:

```
public function findWord (word:String):void {
  var words:Array = queryMan.lang::search(word, lang::options);
  trace(" Words found: " + words);
}
```

Для демонстрационных целей метод-конструктор класса `JEDictionary` выполняет поиск по словарю двух предопределенных слов (хотя в полнофункциональном приложении поиск по словарю обычно осуществляется в ответ на введенную пользователем строку). Процедура поиска выполняется экземпляром приложения `QueryManager`, который создается в конструкторе, как показано ниже:

```
queryMan = new QueryManager( );
```

Стандартные настройки для всех японско-английских и англо-японских поисков также задаются в конструкторе:

```
japanese::options = new JapaneseSearchOptions( );
japanese::options.setMatchType(SearchOptions.MATCH_STARTSWITH);
japanese::options.setEnglishVariant(JapaneseSearchOptions.ENGLISH_US);

english::options = new EnglishSearchOptions( );
english::options.setMatchType(SearchOptions.MATCH_CONTAINS);
english::options.setKanjiInResults(true);
```

Для осуществления поиска конструктор устанавливает режим словаря, после чего передает строку поиска в метод экземпляра `findWord( )` класса `JEDictionary`:

```
// Найти перевод японского слова...
setModeJapaneseToEnglish( );
findWord("sakana");

// Найти перевод английского слова...
setModeEnglishToJapanese( );
findWord("fish");
}
```

В зависимости от режима словаря вызывается подходящий метод `search( )` и используются соответствующие варианты поиска.

Мы рассмотрели наш словарь! Кроме того, подошло к концу наше изучение пространств имен. Помните, что вы можете загрузить исходный код для приложения словаря и других примеров из этой главы по адресу **http://www.moock.org/eas3/examples**.

Последние основные темы

Мы почти завершили рассмотрение базовых возможностей языка ActionScript. В двух последующих главах описываются две последние темы: создание и управление XML-данными и ограничения безопасности приложения Flash Player.

Язык XML и расширение E4X

С момента появления приложения Flash Player 5 язык ActionScript включает инструменты для работы со структурированными данными XML. В ActionScript 1.0 и ActionScript 2.0 создание и управление данными XML осуществлялось с помощью переменных и методов внутреннего класса XML (например, `firstChild`, `nextSibling`, `appendChild( )` и т. д.). Класс XML был основан на стандарте Document Object Model (Объектная модель документа) консорциума W3C, или DOM, — стандарте для программного взаимодействия с документами XML (дополнительную информацию об этом стандарте можно найти по адресу http://www.w3.org/DOM).

В языке ActionScript 3.0 набор инструментов для создания и управления данными XML был полностью переработан. ActionScript 3.0 реализует стандарт *ECMAScript for XML* (E4X) — официальное расширение языка ECMA-262 для работы с данными XML в качестве встроенного типа данных. Стандарт E4X призван улучшить использование и гибкость работы с данными XML в языках, основанных на спецификации ECMA-262 (включая языки ActionScript и JavaScript).

Данные XML в виде иерархии

Перед тем как познакомиться с возможностями управления данными XML с помощью расширения E4X, мы должны понять общий принцип представления данных XML в виде иерархии. И прежний класс XML, и расширение E4X представляют данные XML в виде иерархического дерева, в котором каждый элемент и текстовый блок считается узлом дерева (то есть ветвью или листом). Например, рассмотрим фрагмент XML, приведенный в листинге 18.1 (*фрагмент* XML — это часть данных XML, взятых из документа XML).

Листинг 18.1. Пример XML-фрагмента

```
<BOOK ISBN="0141182806">
  <TITLE>Ulysses</TITLE>
  <AUTHOR>Joyce, James</AUTHOR>
  <PUBLISHER>Penguin Books Ltd</PUBLISHER>
</BOOK>
```

Элементы <BOOK>, <TITLE>, <AUTHOR> и <PUBLISHER>, а также текст "Ulysses", "Joyce, James" и "Penguin Books Ltd" считаются узлами дерева, как показано на рис. 18.1.

Элемент <BOOK> является корнем дерева, который еще называется *корневым узлом* структуры данных XML. Любой корректный XML-документ должен иметь всеобъемлющий корневой элемент наподобие элемента <BOOK>, который включает все остальные элементы.

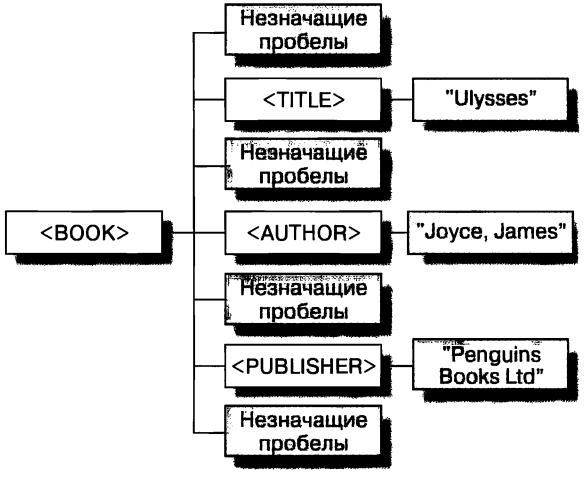

Рис. 18.1. Пример иерархии XML

Когда узел содержится в другом узле, он называется *ребенком* содержащего его узла. С другой стороны, узел, содержащий узел-ребенка, называется его *родителем*. В нашем примере элемент <TITLE> является ребенком элемента <BOOK>, а элемент <BOOK> — родителем элемента <TITLE>.

Как ни удивительно, но элемент <TITLE> — это не первый по счету ребенок элемента <BOOK>, а второй. Первым ребенком фактически являются так называемые *незначащие пробелы* (новая строка и два пробела) в исходном коде фрагмента XML между тегами <BOOK> и <TITLE>. В расширении E4X незначащим пробелом может являться любой из следующих четырех символов форматирования: пробел (\u0020), возврат каретки (\u000D), перевод строки (\u000A) и символ табуляции (\u0009). В дереве документа XML текстовые блоки — даже если они содержат только пробельные символы — считаются узлами дерева. Таким образом, элемент <BOOK> имеет не три ребенка, а семь, четыре из которых являются так называемыми пробельными узлами (текстовые узлы, которые содержат только незначащие пробелы).

Семь детей узла <BOOK> называются *узлами-братьями*, поскольку они находятся на одном уровне в иерархии. Например, мы говорим, что *следующим братом* элемента <TITLE> является пробельный узел, а *предшествующим братом* элемента <AUTHOR> является другой пробельный узел. Вы видите, как текстовые узлы мешают при перемещении по иерархии от одного брата к другому. К счастью, по умолчанию, пробельные узлы игнорируются парсером[1] расширения E4X. Расширение E4X позволяет считать элемент <AUTHOR> следующим братом элемента <TITLE>, что в большинстве случаев нам и нужно. В расширении E4X не придется обрабатывать пробельные узлы самостоятельно до тех пор, пока это действительно не потребуется (за данное поведение отвечает переменная экземпляра ignoreWhitespace класса XML, которая рассматривается далее,

[1] Парсер — синтаксический анализатор.

в подразд. «Преобразование элемента XML в строку» разд. «Преобразование объектов XML и XMLList в строки»).

На последнем уровне иерархии мы видим, что у каждого из узлов <TITLE>, <AUTHOR> и <PUBLISHER> есть один текстовый узел-ребенок: "Ulysses", "Joyce, James" и "Penguin Books Ltd" соответственно. Текстовые узлы являются конечными узлами в дереве.

В исходном коде XML текст, содержащийся в элементе, считается узлом-ребенком данного элемента в соответствующей древовидной иерархии документа XML.

Мы рассмотрели дерево данных XML из листинга 18.1, но по-прежнему ничего не знаем о месте атрибутов в этой иерархии. Можно предположить, что атрибут ISBN элемента <BOOK> превращается в узел-ребенка с именем ISBN. Однако на практике атрибут считается не *ребенком* элемента, определяющего этот атрибут, а его *характеристикой*. Мы рассмотрим способы обращения к атрибутам в расширении E4X далее, в подразд. «Обращение к атрибутам» разд. «Обращение к данным XML».

Теперь, когда известно, как данные XML могут быть представлены в виде концептуальной иерархии, можем рассмотреть способы представления, создания и обработки данных XML в расширении E4X.

Представление данных XML в расширении E4X

В расширении E4X данные XML представляются одним из двух встроенных типов данных языка ActionScript — XML и XMLList, а также их соответствующих классов с такими же именами — XML и XMLList.

Поскольку в расширении E4X появился тип данных XML, существующий класс XML из языков ActionScript 1.0 и ActionScript 2.0 в языке ActionScript 3.0 был переименован в класс XMLDocument и перемещен в пакет flash.mx.

Каждый экземпляр класса XML представляет один из следующих пяти возможных типов XML-содержимого, называемых *типами узлов*:

- элемент;
- атрибут;
- текстовый узел;
- комментарий;
- инструкция обработки.

Если XML-элемент имеет элементы-детей (например, ребенок <AUTHOR> элемента <BOOK>) или текстовые узлы-детей (например, ребенок "Ulysses" элемента <TITLE>), то эти дети представляются в виде экземпляра класса XMLList из

родительского экземпляра класса XML. Каждый экземпляр класса XMLList представляет собой обычную коллекцию, состоящую из одного или более экземпляров класса XML. Например, экземпляр класса XMLList может представлять собой следующее:

- набор атрибутов или элементов, возвращаемых в результате поиска;
- группу фрагментов XML, каждый из которых имеет собственный корневой элемент;
- коллекцию текстовых узлов документа;
- коллекцию комментариев документа;
- коллекцию инструкций обработки документа.

Узлы-дети элемента, представленного экземпляром класса XML, всегда заключаются в экземпляр класса XMLList. Даже если элемент имеет только одного ребенка (скажем, только текстовый узел), этот ребенок все равно будет заключен в экземпляр класса XMLList. Если элемент XML имеет атрибуты, комментарии или инструкции обработки, все они подобным образом заключаются в объект XMLList родительского экземпляра класса XML. Однако комментарии и инструкции обработки по умолчанию игнорируются парсером расширения E4X (чтобы исключить игнорирование этих элементов, присвойте статическим переменным XML.ignoreComments и XML.ignoreProcessingInstructions значение false).

Рассмотрим пример, демонстрирующий, как фрагмент XML представляется экземплярами классов XML и XMLList в расширении E4X. Вспомним исходный код XML из листинга 18.1:

```
<BOOK ISBN="0141182806">
  <TITLE>Ulysses</TITLE>
  <AUTHOR>Joyce, James</AUTHOR>
  <PUBLISHER>Penguin Books Ltd</PUBLISHER>
</BOOK>
```

С точки зрения расширения E4X элемент <BOOK> в данном коде представляется экземпляром класса XML. Он содержит два экземпляра класса XMLList — один для атрибутов элемента <BOOK>, а второй — для его элементов-детей. Элемент <BOOK> имеет только один атрибут, поэтому экземпляр класса XMLList для атрибутов элемента <BOOK> содержит только один экземпляр класса XML (представляющий атрибут ISBN). Экземпляр класса XMLList для элементов-детей элемента <BOOK> содержит три экземпляра класса XML, представляющих три элемента — <TITLE>, <AUTHOR> и <PUBLISHER>. Каждый из этих экземпляров XML отдельно включает в себя экземпляр класса XMLList, содержащий по одному экземпляру класса XML, представляющего соответственно текстовые узлы-дети "Ulysses", "Joyce, James" и "Penguin Books Ltd". Эта структура изображена на рис. 18.2. На рисунке каждый элемент иерархии с корневым элементом <BOOK> обозначен буквой (от A до M), чтобы в дальнейшем было проще ссылаться на эти элементы.

Теперь применим рассмотренную теорию на практике, создав фрагмент с корневым элементом <BOOK> из листинга 18.1 с помощью методик расширения E4X.

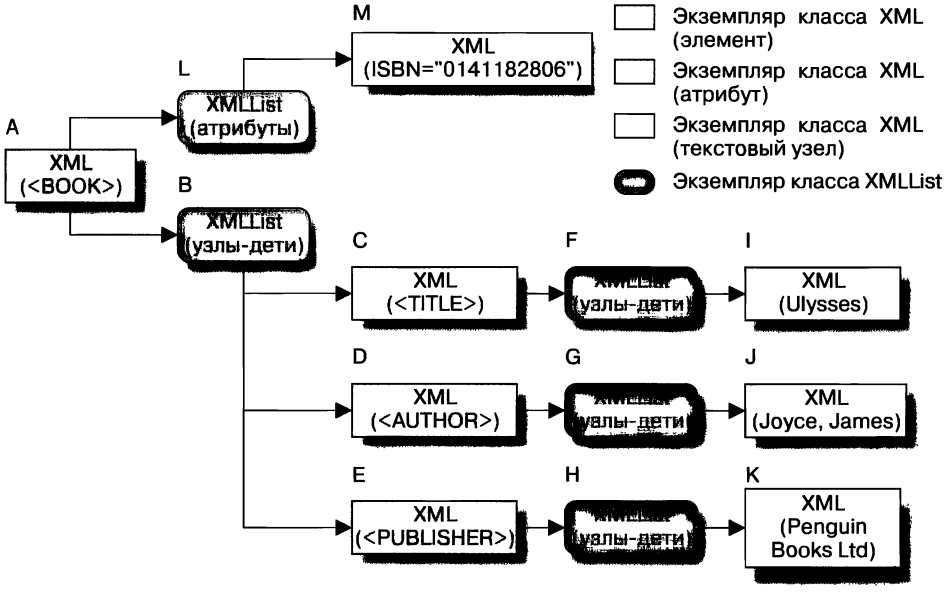

Рис. 18.2. Фрагмент с корневым элементом <BOOK>, представленный в расширении E4X

Создание данных XML с помощью расширения E4X

У нас есть три основных варианта создания фрагмента XML с корневым элементом <BOOK>, представленного в листинге 18.1, с использованием расширения E4X.

- Использовать конструктор класса XML, чтобы создать сначала новый экземпляр класса XML, а затем программным путем — оставшуюся часть фрагмента, применяя методики, описанные далее, в разд. «Изменение или создание нового содержимого XML».
- Использовать конструктор класса XML, чтобы создать новый экземпляр класса XML, а затем импортировать фрагмент из загруженного внешнего файла, как рассматривается далее, в разд. «Загрузка XML-данных».
- Ввести наши XML-данные в форме литерала, как обычную строку или число, в любом месте, где ActionScript допускает использование литералов.

Пока мы воспользуемся третьим подходом — создадим XML-фрагмент с помощью литерала XML. Это демонстрирует листинг 18.2. В нем переменной novel присваивается значение литерала XML (XML-фрагмент из листинга 18.1).

Листинг 18.2. Присваивание литерала XML переменной

```
var novel:XML = <BOOK ISBN="0141182806">
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  </BOOK>;
```

При выполнении предыдущего кода среда Flash создает новый экземпляр класса XML расширения E4X, представляющий литерал фрагмента XML, и присваивает его переменной novel.

Чтобы просмотреть исходный код XML экземпляра класса XML (наподобие экземпляра, на который ссылается переменная novel), используйте метод экземпляра toXMLString( ) класса XML, как показано в следующей строке:

```
trace(novel.toXMLString( ));
```

Метод toXMLString( ) рассматривается далее, в разд. «Преобразование объектов XML и XMLList в строки».

Обратите внимание, что использование переводов строк и кавычек в предыдущем литерале XML является абсолютно нормальным. Компилятор знает, что они являются частью данных XML, и интерпретирует их так, как это необходимо. Где это возможно, компилятор даже конвертирует определенные зарезервированные символы в сущности языка XML. Дополнительную информацию можно найти в подразд. «Использование сущностей XML для специальных символов» разд. «Изменение или создание нового содержимого XML».

Язык ActionScript также позволяет использовать динамические выражения в литерале XML, поэтому названия элементов, атрибутов, значения атрибутов и содержимое элементов можно генерировать программным путем. Чтобы указать динамическое выражение в литерале XML, включите его в фигурные скобки ({ }). Например, следующий код задает название тега <BOOK> динамически:

```
var elementName:String = "BOOK";
var novel:XML = <{elementName}/>;
```

Следующий код представляет слегка усложненный пример, в котором создается иерархия XML, приведенная в листинге 18.2, но при этом все названия элементов, названия атрибутов, значения атрибутов и содержимое элементов задаются динамически.

```
var rootElementName:String = "BOOK";
var rootAttributeName:String = "ISBN";
var childElementNames:Array = ["TITLE", "AUTHOR", "PUBLISHER"];
var bookISBN:String = "0141182806";
var bookTitle:String = "Ulysses";
var bookAuthor:String = "Joyce, James";
var bookPublisher:String = "Penguin Books Ltd";
var novel:XML = <{rootElementName} {rootAttributeName}={bookISBN}>
    <{childElementNames[0]}>{bookTitle}</{childElementNames[0]}>
    <{childElementNames[1]}>{bookAuthor}</{childElementNames[1]}>
    <{childElementNames[2]}>{bookPublisher}</{childElementNames[2]}>
  </{rootElementName}>;
```

Стоит отметить, что, поскольку символы { } применяются для обозначения динамического выражения, их использование в некоторых частях литерала XML недопустимо. В частности, внутри названия элемента, названия атрибута или содержимого элемента для представления символов { и } должны использоваться сущности { и &#x7D соответственно. Тем не менее в виде литерала символы

{ и } могут быть использованы внутри значения атрибута, раздела CDATA, инструкции обработки или комментария.

Теперь, когда у нас появилась переменная `novel`, определенная в листинге 18.2, которая ссылается на фрагмент XML из листинга 18.1, рассмотрим, как можно обращаться к различным частям фрагмента с помощью методик кодирования расширения E4X.

Обращение к данным XML

Расширение E4X предлагает два основных набора инструментов для обращения к данным в иерархии XML:

- методы обращения к содержимому классов `XML` и `XMLList` (`attribute( )`, `attributes( )`, `child( )`, `children( )`, `comments( )`, `descendants( )`, `elements( )`, `parent( )`, `processingInstructions( )` и `text( )`);
- обращение в стиле доступа к переменным с помощью операторов «точка» (`.`), «потомок» (`..`) и «атрибут» (`@`).

Обращение в стиле доступа к переменным предлагается в качестве удобства для программиста и всегда соответствует вызову одного или нескольких методов класса `XML` или `XMLList`. Однако эти два подхода не являются полностью взаимозаменяемыми. Для обращения к следующим типам содержимого должен использоваться подходящий метод класса `XML` или `XMLList`:

- родитель экземпляра класса `XML` (обращение через метод `parent( )`);
- комментарии (обращение через метод `comments( )`);
- инструкции обработки (обращение через метод `processingInstructions( )`);
- элементы или атрибуты, названия которых включают символы, считающиеся недопустимыми в идентификаторе языка ActionScript (обращение через методы `attribute( )`, `child( )`, `descendants( )` или `elements( )`).

Используя наш пример с корневым элементом `<BOOK>`, рассмотрим несколько наиболее распространенных способов обращения к данным XML.

Обращение к корневому узлу XML

В листинге 18.2 мы присвоили фрагмент XML из листинга 18.1 переменной `novel`. Для обращения к корневому элементу `<BOOK>` этого фрагмента (элемент A на рис. 18.2) мы просто используем переменную `novel`. Например, следующий код передает элемент `<BOOK>` (и, как следствие, всех его детей) в гипотетический метод `addToOrder( )`:

```
addToOrder(novel);
```

Обратите внимание, что элемент `<BOOK>` не имеет названия. Иными словами, мы пишем `addToOrder(novel)`, а не так:

```
addToOrder(novel.BOOK); // Неправильно.
addToOrder(novel.child("BOOK")); // Также неправильно.
```

В двух предыдущих примерах элемент <BOOK> ошибочно считается ребенком объекта, на который ссылается переменная novel, хотя это не так. Как обращаться к элементам-детям, мы рассмотрим в следующем разделе.

Стоит отметить, что не существует прямого способа обратиться к корневому узлу из любого заданного ребенка. Однако мы можем использовать метод экземпляра parent( ) (рассматриваемый далее) класса XML, чтобы подняться вверх по иерархии элементов к корневому узлу, как показано в листинге 18.3.

Листинг 18.3. Пользовательский метод для обращения к корневому узлу

```
// Возвращает корневой узел иерархии XML из любого заданного ребенка
public function getRoot (childNode:XML):XML {
  var parentNode:XML = childNode.parent( );
  if (parentNode != null) {
    return getRoot(parentNode);
  } else {
    return childNode;
  }
}

// Применение:
getRoot(некийРебенок);
```

Обращение к узлам-детям

Для обращения к экземпляру класса XMLList, представляющего узлы-детей элемента <BOOK> (элемент B на рис. 18.2), мы используем метод экземпляра children( ) класса XML, не принимающего никаких аргументов. Например:

```
novel.children( )  // Возвращает объект XMLList, представляющий узлы-детей
                   // элемента <BOOK>
```

В качестве альтернативы мы можем обратиться к узлам-детям элемента <BOOK> с помощью более удобного *группового символа свойств* (*) расширения E4X. Например:

```
novel.*  // Также возвращает объект XMLList, представляющий
         // узлы-детей элемента <BOOK>
```

Для обращения к определенному ребенку в объекте XMLList мы используем уже знакомый оператор доступа к элементу массива — [ ]. Например, чтобы обратиться ко второму ребенку элемента <BOOK> — элементу <AUTHOR> (элемент D на рис. 18.2), мы используем:

```
novel.children( )[1]  // Обращаемся ко второму узлу-ребенку элемента <BOOK>
```

или:

```
novel.*[1]  // Также обращаемся ко второму узлу-ребенку элемента <BOOK>
```

Хотя в расширении E4X нет переменной firstChild или lastChild (как в существующем классе XMLDocument), обратиться к первому ребенку в списке узлов-детей можно следующим образом:

```
узел.children( )[0]
```

К последнему ребенку в списке узлов-детей можно обратиться следующим образом:

```
узел.children( )[узел.children( ).length( )-1]
```

Однако обращение к узлу-ребенку по его позиции в списке может быть затруднительным, и поэтому значимость данного метода обращения в расширении E4X была снижена. В расширении E4X обращение к узлам-детям обычно происходит по их именам, а не по позиции. Для обращения к узлу-ребенку по имени применяется метод экземпляра `child( )` класса `XML`, который возвращает объект `XMLList` со всеми элементами-детьми, соответствующими указанному имени. Например, чтобы получить объект `XMLList` со всеми детьми элемента `<BOOK>` с именем `"AUTHOR"`, используется следующий код:

```
novel.child("AUTHOR") // Возвращает все элементы-детей элемента <BOOK>
                      // с именем "AUTHOR"
```

В качестве альтернативы мы можем обратиться к узлам-детям по имени, используя более удобный синтаксис для доступа к переменным расширения E4X. Следующий код возвращает такой же результат, как и предыдущий, но при этом использует более удобный синтаксис обращения к переменной расширения E4X:

```
novel.AUTHOR  // Также возвращает все элементы-детей элемента <BOOK>
              // с именем "AUTHOR"
```

Если элемент `<BOOK>` содержит два элемента `<AUTHOR>`, выражение `novel.AUTHOR` вернет объект `XMLList` с двумя экземплярами класса `XML`, представляющими данные элементы. Для обращения к первому элементу мы могли бы использовать выражение `novel.AUTHOR[0]`. Для обращения ко второму элементу — выражение `novel.AUTHOR[1]`, как показано в следующем коде:

```
var novel:XML = <BOOK>
    <AUTHOR>Jacobs, Tom</AUTHOR>
    <AUTHOR>Schumacher, Jonathan</AUTHOR>
  </BOOK>;

novel.AUTHOR[0];  // Обращение к <AUTHOR>Jacobs, Tom</AUTHOR>
novel.AUTHOR[1];  // Обращение к <AUTHOR>Schumacher, Jonathan</AUTHOR>
```

Безусловно, элемент `<BOOK>` из листинга 18.1 содержит только одного ребенка с именем `"AUTHOR"`, поэтому объект `XMLList`, возвращаемый выражением `novel.AUTHOR`, имеет только один экземпляр класса `XML` (представляющий единственный элемент `<AUTHOR>`). Для обращения к данному элементу `<AUTHOR>` мы могли бы использовать следующий код:

```
novel.AUTHOR[0]  // Обращение к экземпляру элемента <AUTHOR>
```

Однако (и это замечательно!) в большинстве случаев включать `[0]` не требуется. Чтобы сделать обращение к узлу более удобным, расширение E4X реализует специальное поведение для объектов `XMLList`, имеющих только один экземпляр класса `XML` (как в случае с выражением `novel.AUTHOR` в нашем примере). Когда метод класса `XML` вызывается над объектом `XMLList`, имеющим только один экземпляр класса `XML`, этот вызов метода автоматически переадресуется данному экземпляру. Выполняя переадресацию вызова метода, расширение E4X позволяет

программисту рассматривать объект XMLList с одним экземпляром класса XML так, будто данный объект XMLList и является экземпляром класса XML. Как указано в спецификации, расширение E4X «намеренно стирает различие между отдельным объектом класса XML и объектом класса XMLList, содержащим только его».

Предположим, что мы хотим изменить название элемента <AUTHOR> с "AUTHOR" на "WRITER". Мы *могли бы* использовать следующий код, который явно обращается к экземпляру элемента <AUTHOR>:

```
novel.AUTHOR[0].setName("WRITER");
```

Однако обычно используется следующий более удобный код, который неявно обращается к экземпляру элемента <AUTHOR>, опуская оператор обращения к элементу массива ([0], следующий за выражением novel.AUTHOR):

```
novel.AUTHOR.setName("WRITER");
```

Когда мы вызываем метод setName( ) непосредственно над объектом XMLList, возвращаемым выражением novel.AUTHOR, среда выполнения Flash распознает, что данный список содержит только один экземпляр класса XML (<AUTHOR>) и автоматически переадресует вызов метода setName( ) данному экземпляру. В результате название единственного элемента, содержащегося в объекте novel.AUTHOR, изменяется с "AUTHOR" на "WRITER".

В большинстве случаев данный трюк языка ActionScript упрощает написание XML-кода и делает его интуитивно понятным. Однако использовать эту методику следует осторожно. Например, следующий код вызывает метод setName( ) над объектом XMLList, который содержит несколько экземпляров класса XML:

```
var novel:XML = <BOOK>
    <AUTHOR>Jacobs, Tom</AUTHOR>
    <AUTHOR>Schumacher, Jonathan</AUTHOR>
  </BOOK>;
novel.AUTHOR.setName('WRITER');
```

При выполнении предыдущего кода среда Flash сгенерирует следующую ошибку:

```
The setName method works only on lists containing one item.
```

На русском языке она будет выглядеть так: **Метод setName работает только для списков, содержащих один элемент.**

Представление объекта XMLList, содержащего только один экземпляр класса XML, является важным и зачастую неверно трактуемым аспектом программирования с использованием расширения E4X, поэтому мы будем возвращаться к этой теме несколько раз в процессе чтения данной главы.

Обращение к текстовым узлам

Как уже известно из разд. «Данные XML в виде иерархии», текст, содержащийся в элементе, представляется в виде узла в иерархии XML. Например, в следующем XML-фрагменте (повторяемом из листинга 18.2) текст "Ulysses" является текстовым узлом. Он представляется экземпляром класса XML, типом узла которого является текст, как и текстовые узлы "Joyce, James" и "Penguin Books Ltd".

```
var novel:XML = <BOOK ISBN="0141182806">
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  </BOOK>;
```

К текстовым узлам можно обращаться различными способами в зависимости от наших потребностей. Когда необходимо обратиться к текстовому узлу как к экземпляру класса XML, нужно использовать синтаксис обращения к узлу-ребенку, рассмотренный в предыдущем разделе. Например, чтобы обратиться к тексту "Ulysses", который является первым ребенком элемента <TITLE>, мы можем использовать следующий код:

```
novel.TITLE.children( )[0]  // Обращаемся к текстовому узлу Ulysses
```

Или, в качестве альтернативы, мы можем использовать групповой символ свойств:

```
novel.TITLE.*[0] // Тоже обращаемся к текстовому узлу Ulysses
```

Оба предыдущих примера возвращают объект XML (а не строку), который представляет текст элемента "Ulysses". Мы можем вызывать методы класса XML над этим объектом точно так же, как и над любым другим объектом XML. Например:

```
novel.TITLE.*[0].parent( )    // Обращение к элементу <TITLE>
novel.TITLE.*[0].nodeKind( )  // Возвращает строку "text"
novel.TITLE.*[0].toString( )  // Возвращает строку "Ulysses"
```

Однако если мы хотим просто обратиться к содержимому текстового узла, как к значению типа String, а не к экземпляру класса XML, то можем использовать метод экземпляра toString( ) класса XML над его родительским элементом. Для таких элементов, как, например, <TITLE>, которые содержат только один текстовый узел-ребенок (без других промежуточных элементов), метод toString( ) возвращает текст этого узла-ребенка, опуская начальные и конечные теги родительского элемента. Таким образом, выражение novel.TITLE.toString( ) вернет строку "Ulysses":

```
trace(novel.TITLE.toString( ));  // Выводит: Ulysses
```

Обдумывая предыдущую строку кода, не забудьте, что на самом деле это сокращенный вариант следующего кода:

```
trace(novel.TITLE[0].toString( ));  // Выводит: Ulysses
```

Сокращенный вариант выражения novel.TITLE.toString( ) возвращает значение "Ulysses", поскольку среда выполнения Flash знает, что объект XMLList, на который ссылается выражение novel.TITLE, имеет всего один экземпляр класса XML (<TITLE>), и автоматически переадресует вызов метода toString( ) данному экземпляру.

При обращении к содержимому текстового узла как к значению типа String мы можем опускать явный вызов метода toString( ), поскольку среда выполнения Flash вызывает метод toString( ) автоматически, когда вместо строки используется нестроковое значение. Например, функция trace( ) в качестве аргумента принимает строку, поэтому вместо явного вызова метода toString( ):

```
trace(novel.TITLE.toString( ));  // Выводит: Ulysses
```

мы можем позволить среде выполнения Flash вызвать этот метод неявно:

```
trace(novel.TITLE);  // Также выводит: Ulysses
```

Подобным образом при присваивании содержимого текстового узла `Ulysses` переменной типа `String` вместо использования такого полностью явного кода:

```
var titleName:String = novel.TITLE[0].toString( );
```

мы можем использовать просто:

```
var titleName:String = novel.TITLE;
```

Замечательная возможность. И зачастую именно этот способ применяется для получения текста, содержащегося в элементе, в расширении E4X.

Для текстовых узлов, которые разбросаны между другими элементами, можно использовать метод экземпляра `text( )` класса `XML`, чтобы получить текстовые узлы, не содержащиеся в элементах. Чтобы проиллюстрировать данную возможность, временно добавим элемент `<DESCRIPTION>` к элементу `<BOOK>`, как показано в следующем примере:

```
var novel:XML = <BOOK ISBN="0141182806">
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
    <DESCRIPTION>A <B>very</B> thick book.</DESCRIPTION>
  </BOOK>;
```

Элемент `<DESCRIPTION>` содержит как элементы-детей, так и текстовые узлы-детей:

- `A` (текстовый узел);
- `<B>very</B>` (элемент с текстовым узлом-ребенком);
- `thick book.` (текстовый узел).

Чтобы получить объект `XMLList`, содержащий два текстовых узла `A` и `thick book.`, мы используем следующее выражение:

```
novel.DESCRIPTION.text( )
```

Для обращения к этим текстовым узлам используется оператор доступа к элементу массива:

```
trace(novel.DESCRIPTION.text( )[0]);  // Выводит: A
trace(novel.DESCRIPTION.text( )[1]);  // Выводит: thick book.
```

Метод `text( )` также можно применять для получения текстовых узлов из всего объекта `XMLList`, а не только из одного элемента `XML`. Предположим, что у нас есть объект `XMLList`, представляющий всех детей элемента `<BOOK>` из листинга 18.2 (до момента добавления элемента `<DESCRIPTION>`):

```
novel.*
```

Чтобы поместить текстовые узлы каждого ребенка из этого списка в объект `XMLList` для последующей обработки, например, для создания пользовательского интерфейса, применяется следующий код:

```
novel.*.text( )
```

И снова для обращения к текстовым узлам мы используем оператор доступа к элементу массива:

```
trace(novel.*.text( )[0]);  // Выводит: Ulysses
trace(novel.*.text( )[1]);  // Выводит: Joyce, James
trace(novel.*.text( )[2]);  // Выводит: Penguin Books Ltd
```

Однако метод экземпляра `text( )` класса `XMLList` оказывается менее полезным при использовании над списком элементов, который содержит и текстовые узлы-детей, и элементы-детей. Если узел содержит и текстовые узлы-детей, и элементы-детей (например, узел <DESCRIPTION>), то возвращается только первый текстовый узел-ребенок; остальные дети игнорируются. Например:

```
var novel:XML = <BOOK ISBN="0141182806">
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
    <DESCRIPTION>A <B>very</B> thick book.</DESCRIPTION>
  </BOOK>;


trace(novel.*.text( )[3]); // Выводит: A
                           // остальные узлы-дети – <B>very</B>
                           // и thick book. – игнорируются
```

Обращение к узлам-родителям

Для обращения к узлу-родителю данного узла используется метод экземпляра `parent( )` класса `XML`, не принимающий аргументов. Предположим, что переменная `pub` содержит ссылку на элемент <PUBLISHER> из листинга 18.2.

```
var pub:XML = novel.PUBLISHER[0];
```

Для обращения к родителю узла <PUBLISHER> (которым является <BOOK>) мы используем следующий код:

```
pub.parent( )
```

Метод `parent( )` может так же успешно применяться для обращения к любому узлу-предку, как показано в следующем коде:

```
// Создаем иерархию XML с тремя уровнями.
var doc:XML = <grandparent><parent><child></child></parent></grandparent>;

// Присваиваем ссылку на элемент <child>
var kid:XML = doc.parent.child[0];

// Используем метод parent( ) для последовательного обращения к элементу
// <grandparent> из элемента <child>
var grandparent:XML = kid.parent( ).parent( );
```

В отличие от методов children( ) и child( ), метод экземпляра parent( ) класса XML не имеет альтернативного синтаксиса в стиле обращения к переменным.

Когда метод `parent( )` применяется к экземпляру класса `XMLList`, он возвращает значение `null`, если только родителем всех элементов списка не является один и тот же элемент, который и возвращается в этом случае. Например, в следующем коде мы получаем экземпляр класса `XMLList`, представляющий три ребенка элемента `<BOOK>`, а затем вызываем метод `parent( )` над этим списком. Поскольку родителем всех трех детей является один и тот же элемент, возвращается этот родитель.

```
var bookDetails:XMLList = novel.*;
var book:XML = bookDetails.parent( );  // Возвращает элемент <BOOK>
```

Вызов метода `parent( )` над объектом `XMLList`, содержащим всего один экземпляр класса `XML`, идентичен вызову метода `parent( )` над этим экземпляром. Например, следующие две строки кода являются тождественными:

```
novel.PUBLISHER[0].parent( )  // Обращается к <BOOK>
novel.PUBLISHER.parent( )     // Также обращается к <BOOK>
```

Когда метод `parent( )` вызывается над экземпляром класса `XML`, представляющим атрибут, возвращается элемент, для которого этот атрибут определен. Это демонстрирует следующий код, в котором используется еще не рассмотренный метод доступа к атрибутам (этот метод будет описан в ближайшее время):

```
novel.@ISBN.parent( )  // Возвращает элемент <BOOK>
```

Обращение к узлам-братьям

Как уже известно из разд. «Данные XML в виде иерархии», узел-брат — это узел, который находится непосредственно возле другого узла на данном уровне иерархии XML. Например, в знакомой нам иерархии `<BOOK>` элемент `<TITLE>` является предшествующим братом элементов `<AUTHOR>` и `<PUBLISHER>` и следующим братом элемента `<AUTHOR>`.

```
var novel:XML = <BOOK ISBN="0141182806">
    <TITLE>Ulysses</TITLE>                    <!--Предшествующий брат -->
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>  <!--Следующий брат  -->
  </BOOK>;
```

В расширении E4X отсутствует встроенная поддержка перемещений между узлами-братьями в иерархии XML. Переменные `nextSibling` и `previousSibling`, поддерживаемые в модели DOM, не являются частью интерфейса API расширения E4X. Однако следующий брат любого узла может быть получен с помощью следующего кода, предполагая, что у данного узла есть узел-родитель:

```
некийУзел.parent( ).*[некийУзел.childIndex( )+1];
```

Предшествующий брат может быть найден с помощью следующего кода:

```
некийУзел.parent( ).*[некийУзел.childIndex( )-1];
```

Например, данный код обращается к предшествующему и следующему братьям элемента `<AUTHOR>`:

```
var author:XML = novel.AUTHOR[0];
// Предшествующий брат
trace(author.parent( ).*[author.childIndex( )-1]); // Выводит: Ulysses
// Следующий брат
```

```
trace(author.parent( ).*[author.childIndex( )+1]); // Выводит:
                                                   // Penguin Books Ltd
```

В листинге 18.4 представлен код пользовательского метода для обращения к предшествующему брату данного узла. Обратите внимание, что метод добавляет код для проверки того, что указанный узел действительно имеет предшествующего брата, перед тем как вернуть его.

Листинг 18.4. Пользовательский метод previousSibling( )

```
public function previousSibling (theNode:XML):XML {
  // Проверяем, что узел действительно имеет предшествующего брата,
  // перед тем как вернуть его
  if (theNode.parent( ) != null && theNode.childIndex( ) > 0) {
    return theNode.parent( ).*[theNode.childIndex( )-1];
  } else {
    return null;
  }
}

// Использование:
previousSibling(некийУзел);
```

В листинге 18.5 определен метод `nextSibling( )` — метод-компаньон пользовательского метода `previousSibling( )`, определенного в листинге 18.4. Обратите внимание, что метод добавляет код для проверки, что указанный узел действительно имеет следующего брата, перед тем как вернуть его.

Листинг 18.5. Пользовательский метод nextSibling( )

```
public function nextSibling (theNode:XML):XML {
  if (theNode.parent( ) != null
      && theNode.childIndex( ) < theNode.parent( ).children( ).length( )-1) {
    return theNode.parent( ).*[theNode.childIndex( )+1];
  } else {
    return null;
  }
}

// Использование:
nextSibling(некийУзел);
```

Расширение E4X уменьшает необходимость доступа к узлам-братьям, поскольку основное внимание уделяется обращению к элементам по их имени. Например, для обращения к элементу <TITLE> с помощью расширения E4X мы обычно используем простую запись novel.TITLE, а не author.parent( ).*[author.childIndex( )–1].

Обращение к атрибутам

Для обращения к объекту `XMLList`, представляющему все атрибуты элемента, используется метод экземпляра `attributes( )` класса `XML`, не принимающий аргументов. Он имеет следующий общий вид:

```
некийЭлемент.attributes( )
```

Например, следующий код возвращает объект XMLList, представляющий атрибуты элемента <BOOK> (элемент L на рис. 18.2):

```
novel.attributes( )
```

В качестве альтернативы для обращения к объекту XMLList, представляющему атрибуты элемента, можно использовать более удобный синтаксис *специального символа атрибутов* (@*) расширения E4X, который записывается следующим образом:

```
некийЭлемент.@*  // Возвращает объект XMLList, представляющий все атрибуты
                 // элемента некийЭлемент
```

Например, приведенный ниже код, который является эквивалентом выражения novel.attributes( ), возвращает объект XMLList, представляющий атрибуты элемента <BOOK> (элемент L на рис. 18.2):

```
novel.@*
```

Как и в случае с элементами, для обращения к атрибутам в объекте XMLList можно использовать оператор доступа к элементу массива ([ ]). Например, следующий код обращается к первому и единственному атрибуту элемента <BOOK> — ISBN (элемент M на рис. 18.2):

```
novel.attributes( )[0]
```

Следующий код также обращается к первому атрибуту элемента <BOOK> (снова ISBN), но использует синтаксис специального символа атрибутов расширения E4X:

```
novel.@*[0]
```

Однако ни выражение novel.@*[0], ни выражение novel.attributes( )[0] не представляют обычный код расширения E4X. В расширении E4X обращение к атрибутам редко происходит по их порядковому номеру в документе XML. Обычно к атрибутам обращаются по имени, используя либо метод attribute( ), либо более удобный синтаксис обращения к переменным расширения E4X. Для обращения к атрибуту по его имени с помощью метода attribute( ) используется следующий обобщенный код:

```
некийЭлемент.attribute("имяАтрибута")
```

Данный код возвращает объект XMLList, содержащий атрибут с именем *имяАтрибута* элемента *некийЭлемент*. Например, следующий код возвращает объект XMLList, который содержит один экземпляр класса XML, представляющий атрибут ISBN (элемент M на рис. 18.2) элемента <BOOK>:

```
novel.attribute("ISBN")
```

Данное выражение является эквивалентом обращения к атрибуту по имени с использованием синтаксиса обращения к переменной:

```
некийЭлемент.@имяАтрибута
```

Например, следующий код также вернет объект XMLList, содержащий один экземпляр класса XML, который представляет атрибут ISBN элемента <BOOK>, но в данном случае используется синтаксис обращения к переменной:

```
novel.@ISBN
```

Как и `child( )`, метод `attribute( )` возвращает объект `XMLList`, содержащий экземпляры класса `XML`, которые соответствуют указанному имени. Однако, поскольку два или более атрибута одного элемента не должны иметь одинаковые имена, объект `XMLList`, возвращаемый методом `attribute( )`, всегда содержит только один экземпляр класса `XML` (представляющий атрибут с указанным именем).

Для обращения к экземпляру класса `XML`, который содержится в объекте `XMLList`, возвращаемом выражением `novel.@ISBN`, мы *могли бы* использовать следующий код:

```
novel.@ISBN[0]
```

Но при вызове метода класса `XML` над данным экземпляром мы обычно опускаем оператор обращения к элементу массива (`[0]`), как показано в следующем коде:

```
novel.@ISBN.некийМетодКлассаXML( )
```

Можно опустить запись `[0]`, поскольку, как мы уже знаем, когда метод класса `XML` вызывается над объектом `XMLList`, содержащим только один экземпляр этого класса, вызов метода автоматически переадресуется данному экземпляру. Например, следующий явный код:

```
novel.@ISBN[0].parent( )  // Возвращает узел <BOOK>
```

является эквивалентом такого неявного кода:

```
novel.@ISBN.parent( )  // Также возвращает узел <BOOK>
```

С другой стороны, экземпляры класса `XML`, представляющие атрибуты, никогда не имеют детей и, следовательно, большая часть методов класса `XML` остается невостребованной. Вместо этого экземпляр класса `XML`, представляющий атрибут, используется всего лишь для хранения значения данного атрибута. Для обращения к значению атрибута используется метод экземпляра `toString( )` класса `XML`. Например, следующий код присваивает значение атрибута `ISBN` элемента `<BOOK>` переменной `bookISBN`, используя полностью явное выражение:

```
var bookISBN:String = novel.@ISBN[0].toString( );
```

Однако не забывайте, что мы можем вызвать метод `toString( )` непосредственно над результатом выражения `novel.@ISBN` (вместо выражения `novel.@ISBN[0]`), поскольку возвращаемый объект `XMLList` содержит только один экземпляр класса `XML`. Вот более короткая и более типичная запись:

```
var bookISBN:String = novel.@ISBN.toString( );  // Опущена запись [0]
```

Приведенную строку кода можно сделать еще короче. Класс `XML` является динамическим. Таким образом, мы можем использовать возможность автоматического преобразования типов данных языка ActionScript, чтобы преобразовать значение любой переменной экземпляра класса `XML` к строке (правила преобразования типов данных языка ActionScript описаны в гл. 8):

```
var bookISBN:String = novel.@ISBN;
```

В данном коде переменная `novel` представляет экземпляр динамического класса (`XML`). Следовательно, когда мы присваиваем типизированной переменной `bookISBN` значение переменной `ISBN` этого экземпляра, компилятор откладывает проверку типов до этапа выполнения программы. На этапе выполнения, поскольку

тип данных переменной bookISBN является примитивным (String), значение переменной ISBN автоматически преобразуется к этому типу.

Достаточно удобно. Эту возможность можно использовать и для преобразования к другим примитивным типам данных. Например, следующий код преобразует значение атрибута ISBN к числу путем простого присваивания этого значения переменной с типом данных Number:

```
var bookISBN:Number = novel.@ISBN;
```

При работе с атрибутами следует помнить, что значение атрибута всегда имеет тип String, даже когда кажется, что с логической точки зрения оно должно иметь другой тип. Для использования этого значения в качестве другого типа данных (не String) необходимо выполнить его явное или неявное преобразование. Чтобы избежать неприятных сюрпризов, следует постоянно помнить о правилах преобразования типов данных, которые были рассмотрены в гл. 8. В частности, запомните, что строковое значение "false" преобразуется в значение true типа Boolean! По этой причине при работе с атрибутами, которые хранят булеву информацию, проще использовать сравнение строк, чем преобразовывать значение атрибута к типу данных Boolean.

Например, следующий код добавляет новый атрибут INSTOCK, который обозначает доступность книги в настоящий момент, к элементу <BOOK>. Чтобы отобразить сообщение о доступности книги, сравним значение выражения novel.@INSTOCK со строкой "false" вместо того, чтобы преобразовывать значение выражения novel.@INSTOCK в значение типа Boolean. Перед сравнением в качестве меры предосторожности мы также преобразуем все символы значения атрибута к нижнему регистру.

При сравнении атрибутов помните, что они всегда являются строками, а сравнение выполняется с учетом регистра символов.

```
var novel:XML = <BOOK ISBN="0141182806" INSTOCK="false">
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  </BOOK>;

// Сравниваем со строкой "false" вместо преобразования в тип Boolean
if (novel.@INSTOCK.toLowerCase( ) == "false") {
  trace("Not Available!");
} else {
  trace("Available!");
}
```

Обращение к комментариям и инструкциям обработки

Двумя последними типами узлов, к которым можно обращаться с помощью расширения E4X, являются комментарии и инструкции обработки. Комментарии языка XML имеют следующий вид:

```
<!-- Здесь находится текст комментария -->
```

Инструкции обработки языка XML принимают такой вид:

<?некоеЦелевоеПриложение некиеДанные?>

Для обращения к этим двум вспомогательным типам данных можно использовать методы экземпляра `comments( )` и `processingInstructions( )` класса `XML`. Оба метода возвращают объект `XMLList`, представляющий всех непосредственных детей элемента, которые являются либо комментариями, либо инструкциями обработки соответственно. Однако по умолчанию парсер расширения E4X игнорирует и комментарии, и инструкции обработки. Чтобы получить доступ к комментариям документа XML или фрагмента XML, перед обработкой данных переменной `XML.ignoreComments` необходимо присвоить значение `false`, как показано в следующем коде:

```
XML.ignoreComments = false;
```

Подобным образом, для того чтобы получить доступ к инструкциям обработки документа XML или фрагмента XML, перед обработкой данных переменной `XML.ignoreProcessingInstructions` необходимо присвоить значение `false`, как показано в следующем коде:

```
XML.ignoreProcessingInstructions = false;
```

Следует обратить внимание на то, что переменные `XML.ignoreComments` и `XML.ignoreProcessingInstructions` являются статическими, значения им присваиваются через класс `XML`, а не через отдельные экземпляры класса `XML`. Значения, присвоенные переменным `XML.ignoreComments` и `XML.ignoreProcessingInstructions`, влияют на все последующие операции обработки XML-данных.

В листинге 18.6, например, с корневым узлом `<BOOK>` добавляются два комментария и две инструкции обработки, а также показано, как обращаться к добавленным элементам. Обратите внимание, что переменным `XML.ignoreComments` и `XML.ignoreProcessingInstructions` присваивается значение `false` *до того*, как литерал XML будет присвоен переменной `novel`. Обратите также внимание, что, хотя комментарии и инструкции обработки разбросаны между детьми элемента `<BOOK>`, методы `comments( )` и `processingInstructions( )` игнорируют других детей и возвращают список, состоящий только из комментариев и инструкций обработки.

Листинг 18.6. Обращение к комментариям и инструкциям обработки

```
XML.ignoreComments = false;
XML.ignoreProcessingInstructions = false;

// Создаем XML-фрагмент, который содержит комментарии и инструкции обработки
var novel:XML = <BOOK ISBN="0141182806">
    <!--Hello world-->
    <?app1 someData?>
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <?app2 someData?>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
    <!--Goodbye world-->
```

```
  </BOOK>
trace(novel.comments( )[0]);                    // <!--Hello world-->
trace(novel.comments( )[1]);                    // <!--Goodbye world-->
trace(novel.processingInstructions( )[0]);  // <?app1 someData?>
trace(novel.processingInstructions( )[1]);  // <?app2 someData?>
```

Чтобы получить объект `XMLList`, представляющий все комментарии и инструкции обработки для всего дерева XML (а не только непосредственных детей узла), используйте оператор «потомок» в сочетании с групповым символом свойств, как показано в следующем коде:

```
var tempRoot:XML = <tempRoot/>;
tempRoot.appendChild(novel);
trace(tempRoot..*.comments( )[0]);  // Первый комментарий в документе
```

Мы рассмотрим представленный метод более подробно далее, в разд. «Обход деревьев XML».

Обращение к атрибутам и элементам с зарезервированными символами в именах

Если имя атрибута или элемента содержит символ, который считается недопустимым для использования в идентификаторах языка ActionScript (например, дефис), вы не сможете обратиться к этому атрибуту или элементу с помощью оператора «точка». Вместо этого необходимо использовать метод `attribute( )`, `child( )` или оператор `[ ]`. Например:

```
var saleEndsDate:XML = <DATE TIME-ZONE="PST">February 1, 2006</DATE>
trace(saleEndsDate.@TIME-ZONE);              // НЕДОПУСТИМО! Не делайте так.
trace(saleEndsDate.attribute("TIME-ZONE"));  // Допустимо. Делайте так.
trace(saleEndsDate.@["TIME-ZONE"]);          // Тоже допустимо.
```

В случае с недопустимым кодом `saleEndsDate.@TIME-ZONE` среда выполнения Flash трактует дефис как операцию вычитания и интерпретирует выражение как `saleEndsDate.@TIME` *минус* `ZONE`! По всей видимости, переменная (или метод) с именем `ZONE` не существует, поэтому среда выполнения Flash сгенерирует следующее сообщение об ошибке:

```
Access of undefined property 'ZONE'
```

По-русски это будет выглядеть так: Обращение к неопределенному свойству 'ZONE'.

Однако если бы переменная `ZONE` существовала, ее значение было бы вычтено из пустого объекта `XMLList`, представленного выражением `saleEndsDate.@TIME`, и никакой ошибки не возникло бы! Без сообщений об ошибках неправильное обращение к элементу `saleEndsDate.@TIME-ZONE` будет очень сложно выявить. Принимая во внимание, что атрибут `saleEndsDate.@TIME` не существует, нам бы хотелось, чтобы среда выполнения Flash сгенерировала ошибку «несуществующего атрибута», но, к сожалению, в версии спецификации E4X, реализованной в языке ActionScript 3.0, оговаривается, что в результате обращения к несуществующим атрибутам должен возвращаться пустой объект `XMLList`, не приводя к возникновению ошибки. Будущие версии языка ActionScript могут исправить эту ситуацию.

Мы завершили рассмотрение базовых приемов обращения к данным XML. Перед тем как продолжить изучение расширения E4X, еще раз вернемся к важной теме, касающейся интерпретации экземпляра класса XMLList в качестве экземпляра класса XML.

Интерпретация объекта XMLList как экземпляра класса XML

Как мы уже знаем, в расширении E4X ссылка на объект XMLList, содержащий всего один экземпляр класса XML, может рассматриваться как ссылка на этот экземпляр. Например, мы видели, что выражение:

```
novel.AUTHOR[0].setName("WRITER");
```

эквивалентно выражению:

```
novel.AUTHOR.setName("WRITER");  // Опущен [0]
```

Они являются эквивалентными, потому что выражение novel.AUTHOR ссылается на объект XMLList, содержащий всего один экземпляр класса XML (элемент <AUTHOR>).

Интерпретация экземпляра класса XMLList как экземпляра класса XML делает использование расширения E4X более простым и удобным, но при этом порождает некоторые сбивающие с толку детали, особенно когда эта возможность применяется вместе с автоматическим преобразованием строк. Поближе познакомимся с этой проблемой.

Предположим, что мы создаем пользовательский интерфейс для книжного интернет-магазина, где каждая книга представлена XML-фрагментом, структура которого совпадает со структурой нашего примера с фрагментом <BOOK>. Когда пользователь выбирает книгу, на экране появляется соответствующее имя автора.

В нашем коде мы создаем метод displayAuthor( ), который выводит имя автора. В первой реализации метода мы будем требовать, чтобы имя автора передавалось в виде строки:

```
public function displayAuthor (name:String):void {
  // authorField ссылается на экземпляр класса TextField,
  // в котором отображается имя автора
  authorField.text = name;
}
```

Когда пользователь выбирает книгу, мы получаем имя ее автора из элемента <AUTHOR> и передаем его в метод displayAuthor( ), используя код наподобие следующего:

```
displayAuthor(novel.AUTHOR);
```

Данная инструкция проста и интуитивно понятна, но, как мы уже знаем из этой главы, «за кадром» происходит очень много действий. В качестве повторения проанализируем, как это работает. Во-первых, среда выполнения Flash передает выражение novel.AUTHOR в метод displayAuthor( ) в качестве значения параметра name. Типом данных параметра name является String, поэтому среда Flash автоматически пытается преобразовать значение выражения novel.AUTHOR в строку, используя следующее выражение:

```
novel.AUTHOR.toString( )
```

По умолчанию вызов метода `toString( )` над объектом возвращает строку в формате `[object ИмяКласса]`, но выражение `novel.AUTHOR` представляет экземпляр класса `XMLList`, а класс `XMLList` переопределяет метод `toString( )` собственной версией. В частности, версия метода `toString( )` класса `XMLList` узнает, что значение выражения `novel.AUTHOR` содержит только один элемент, поэтому возвращается результат вызова метода `toString( )` класса `XML` над этим элементом. Таким образом, вызов `novel.AUTHOR.toString( )` автоматически преобразуется в вызов `novel.AUTHOR[0].toString( )`. Что же является возвращаемым значением выражения `novel.AUTHOR[0].toString( )`? Как мы уже знаем, ответ основан на факте, что выражение `novel.AUTHOR[0]` представляет простой элемент XML, который не содержит других элементов-потомков. Для элемента XML, не содержащего других элементов, метод `toString( )` класса `XML` возвращает текстовый узел данного элемента в виде строки, исключая охватывающие теги. Значит, выражение `novel.AUTHOR[0].toString( )` в качестве окончательного значения, передаваемого в метод `displayAuthor( )`, вернет значение `"Joyce, James"` (а не `"<AUTHOR>Joyce, James</AUTHOR>"`).

Подведем итоги.

- Передача значения выражения `novel.AUTHOR` в качестве параметра типа `String` приводит к неявному преобразованию значения выражения `novel.AUTHOR` в строку.
- Значение выражения `novel.AUTHOR` преобразуется в строку с помощью вызова `novel.AUTHOR.toString( )`.
- Выражение `novel.AUTHOR.toString( )` автоматически возвращает результат выражения `novel.AUTHOR[0].toString( )`, поскольку значение выражения `novel.AUTHOR` представляет экземпляр класса `XMLList` с одним-единственным элементом.
- Выражение `novel.AUTHOR[0].toString( )` возвращает текст, содержащийся в элементе `<AUTHOR>` (`"Joyce, James"`), в соответствии с реализацией метода `toString( )` класса `XML` (дополнительную информацию можно найти далее, в разд. «Преобразование объектов XML и XMLList в строки»).

После всего сказанного и сделанного выражение

```
displayAuthor(novel.AUTHOR);
```

преобразуется в выражение

```
displayAuthor("Joyce, James");
```

которое мы и предполагали увидеть.

В большинстве случаев мы можем игнорировать предыдущее усложнение, поскольку расширение E4X, говоря ненаучным языком, «делает то, что подразумевается». Однако существуют моменты, когда необходимо понимать действия «автопилота» в лице расширения E4X, чтобы осуществлять «ручное управление». Предположим, мы решили, что в книжном магазине должно выводиться не только имя, но и дата рождения каждого автора. Изменим структуру нашего фрагмента XML, чтобы включить дату рождения в качестве ребенка элемента `<AUTHOR>`, как показано в следующем коде:

```
var novel:XML = <BOOK ISBN="0141182806">
  <TITLE>Ulysses</TITLE>
  <AUTHOR>
    <NAME>Joyce, James</NAME>
    <BIRTHDATE>February 2 1882</BIRTHDATE>
  </AUTHOR>
  <PUBLISHER>Penguin Books Ltd</PUBLISHER>
</BOOK>
```

Соответственно изменим метод `displayAuthor( )`, чтобы он принимал элемент `<AUTHOR>` в качестве параметра и получал имя и дату рождения автора непосредственно из элементов-детей `<NAME>` и `<BIRTHDATE>`:

```
public function displayAuthor (author:XML):void {
  authorField.text = "Name: " + author.NAME
                   + " Birthdate: " + author.BIRTHDATE;
}
```

Обратите внимание, что в данном коде тип данных параметра был изменен с типа `String` на `XML`. Если сейчас мы попытаемся передать значение выражения `novel.AUTHOR` в метод `displayAuthor( )`, на этапе выполнения возникнет ошибка несоответствия типов, поскольку среда Flash не может осуществить неявное преобразование значения выражения `novel.AUTHOR` (которое является объектом `XMLList`) к экземпляру класса `XML`:

```
displayAuthor(novel.AUTHOR); //Ошибка #1034: Ошибка операции приведения
                             //типов: невозможно преобразовать XMLList в XML
```

Чтобы исправить данную ошибку, мы должны явно ссылаться на экземпляр класса `XML`, представляющий элемент `<AUTHOR>`, при передаче его в метод `displayAuthor( )`, как показано в следующем коде:

```
displayAuthor(novel.AUTHOR[0]);  // Передаем единственный экземпляр класса
                                 // XML, принадлежащий значению выражения
                                 // novel.AUTHOR, в метод displayAuthor( )
```

Обратите внимание на важное отличие: когда мы хотим обратиться к тексту, содержащемуся в элементе `<AUTHOR>`, как к значению типа `String`, мы можем положиться на автоматическое поведение расширения E4X. Однако когда мы хотим обратиться к настоящему экземпляру класса `XML`, представляющему элемент `<AUTHOR>`, то должны явно ссылаться на этот экземпляр.

Теперь предположим, что нас попросили изменить наш магазин для поддержки книг с несколькими авторами. Мы снова изменим структуру нашего XML-фрагмента, чтобы включить несколько элементов `<AUTHOR>`. В листинге 18.7 представлен пример фрагмента XML, демонстрирующий новую структуру (даты рождения авторов являются вымышленными).

Листинг 18.7. Фрагмент `<BOOK>` с несколькими авторами

```
var oopBook:XML = <BOOK ISBN="0596007124">
  <TITLE>Head First Design Patterns</TITLE>
  <AUTHOR>
    <NAME>Eric Freeman</NAME>
    <BIRTHDATE>January 1 1970</BIRTHDATE>
```

```
  </AUTHOR>
  <AUTHOR>
    <NAME>Elisabeth Freeman</NAME>
    <BIRTHDATE>January 1 1971</BIRTHDATE>
  </AUTHOR>
  <AUTHOR>
    <NAME>Kathy Sierra</NAME>
    <BIRTHDATE>January 1 1972</BIRTHDATE>
  </AUTHOR>
  <AUTHOR>
    <NAME>Bert Bates</NAME>
    <BIRTHDATE>January 1 1973</BIRTHDATE>
  </AUTHOR>
  <PUBLISHER>O'Reilly Media, Inc</PUBLISHER>
</BOOK>;
```

Для работы с новой структурой XML мы изменим метод `displayAuthor( )`, чтобы он принимал объект `XMLList`, представляющий несколько элементов `<AUTHOR>` (а не один элемент `<AUTHOR>` из предыдущего примера). В новой версии метода `displayAuthor( )` для перемещения по элементам `<AUTHOR>` используется инструкция `for-each-in` (мы рассмотрим использование инструкции `for-each-in` далее, в разд. «Обработка данных XML с помощью циклов for-each-in и for-in»).

```
public function displayAuthor (authors:XMLList):void {
  for each (var author:XML in authors) {
    authorField.text += "Name: " + author.NAME
                  + ", Birthdate: " + author.BIRTHDATE + "\n";
  }
}
```

Чтобы передать список элементов `<AUTHOR>` в метод `displayAuthor( )`, мы используем следующий код:

```
displayAuthor(oopBook.AUTHOR);
```

Эта строка кода соответствует нашему первоначальному подходу, а именно:

```
displayAuthor(novel.AUTHOR);
```

На этот раз объект `XMLList` передается непосредственно в метод `displayAuthor( )` без преобразования, поскольку типом данных получаемого параметра является `XMLList`, а не `String`. И снова обратите внимание на отличие: если при передаче объекта `XMLList` в функцию мы хотим преобразовать список в значение типа `String`, то для получаемого параметра указываем тип данных `String` и позволяем «колдовать» расширению E4X. Однако если мы хотим сохранить тип данных списка, то должны указать `XMLList` в качестве типа данных получаемого параметра. И сама ссылка (`oopBook.author`), и тип данных получаемого параметра (`authors`) оказывают влияние на поведение кода.

В табл. 18.1 представлены результаты передачи различных выражений расширения E4X, которые мы уже рассмотрели, в качестве параметров различных типов данных.

Таблица 18.1. Обзор: выражения расширения E4X и результаты

Выражение	Тип данных параметра	Результат
novel.AUTHOR	String	"Joyce, James"
novel.AUTHOR	XML	Ошибка несоответствия типов (невозможно преобразовать тип XMLList к типу XML)
novel.AUTHOR[0]	String	"Joyce, James"
novel.AUTHOR[0]	XML	Экземпляр класса XML, представляющий элемент <AUTHOR>
oopBook.AUTHOR	String	Строка, содержащая исходный код XML для четырех элементов <AUTHOR>
oopBook.AUTHOR	XMLList	Объект XMLList с четырьмя экземплярами класса XML, представляющими четыре элемента <AUTHOR>

Не беспокойтесь. Расширение E4X отлично продумано. Пусть его автоматическое поведение не тревожит вас. Большую часть времени оно будет помогать вам. Тем не менее при обращении к узлам XML с использованием синтаксиса обращения к переменным (оператор «точка») имейте в виду следующие потенциальные источники недоразумений.

- Выражение *узелРодитель*.*имяУзлаРебенка* является эквивалентным выражению *узелРодитель*.`child(`*имяУзлаРебенка*`)` и всегда ссылается на экземпляр класса `XMLList`, а не на экземпляр класса `XML`.
- Когда экземпляр класса `XMLList` содержит только один экземпляр класса `XML`, методы класса `XML` могут вызываться над экземпляром класса `XMLList`. Он автоматически переадресует эти вызовы экземпляру класса `XML`.
- Чтобы получить объектную ссылку на экземпляр класса `XML`, который содержится в элементе *узелРодитель*.*имяУзлаРебенка*, необходимо использовать выражение вида *узелРодитель*.*имяУзлаРебенка*`[`*индекс*`]`, даже если нужный экземпляр класса `XML` является единственным элементом в объекте `XMLList` (в этом случае для обращения к элементу используется выражение *узелРодитель*.*имяУзлаРебенка*`[0]`).
- Если XML-элемент содержит только текст (и *не* содержит других элементов-детей), в результате преобразования этого элемента в строку будет возвращен содержащийся в нем текст без окружающих тегов (например, в результате преобразования элемента `<TITLE>Ulysses</TITLE>` в строку будет возвращена строка `"Ulysses"`, а не `"<TITLE>Ulysses</TITLE>"`).
- Если XML-элемент содержит текст и элементы-детей, в результате его преобразования в строку будет получен исходный код данного элемента вместе с тегами. Например, в результате преобразования элемента

  ```
  <AUTHOR>Joyce, <FIRSTNAME>James</FIRSTNAME></AUTHOR>
  ```

  в строку будет получено

  ```
  "<AUTHOR>Joyce, <FIRSTNAME>James</FIRSTNAME></AUTHOR>"
  ```

  а не

  ```
  Joyce, James
  ```

Если вы сомневаетесь, то можете воспользоваться методами класса `XML` для обращения к интересующему вас содержимому. Явные названия методов класса `XML` иногда легче понять, несмотря на более громоздкую запись.

Обработка данных XML с помощью циклов for-each-in и for-in

Структурированные документы в формате XML зачастую содержат наборы данных, которые должны обрабатываться систематично. Например, XML-документ может содержать информацию о населении разных стран мира или точек на карте либо стоимость товаров в заказе. Независимо от данных, базовый подход одинаков: каждый элемент должен быть обработан и использован приложением определенным способом. Чтобы упростить процесс обработки информации в формате XML, расширение E4X вводит в язык ActionScript новый вид цикла, называемый циклом `for-each-in`.

Как уже рассказывалось в гл. 15, цикл `for-each-in` обеспечивает простой доступ к значениям динамических переменных экземпляра объекта или элементам массива. Напомним обобщенный синтаксис цикла `for-each-in`:

```
for each (значениеПеременнойИлиЭлемента in некийОбъект) {
  инструкции
}
```

Мы можем применять приведенный синтаксис для обработки экземпляров класса XML, содержащихся в объекте XMLList, так же легко, как для обработки элементов массива или динамических переменных экземпляра объекта. Это продемонстрировано в листинге 18.8.

Листинг 18.8. Использование цикла for-each-in для обработки экземпляров класса XML

```
var novel:XML = <BOOK ISBN="0141182806">
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  </BOOK>;

for each (var child:XML in novel.*) {
  trace(child);
}
```

Цикл `for-each-in` в листинге 18.8 выполняется три раза, по одному для каждого узла-ребенка в объекте XMLList, который возвращается выражением `novel.*`. При выполнении цикла в первый раз переменной `child` присваивается ссылка на первый узел-ребенок элемента <BOOK> (то есть на экземпляр класса XML, представляющий элемент <TITLE>). При выполнении цикла во второй раз переменной `child` присваивается ссылка на второй узел-ребенок элемента <BOOK> (экземпляр класса XML, представляющий элемент <AUTHOR>). При выполнении цикла в третий раз переменной `child` присваивается ссылка на третий узел-ребенок элемента <BOOK> (экземпляр класса XML, представляющий элемент <PUBLISHER>). Результат выполнения цикла выглядит следующим образом:

```
Ulysses
Joyce, James
Penguin Books Ltd
```

В листинге 18.9 представлен более сложный сценарий — подсчет общей стоимости заказа покупателя. Разобраться в этом коде помогут комментарии.

Листинг 18.9. Подсчет общей стоимости заказа

```
// Формируем заказ. В обычной ситуации заказ формируется программным путем
// в ответ на действия пользователя, но в этом примере мы включим
// в код уже сформированный заказ.
var order:XML = <ORDER>
  <ITEM SKU="209">
    <NAME>Trinket</NAME>
    <PRICE>9.99</PRICE>
    <QUANTITY>3</QUANTITY>
  </ITEM>

  <ITEM SKU="513">
    <NAME>Gadget</NAME>
    <PRICE>149.99</PRICE>
    <QUANTITY>1</QUANTITY>
  </ITEM>

  <ITEM SKU="374">
    <NAME>Toy</NAME>
    <PRICE>39.99</PRICE>
    <QUANTITY>2</QUANTITY>
  </ITEM>
</ORDER>

// Создаем текстовое поле, в котором будет отображаться описание заказа.
var outField:TextField = new TextField( );
outField.width = 300;
outField.height = 300;
outField.text = "Here is your order:\n";
addChild(outField);

// Изначально общая стоимость заказа равна 0.
var total:Number = 0;

// Этот цикл выполняется по одному разу для каждого элемента <ITEM>.
for each (var item:XML in order.*) {
  // Отображаем описание данного элемента в текстовом поле outField.
  outField.text += item.QUANTITY
                + " " + item.NAME + "(s)."
                + " $" + item.PRICE + " each.\n";

  // Прибавляем стоимость этого элемента к общей стоимости заказа.
  // Обратите внимание, что операция умножения автоматически преобразует
  // значения количества и стоимости в числа.
  total += item.QUANTITY * item.PRICE;
}

// Отображаем общую стоимость заказа.
outField.appendText("TOTAL: " + total);
```

Вот результат выполнения кода из листинга 18.9:

```
Here is your order:
3 Trinket(s). $9.99 each.
1 Gadget(s). $149.99 each.
2 Toy(s). $39.99 each.
TOTAL: 259.94
```

Рассмотрим последний пример, демонстрирующий возможности управления содержимым заказа с помощью цикла `for-each-in`. Этот код присваивает одно и то же значение всем элементам `<PRICE>` из листинга 18.9:

```
// Большая РАСПРОДАЖА!
// Все по цене $1!
for each (var item:XML in order.*) {
  item.PRICE = 1;
}
```

Более подробно возможность изменения содержимого элемента XML будет рассмотрена далее, в разд. «Изменение или создание нового содержимого XML».

Ошибочно полагать, что имена переменных экземпляров класса `XML` в объекте `XMLList` совпадают с названиями соответствующих элементов XML. Вместо этого экземпляры класса `XML` в объекте `XMLList`, как и элементы массива, расположены по порядку и в качестве имен переменных используются их порядковые номера. Это демонстрируется в следующем коде с помощью цикла `for-in`. Имена 0, 1 и 2 представляют порядковый номер каждого экземпляра класса `XML` в объекте `XMLList`, возвращаемом выражением `order.*`.

```
for (var childName:String in order.*) {
  trace(childName);
}

// Вывод:
// 0
// 1
// 2
```

Более подробную информацию по инструкциям for-each-in и for-in можно найти в гл. 15.

Обращение к потомкам

Мы уже достаточно попрактиковались в обращении к узлам-детям элемента XML. Теперь рассмотрим обращение не только к узлам-детям данного элемента, но и к так называемым *узлам-потомкам*. Потомками элемента являются все узлы, содержащиеся в этом элементе, на любом уровне иерархии XML (то есть узлы-внуки, узлы-правнуки и т. д.).

Например, рассмотрим XML-фрагмент из листинга 18.10, представляющий взятые из библиотеки книги и фильмы.

Листинг 18.10. Запись о взятых из библиотеки источниках

```
var loan:XML = <LOAN>
  <BOOK ISBN="0141182806" DUE="1136091600000">
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  </BOOK>

  <DVD ISBN="0790743086" DUE="1136610000000">
    <TITLE>2001 A Space Odyssey</TITLE>
    <DIRECTOR>Stanley Kubrick</DIRECTOR>
    <PUBLISHER>Warner Home Video</PUBLISHER>
  </DVD>

  <DVD ISBN="078884461X" DUE="1137214800000">
    <TITLE>Spirited Away</TITLE>
    <DIRECTOR>Hayao Miyazaki</DIRECTOR>
    <PUBLISHER>Walt Disney Video</PUBLISHER>
  </DVD>
</LOAN>
```

В данном примере к потомкам элемента <LOAN> относятся:

- непосредственные дети: элемент <BOOK> и два элемента <DVD>;
- внуки: все элементы <TITLE>, <AUTHOR>, <PUBLISHER> и <DIRECTOR>;
- правнуки: все текстовые узлы, содержащиеся в элементах <TITLE>, <AUTHOR>, <PUBLISHER> и <DIRECTOR>.

Для обращения к потомкам элемента используется оператор «потомок» (..) расширения E4X, записываемый в следующем виде:

элемент..идентификатор

Выражение для обращения к потомкам возвращает объект XMLList, представляющий всех потомков элемента *элемент*, имена которых соответствуют имени *идентификатор*. Например, следующее выражение возвращает объект XMLList, содержащий два экземпляра класса XML, которые представляют два элемента <DIRECTOR> из листинга 18.10:

```
loan..DIRECTOR
```

Обратите внимание, что элементы <DIRECTOR> не являются непосредственными детьми элемента <LOAN>; они являются его внуками. Оператор «потомок» предоставляет непосредственный, простой доступ к узлам, находящимся в любом месте иерархии XML. Например, чтобы получить список всех элементов <TITLE> в записи о взятых из библиотеки источниках, мы используем следующий код:

```
loan..TITLE
```

Чтобы вывести названия всех взятых источников, мы можем использовать код наподобие следующего:

```
trace("You have borrowed the following items:");
for each (var title:XML in loan..TITLE) {
  trace(title);
}
```

```
// Вывод:
You have borrowed the following items:
Ulysses
2001 A Space Odyssey
Spirited Away
```

Это очень удобно!

Выражение a.b возвращает список всех непосредственных элементов-детей с именем b; выражение a..b возвращает список всех элементов-потомков с именем b. Синтаксис намеренно сделан похожим — единственным отличием является глубина возвращаемых узлов.

Оператор «потомок» работает и с атрибутами. Чтобы получить список атрибутов-потомков, а не элементов, используется запись следующего вида:

элемент`..@`*имяАтрибута*

Например, следующее выражение возвращает объект `XMLList`, имеющий три экземпляра класса `XML`, которые представляют три атрибута `DUE` из листинга 18.10.

```
loan..@DUE
```

Вот еще один удобный фрагмент кода:

```
trace("Your items are due on the following dates:");
for each (var due:XML in loan..@DUE) {
  trace(new Date(Number(due)));
}

// В часовом поясе стандартного восточного времени будут выведено следующее:
Your items are due:
Sun Jan 1 00:00:00 GMT-0500 2006
Sat Jan 7 00:00:00 GMT-0500 2006
Sat Jan 14 00:00:00 GMT-0500 2006
```

Чтобы получить объект `XMLList`, включающий все узлы, которые являются потомками данного элемента, используйте следующую запись:

элемент`..*`

Например, следующий код возвращает объект `XMLList` и 21 потомок элемента `<LOAN>`:

```
loan..*
```

Можете определить всех потомков? Все они представлены в листинге 18.11, нарисованном с помощью удобной псевдографики. Позиция каждого узла в объекте `XMLList`, возвращаемом выражением `loan..*`, выделена круглыми скобками. Вы не забыли текстовые узлы, правда? Помните, что они считаются потомками?

Листинг 18.11. Узлы, возвращаемые выражением loan..*

```
BOOK (1)
  |-TITLE (2)
  |   |-Ulysses (3)
  |
```

```
  |-AUTHOR (4)
  |   |-Joyce, James (5)
  |
  |-PUBLISHER (6)
      |-Penguin Books Ltd (7)

DVD (8)
  |-TITLE (9)
  |   |-2001 A Space Odyssey (10)
  |
  |-AUTHOR (11)
  |   |-Stanley Kubrick (12)
  |
  |-DIRECTOR (13)
      |-Warner Home Video (14)

DVD (15)
  |-TITLE (16)
  |   |-Spirited Away (17)
  |
  |-AUTHOR (18)
  |   |- Hayao Miyazaki (19)
  |
  |-DIRECTOR (20)
      |- Walt Disney Video (21)
```

Чтобы получить объект `XMLList`, который включает все атрибуты, определенные как в самом элементе, так и во всех его потомках, используется запись следующего вида:

элемент`..@*`

Например, следующий код возвращает объект `XMLList`, который содержит все атрибуты, определенные в потомках элемента <LOAN> (всего шесть). Стоит отметить, что если бы в элементе <LOAN> были определены атрибуты (у элемента <LOAN> они отсутствуют), они были бы включены в этот список.

```
loan..@*
```

Следующий код выводит атрибуты, возвращаемые выражением `loan..@*`, с помощью цикла `for-each-in`. Для каждого атрибута отображается его имя и значение, а также содержимое элемента-ребенка <TITLE> его родителя.

```
for each (var attribute:XML in loan..@*) {
  trace(attribute.parent( ).TITLE
        + ": " + attribute.name( ) + "=" + attribute);
}

// Вывод:
Ulysses: ISBN=0141182806
Ulysses: DUE=1136091600000
2001 A Space Odyssey: ISBN=0790743086
2001 A Space Odyssey: DUE=1136610000000
Spirited Away: ISBN=078884461X
Spirited Away: DUE=1137214800000
```

Чтобы получить объект `XMLList`, который включает все атрибуты, определенные в потомках элемента, *исключая* атрибуты самого элемента, применяется запись следующего вида:

```
элемент..*.@*
```

или следующий более громоздкий, но менее загадочный код:

```
элемент..*.attributes( )
```

Предыдущий код читается следующим образом: «вызвать метод экземпляра `attributes( )` класса `XMLList` над объектом `XMLList`, который представляет потомки элемента *элемент*». Результатом этого выражения является объект `XMLList`, представляющий все атрибуты, определенные в потомках элемента *элемент*. Чтобы освежить в памяти описание метода `attributes( )`, обратитесь к подразд. «Обращение к атрибутам» разд. «Обращение к данным XML».

Для обращения к атрибутам или элементам, имена которых содержат символы, считающиеся недопустимыми в идентификаторах языка ActionScript, вместо оператора «потомок» необходимо использовать метод экземпляра descendants( ) класса XML. Оператор «потомок» не позволяет использовать запись вида элемент..["некоеИмя"].

Оператор «потомок» полезен сам по себе, однако при использовании возможностей фильтрации расширения E4X он становится просто незаменимым. Как только вы разберетесь с оператором «потомок» и возможностями фильтрации расширения E4X, сможете быстро и легко выполнять практически любую обработку данных в формате XML. Мы рассмотрим это в следующем разделе.

Фильтрация данных XML

Оператор фильтрующего предиката расширения E4X — это простой, но очень мощный инструмент поиска. Он может принимать любой объект `XMLList` и возвращать подмножество элементов из этого списка в соответствии с указанным условием.

Термин «предикат» заимствован из спецификации языка XPath консорциума W3C. Более подробную информацию можно найти по адресу http://www.w3.org/TR/xpath20/#id-predicates.

Оператор фильтрующего предиката записывается в следующем обобщенном виде:

```
объектXMLList.(условноеВыражение)
```

Для каждого элемента в объекте *объектXMLList* выражение *условноеВыражение* выполняется один раз. Если для элемента выражение *условноеВыражение* возвращает значение `true`, этот элемент добавляется в объект `XMLList`, который возвращается после обработки всех элементов. Стоит отметить, что при каждом выполнении выражения *условноеВыражение* текущий элемент временно добавляется в начало цепочки областей видимости, позволяя внутри выражения непосредственно обращаться к его потомкам и атрибутам по имени.

Использовать оператор фильтрующего предиката чрезвычайно просто. Рассмотрим новый XML-фрагмент и выполним фильтрацию. В листинге 18.12 новый фрагмент представляет список сотрудников компании.

Листинг 18.12. Список сотрудников

```
var staff:XML = <STAFF>
  <EMPLOYEE ID="501" HIRED="1090728000000">
    <NAME>Marco Crawley</NAME>
    <MANAGER>James Porter</MANAGER>
    <SALARY>25000</SALARY>
    <POSITION>Designer</POSITION>
   </EMPLOYEE>

   <EMPLOYEE ID="500" HIRED="1078462800000">
     <NAME>Graham Barton</NAME>
     <MANAGER>James Porter</MANAGER>
     <SALARY>35000</SALARY>
     <POSITION>Designer</POSITION>
   </EMPLOYEE>

   <EMPLOYEE ID="238" HIRED="1014699600000">
     <NAME>James Porter</NAME>
     <MANAGER>Dorian Schapiro</MANAGER>
     <SALARY>55000</SALARY>
     <POSITION>Manager</POSITION>
  </EMPLOYEE>
</STAFF>
```

Теперь выполним нашу первую операцию по фильтрации данных: предположим, что мы хотим получить список сотрудников, которыми руководит James Porter. Мы можем отфильтровать список элементов <EMPLOYEE> из листинга 18.12 следующим образом:

```
// Сначала получаем объект XMLList, представляющий все элементы <EMPLOYEE>
var allEmployees:XMLList = staff.*;

// Теперь фильтруем список элементов <EMPLOYEE>
var employeesUnderJames:XMLList = allEmployees.(MANAGER == "James Porter");
```

Выражение `allEmployees.(MANAGER == "James Porter")` возвращает объект `XMLList`, включающий все элементы из фрагмента `allEmployees`, у которых элемент <MANAGER> содержит текст `"James Porter"`. Вам должна понравиться простота и читабельность кода расширения E4X. Просто помните, что предыдущая строка кода работает потому, что каждый раз при выполнении выражения `(MANAGER == "James Porter")` проверяемый элемент из фрагмента `allEmployees` добавляется в цепочку областей видимости. Таким образом, выполняемое выражение `(MANAGER == "James Porter")` имеет следующее концептуальное значение, записанное в псевдокоде:

```
if (текущийСотрудник.MANAGER == "James Porter")
// добавить текущийСотрудник в список результатов
```

Для сравнения приведем реальный код на языке ActionScript, который делает то же самое, что и выражение `allEmployees.(MANAGER == "James Porter")`:

```
var resultList:XMLList = new XMLList( );
var counter:int = 0;
for each (var employee:XML in allEmployees) {
  if (employee.MANAGER == "James Porter") {
    resultList[counter] = employee;
    counter++;
  }
}
```

Рассмотрим еще несколько примеров, которые демонстрируют, как можно обращаться к информации из листинга 18.12, основываясь на множестве условий. Следующее выражение возвращает список сотрудников, зарплата которых меньше либо равна $35 000.

```
staff.*.(SALARY <= 35000)
```

Следующее выражение возвращает список сотрудников с зарплатой от $35 000 до 50 000:

```
staff.*.(SALARY >= 35000 && SALARY <= 50000)
```

Это выражение возвращает список дизайнеров, работающих в компании:

```
staff.*.(POSITION == "Designer")
```

Следующее выражение возвращает список сотрудников с идентификационным номером 238 (так получилось, что в этом примере только один сотрудник с указанным номером, но данный элемент все равно помещается в экземпляр класса `XMLList`).

```
staff.*.(@ID == "238")
```

Следующий код получает список сотрудников, которые работают в компании с 2004 года (для представления времени мы используем стандартный формат «миллисекунды-с-1970-года», применяемый в классе `Date`):

```
staff.*.(@HIRED >= 1072933200000 && @HIRED <= 1104555600000)
```

Наконец, мы выводим дату, когда был нанят сотрудник Graham:

```
// В часовом поясе стандартного восточного времени будет отображено:
// Fri Mar 5 00:00:00 GMT-0500 2004
trace(new Date(Number(staff.*.(NAME == "Graham Barton").@HIRED)));
```

Забавно, не правда ли? Предикаты — это сильно!

Чтобы отфильтровать список, в котором не каждый элемент имеет заданный атрибут или элемент-ребенка, мы должны использовать метод hasOwnProperty( ) для проверки существования данного атрибута или ребенка перед применением фильтра. В противном случае возникнет ошибка обращения. Например, следующий код возвращает все элементы документа некийДокумент, у которых для атрибута color установлено значение "red":

некийДокумент..*.(hasOwnProperty("@color") && @color == "red")

Мы познакомились с множеством способов обращения к конкретным узлам и группам узлов внутри XML-документа. Далее рассмотрим использование обхода дерева для обращения не к определенным узлам документа, а к *каждому* его узлу.

Обход деревьев XML

В терминах программирования *обход* означает обращение к каждому узлу в структуре данных и его обработку. *Обход дерева* означает использование рекурсивного алгоритма для обхода узлов древовидной структуры.

В реализациях языка XML, основанных на модели DOM (например, унаследованный класс `flash.xml.XMLDocument`), для поиска информации программисты часто пишут собственный код, осуществляющий обход деревьев XML. Например, программа для работы с персоналом может включать код, который просматривает весь документ XML в поисках сотрудников с должностью «руководитель» или с зарплатой, находящейся в заданном интервале. Как мы уже видели в предыдущем разделе, такой пользовательский код для обхода деревьев почти не нужен в расширении E4X, поскольку большинство операций поиска могут быть выполнены с помощью операторов «потомок» и фильтрующего предиката расширения E4X. Однако бывают ситуации, в которых необходимо выполнить обход дерева даже в расширении E4X. К счастью, для этих целей код расширения E4X прост.

В расширении E4X мы можем использовать оператор «потомок» вместе с групповым символом свойств, чтобы получить объект `XMLList`, содержащий все узлы-потомки для данного элемента, как показано в следующем коде:

```
некийЭлемент..*  // Возвращает список, содержащий все потомки
                 // элемента некийЭлемент
```

Можно также использовать инструкцию `for-each-in` для перемещения по всем элементам объекта `XMLList`. Объединив эти методики вместе, мы легко обойдем любой узел в дереве XML, как показано в следующем коде:

```
for each (var ребенок:XML in некийЭлемент..*) {
  // обработка элемента ребенок...
}
```

Однако стоит отметить, что предыдущий код не в полном объеме соответствует классическому определению обхода дерева, поскольку он никогда не обращается к корневому элементу иерархии (*некийЭлемент*). Если корневой узел должен обрабатываться вместе с его детьми, просто временно добавьте его к другому экземпляру класса `XML`, как показано ниже:

```
var tempRoot:XML = <root/>;
tempRoot.appendChild(некийЭлемент);
for each (var ребенок:XML in tempRoot..*) {
  // Обработка элемента ребенок...
}
```

Рассмотрим простой пример реального обхода дерева. Предположим, что мы создаем форум, который позволяет пользователям оставлять ответы, включающие

разметку HTML. Чтобы ответы соответствовали стандарту XHTML, мы хотим, чтобы все имена тегов, найденные в ответах пользователей, преобразовывались в нижний регистр. Вот пример ответа, который включает проблематичный верхний регистр и смешанные регистры в именах тегов:

```
var message:XML = <message>
<B>HEY!</B> I just wanted to say that your site is so cool!!
You should <A Href="http://mysiteiscooltoo.com/">visit mine</A> sometime.
</message>;
```

Рассмотрим код для обхода дерева, который преобразует имена всех элементов и атрибутов из предыдущего XML-фрагмента в нижний регистр:

```
for each (var child:XML in message..*) {
  // Если узел является элементом...
  if (child.nodeKind( ) == "element") {
    // ...преобразуем его имя в нижний регистр.
    child.setName(child.name( ).toString( ).toLowerCase( ));
    // Если узел имеет атрибуты, преобразуем их имена в нижний регистр.
    for each (var attribute:XML in child.@*) {
      attribute.setName(attribute.name( ).toString( ).toLowerCase( ));
    }
  }
}
```

Вот новый фрагмент XML, полученный в результате выполнения предыдущего кода, — все имена тегов и атрибутов преобразованы в нижний регистр.

```
<message>
<b>HEY!</b>I just wanted to say that your site is so cool!!
You should <a href="http://mysiteiscooltoo.com/">visit mine</a> sometime.
</message>
```

На протяжении всей этой главы мы знакомились со способами обращения к содержимому существующих XML-документов. Теперь рассмотрим, как создавать или изменять это содержимое.

Изменение или создание нового содержимого XML

В расширении E4X наиболее распространенные операции добавления и изменения существующего экземпляра класса `XML` могут быть выполнены с помощью обычных операторов присваивания. Тем не менее результаты операторов присваивания расширения E4X отличаются и зависят от типа присваиваемого значения и типа целевого объекта присваивания. Рассмотрим различные сценарии по отдельности.

Изменение содержимого элемента

Чтобы изменить содержимое XML-элемента, мы присваиваем ему любое значение, не являющееся объектом `XMLList` или `XML`. Это значение преобразуется в строку и заменяет текущее содержимое элемента. Вспомним наш фрагмент `<BOOK>`:

```
var novel:XML = <BOOK ISBN="0141182806">
  <TITLE>Ulysses</TITLE>
  <AUTHOR>Joyce, James</AUTHOR>
  <PUBLISHER>Penguin Books Ltd</PUBLISHER>
</BOOK>
```

Чтобы изменить содержимое элемента `<TITLE>` со строки `"Ulysses"` на строку `"The Sun Also Rises"`, мы используем следующий код:

```
novel.TITLE[0] = "The Sun Also Rises";
```

Однако не забывайте, что расширение E4X по возможности позволяет интерпретировать объект `XMLList` в виде объекта `XML`. Поскольку объект `XMLList`, возвращаемый выражением `novel.TITLE`, имеет один-единственный экземпляр класса `XML`, мы можем использовать следующий более удобный код:

```
novel.TITLE = "The Sun Also Rises";  // Опущено [0]
```

Однако если бы объект `XMLList`, возвращаемый выражением `novel.TITLE`, содержал несколько элементов `<TITLE>`, операция присваивания имела бы другой результат, описываемый далее, в подразд. «Присваивание значений объекту XMLList» разд. «Изменение или создание нового содержимого XML».

Если вы хотите вспомнить отличие между выражением novel.TITLE[0] и novel.TITLE, обратитесь к подразд. «Интерпретация объекта XMLList как экземпляра класса XML» разд. «Обращение к данным XML».

Теперь также изменим имя автора и название издательства:

```
novel.AUTHOR = "Hemingway, Ernest";
novel.PUBLISHER = "Scribner";
```

В качестве альтернативы содержимое элемента можно изменить с помощью метода экземпляра `setChildren( )` класса `XML`. Например:

```
novel.TITLE.setChildren("The Sun Also Rises");
```

Изменение значения атрибута

Чтобы изменить значение атрибута XML, просто присвойте атрибуту любое новое значение, используя оператор присваивания. Новое значение будет преобразовано в строку и заменит существующее значение атрибута. Например, следующий код изменяет значение атрибута `ISBN` со строки `"0141182806"` на строку `"0684800713"`:

```
novel.@ISBN = "0684800713";
```

В этом коде использование строки вместо числа в качестве нового значения позволяет сохранить первый нуль.

Если присваиваемое атрибуту значение является объектом `XMLList`, содержащим атрибуты, то значения атрибутов из этого объекта `XMLList` будут объединены в одну строку (в качестве разделителей используются пробелы), которая затем будет присвоена атрибуту. Это слегка необычное поведение может использоваться для объединения группы атрибутов в один атрибут. Например:

```
var books:XML = <BOOKS>
    <BOOK ISBN="0141182806"/>
    <BOOK ISBN="0684800713"/>
    <BOOK ISBN="0198711905"/>
  </BOOKS>;

var order:XML = <ORDER ITEMS=""/>;
order.@ITEMS = books.*.@ISBN;

// Выдает:
<ORDER ITEMS="0141182806 0684800713 0198711905"/>
```

Замена всего элемента

Чтобы заменить элемент XML новыми элементами, ему присваивается либо объект `XMLList`, либо объект `XML`. Например, в следующем коде элемент `<DIV>` заменяет элемент `<P>`:

```
var doc:XML = <DOC>
    <P ALIGN="CENTER">E4X is fun</P>
  </DOC>;
doc.P = <DIV>E4X is convenient</DIV>;

// Выдает:
<DOC>
  <DIV>E4X is convenient</DIV>
</DOC>
```

Содержимое элемента может также быть заменено с помощью метода экземпляра `replace( )` класса `XML`. Например:

```
// Тождественно: doc.P = <DIV>E4X is convenient</DIV>
doc.replace("P", <DIV>E4X is convenient</DIV>);
```

Обратите внимание, что, когда XML-элемент заменяется содержимым из другого документа, новое содержимое представляет собой копию содержимого элемента другого документа, а не ссылку на него. Например, рассмотрим два следующих фрагмента XML:

```
var user1:XML = <USERDETAILS>
    <LOGIN>joe</LOGIN>
    <PASSWORD>linuxRules</PASSWORD>
  </USERDETAILS>;

var user2:XML = <USERDETAILS>
    <LOGIN>ken</LOGIN>
    <PASSWORD>default</PASSWORD>
  </USERDETAILS>;
```

Мы можем заменить элемент `<PASSWORD>` фрагмента `user2` элементом `<PASSWORD>` фрагмента `user1` следующим образом:

```
user2.PASSWORD = user1.PASSWORD;
```

После замены два элемента <PASSWORD> будут иметь одинаковое содержимое:

```
trace(user1.PASSWORD[0] == user2.PASSWORD[0]); // Выводит: true
```

Но они ссылаются на разные экземпляры класса *XML*:

```
trace(user1.PASSWORD[0] === user2.PASSWORD[0]); // Выводит: false
```

Получить информацию о различиях между двумя предыдущими выражениями равенства можно далее, в разд. «Определение равенства в расширении E4X».

Добавление новых атрибутов и элементов

Мы можем добавлять новые атрибуты и элементы в документ, используя тот же синтаксис присваивания, который применяется для изменения и замены существующих атрибутов и элементов.

В расширении E4X, когда значение присваивается несуществующему атрибуту или элементу, среда выполнения Flash автоматически добавляет указанный атрибут или элемент в документ. В качестве примера создадим наш фрагмент <BOOK> с нуля. Мы начнем с пустого элемента <BOOK>:

```
var novel:XML = <BOOK/>;
```

Затем добавим атрибут ISBN:

```
novel.@ISBN = "0141182806";
```

Наконец, добавим элементы <TITLE>, <AUTHOR> и <PUBLISHER>:

```
novel.TITLE = "Ulysses";
novel.AUTHOR = "Joyce, James";
novel.PUBLISHER = "Penguin Books Ltd";
```

Для каждой операции присваивания к тегу <BOOK> в качестве нового последнего ребенка добавляется новый элемент. Таким образом, в результате выполнения предыдущего кода будет получен следующий XML-фрагмент:

```
<BOOK ISBN="0141182806">
  <TITLE>Ulysses</TITLE>
  <AUTHOR>Joyce, James</AUTHOR>
  <PUBLISHER>Penguin Books Ltd</PUBLISHER>
</BOOK>
```

Синтаксис присваивания также может быть использован для добавления вложенной структуры XML с помощью одного оператора. Предположим, что мы хотим добавить следующее вложенное содержимое, описывающее место действия романа, к элементу <BOOK>:

```
<SETTING>
  <CITY>Dublin</CITY>
  <COUNTRY>Ireland</COUNTRY>
</SETTING>
```

Для этого можно воспользоваться следующим кодом:

```
novel.SETTING.CITY = "Dublin";
novel.SETTING.COUNTRY = "Ireland";
```

Когда среда Flash пытается исполнить первую инструкцию, она определяет, что в документе нет ни элемента <SETTING>, ни элемента <CITY>, и, следовательно, создает оба этих элемента. Когда среда Flash исполняет вторую инструкцию, она видит, что элемент <SETTING> уже существует, и поэтому не создает его повторно. Вместо этого среда выполнения просто добавляет элемент <COUNTRY> к существующему элементу <SETTING>. Вот так выглядит получившийся фрагмент XML:

```
<BOOK ISBN="0141182806">
  <TITLE>Ulysses</TITLE>
  <AUTHOR>Joyce, James</AUTHOR>
  <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  <SETTING>
    <CITY>Dublin</CITY>
    <COUNTRY>Ireland</COUNTRY>
  </SETTING>
</BOOK>
```

Мы можем использовать аналогичный подход, чтобы представить информацию о месте действия в одном элементе, имеющем следующий формат:

```
<SETTING CITY="Dublin" COUNTRY="Ireland"/>
```

Для этого мы просто присваиваем желаемым атрибутам необходимые значения, как показано в следующем коде:

```
novel.SETTING.@CITY = "Dublin";
novel.SETTING.@COUNTRY = "Ireland";

//Выдает:
<BOOK ISBN="0141182806">
  <TITLE>Ulysses</TITLE>
  <AUTHOR>Joyce, James</AUTHOR>
  <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  <SETTING CITY="Dublin" COUNTRY="Ireland"/>
</BOOK>
```

В этом разделе мы узнали, что присваивание значения несуществующему элементу приводит к добавлению этого элемента в документ. Но что делать, если мы хотим добавить элемент с таким же именем, как имя существующего элемента? Например, как добавить несколько элементов <AUTHOR> к элементу <BOOK>? Ответы на этот вопрос вы найдете далее. По мере чтения следующих разделов обратите внимание на использование аддитивного оператора (+), который создает новый объект XMLList из набора экземпляров классов XML или XMLList. Аддитивный оператор принимает следующий вид:

ЭкземплярКлассаXMLИлиXMLList1 + ЭкземплярКлассаXMLИлиXMLList2

Он возвращает новый экземпляр класса XMLList, содержащий объединенный список всех экземпляров класса XML из экземпляров *ЭкземплярКлассаXMLИлиXMLList1* и *ЭкземплярКлассаXMLИлиXMLList2*.

Добавление нового ребенка за всеми существующими детьми

Чтобы добавить нового ребенка последним к существующему элементу, используйте один из следующих способов:

родитель.insertChildAfter(*родитель*.*[*родитель*.*.length( )-1], *<новыйРебенок/>*)

или:

родитель.*[*родитель*.*.length( )-1] = *родитель*.*[*родитель*.*.length( )-1] + *<новыйРебенок/>*

или:

родитель.appendChild(*<новыйРебенок/>*)

Например, следующий код добавляет новый элемент <DESCRIPTION> к <BOOK> сразу за существующим элементом <PUBLISHER>:

```
var novel:XML = <BOOK ISBN="0141182806">
  <TITLE>Ulysses</TITLE>
  <AUTHOR>Joyce, James</AUTHOR>
  <PUBLISHER>Penguin Books Ltd</PUBLISHER>
</BOOK>

novel.insertChildAfter(novel.*[novel.*.length( )-1],
                       <DESCRIPTION>A modern classic</DESCRIPTION>);
```

Приведенная строка является синонимом следующего кода, который заменяет последнего ребенка элемента <BOOK> объектом XMLList, содержащим последнего ребенка элемента <BOOK> (элемент <PUBLISHER>) и элемент <DESCRIPTION>:

```
novel.*[novel.*.length( )-1] = novel.*[novel.*.length( )-1]
                             + <DESCRIPTION>A modern classic</DESCRIPTION>;
```

Чтобы облегчить восприятие, рассмотрим данную строку кода еще раз, записав ее с помощью псевдокода:

```
// ПСЕВДОКОД:
<PUBLISHER> = <PUBLISHER> + <DESCRIPTION>A modern classic</DESCRIPTION>
```

То же самое мы можем записать, используя более сжатую форму (это настоящий код на языке ActionScript), как показано ниже:

```
novel.*[novel.*.length( )-1] += <DESCRIPTION>A modern classic</DESCRIPTION>;
```

А вот наиболее удобный подход:

```
novel.appendChild(<DESCRIPTION>A modern classic</DESCRIPTION>);
```

Добавление нового ребенка за указанным существующим ребенком

Чтобы добавить нового ребенка за указанным существующим ребенком, используйте один из следующих способов:

родитель.insertChildAfter(*родитель.существующийРебенок*[*n*], *<новыйРебенок/>*)

или:

родитель.существующийРебенок[*n*] = *родитель.существующийРебенок*[*n*] + *<новыйРебенок/>*

или:

родитель.*[*индексРебенка*] = *родитель*.*[*индексРебенка*] + *<новыйРебенок/>*

Например, следующий код добавляет второй элемент <AUTHOR> к элементу <BOOK> сразу за существующим элементом <AUTHOR>:

```
novel.insertChildAfter(novel.AUTHOR[0], <AUTHOR>Dave Luxton</AUTHOR>);
```

В качестве первого аргумента в метод `insertChildAfter( )` необходимо передавать экземпляр класса XML (а не класса XMLList!), поэтому мы должны использовать прямую ссылку на экземпляр класса XML — `novel.AUTHOR[0]`.

Чтобы вспомнить отличия между экземплярами классов XML и XMLList, обратитесь к подразд. «Интерпретация объекта XMLList как экземпляра класса XML» разд. «Обращение к данным XML».

В качестве альтернативы подходу с применением метода `insertChildAfter( )` можно использовать следующий код:

```
novel.AUTHOR[0] = novel.AUTHOR[0] + <AUTHOR>Dave Luxton</AUTHOR>;
```

Или более сжато:

```
novel.AUTHOR[0] += <AUTHOR>Dave Luxton</AUTHOR>;
```

А вот еще один аналогичный подход:

```
// Добавляем новый элемент XML за вторым ребенком элемента novel
novel.*[1] = novel.*[1] + <AUTHOR>Dave Luxton</AUTHOR>;
```

И вновь предыдущая строка может быть записана более сжато:

```
novel.*[1] += <AUTHOR>Dave Luxton</AUTHOR>;
```

Добавление нового ребенка перед указанным существующим ребенком

Чтобы добавить нового ребенка перед указанным существующим ребенком, используйте один из следующих способов:

родитель`.insertChildBefore(`*родитель.существующийРебенок*`[`*n*`], `*<новыйРебенок/>*`)`

или:

родитель.существующийРебенок`[`*n*`] = `*родитель.существующийРебенок*`[`*n*`]`
`+ `*<новыйРебенок/>*

или:

родитель`.*[`*индексРебенка*`] = `*родитель*`.*[`*индексРебенка*`] + `*<новыйРебенок/>*

Например, следующий код добавляет новый элемент <PRICE> к нашей книге непосредственно перед первым элементом <AUTHOR>:

```
novel.insertChildBefore(novel.AUTHOR[0], <PRICE>19.99</PRICE>);
```

Как и в случае с методом `insertChildAfter( )`, обратите внимание, что первым аргументом метода `insertChildBefore( )` должен быть экземпляр класса XML (а не класса XMLList!).

Предыдущая строка является синонимом следующей строки:

```
novel.AUTHOR = <PRICE>19.99</PRICE> + novel.AUTHOR;
```

Вот еще один аналогичный подход:

```
// Добавление нового элемента XML
// перед вторым ребенком элемента novel
novel.*[1] = <PRICE>19.99</PRICE> + novel.*[1];
```

Добавление нового ребенка перед всеми существующими детьми

Чтобы добавить новый элемент в качестве *первого* ребенка существующего элемента, используйте любой из следующих способов:

родитель.insertChildBefore(*родитель*.*[0], <*новыйРебенок*/>)

или:

родитель.*[0] = <*новыйРебенок*/> + *родитель*.*[0]

или:

родитель.prependChild(<*новыйРебенок*/>)

Например, следующий код добавляет новый элемент <PAGECOUNT> к нашей книге непосредственно перед существующим элементом <TITLE>:

```
novel.insertChildBefore(novel.*[0], <PAGECOUNT>1040</PAGECOUNT>);
```

Предыдущая строка аналогична следующей:

```
novel.*[0] = <PAGECOUNT>1040</PAGECOUNT> + novel.*[0];
```

А вот наиболее удобный подход:

```
novel.prependChild(<PAGECOUNT>1040</PAGECOUNT>);
```

Удаление элементов и атрибутов

Чтобы удалить элемент или атрибут из документа, используйте оператор `delete` следующим образом:

delete *элементИлиАтрибут*

Например, следующий код удаляет атрибут ISBN из элемента <BOOK>:

```
delete novel.@ISBN;
```

Следующий код удаляет элемент <TITLE> из элемента <BOOK>:

```
delete novel.TITLE;
```

Вот как можно удалить всех детей, содержащихся в элементе:

```
delete novel.*;  // Удаляет <TITLE>, <AUTHOR> и <PUBLISHER>
                 // из исходного XML-фрагмента.
```

Та же методика может применяться при удалении текстового содержимого из элемента:

```
delete novel.TITLE.*;  // Удаляет "Ulysses"
                       // из исходного XML-фрагмента.
```

Вот как можно удалить все атрибуты элемента:

```
delete novel.@*;  // Удаляет все атрибуты (в данном случае ISBN)
```

Ссылки на части документа не являются обновляемыми

Изменяя или добавляя новое содержимое в объект XML, имейте в виду, что любые вносимые изменения не отражаются на переменных, которые ссылаются на части этого документа. Например, следующий код создает переменную children, которая ссылается на узлы-детей элемента <BOOK>:

```
var novel:XML = <BOOK ISBN="0141182806">
  <TITLE>Ulysses</TITLE>
  <AUTHOR>Joyce, James</AUTHOR>
  <PUBLISHER>Penguin Books Ltd</PUBLISHER>
</BOOK>

var children:XMLList = novel.*;
```

Если теперь мы удалим элемент <PUBLISHER>, изменение будет внесено в исходный документ, но не затронет переменную children:

```
// Удаляем <PUBLISHER>
delete novel.PUBLISHER;
trace(novel);  // Выводит: <BOOK ISBN="0141182806">
               //             <TITLE>Ulysses</TITLE>
               //             <AUTHOR>Joyce, James</AUTHOR>
               //             </BOOK>

trace(children);  // Выводит: <TITLE>Ulysses</TITLE>
                  //             <AUTHOR>Joyce, James</AUTHOR>
                  //             <PUBLISHER>Penguin Books Ltd</PUBLISHER>
                  // <PUBLISHER> по-прежнему существует!
```

Возможно, будущие версии расширения E4X будут поддерживать обновляемые ссылки на части документа.

Использование сущностей XML для специальных символов

В расширении E4X реализованы особая интерпретация и правила для определенных символов пунктуации, когда они используются в литерале XML или в выражении присваивания значения XML-элементу. В табл. 18.2 показано, как включать эти символы в документ XML с помощью языка ActionScript. В левом столбце таблицы перечислены символы, а в остальных столбцах показан код, необходимый для включения этих символов в четырех различных контекстах. Для сравнения следующий код демонстрирует пример каждого из четырех типов контекста (контекст представлен текстом *некийТекст*).

```
// Текст литерала атрибута
var xml:XML = <некийЭлемент некийАтрибут="некийТекст"/>

// Текст, присваиваемый атрибуту
xml.@другойНекийАтрибут = "некийТекст"

// Текстовый узел в литерале элемента
var xml:XML = <некийЭлемент>некийТекст</некийЭлемент>
```

```
// Текстовый узел, присваиваемый элементу
xml.другойНекийЭлемент = "некийТекст";
```

Таблица 18.2. Присваивание специальных символов пунктуации

Символ	Текст литерала атрибута	Текст, присваиваемый атрибуту	Текстовый узел в литерале элемента	Текстовый узел, присваиваемый элементу
\	\\	\\	****	\\
&	&	&	&	&
"	"	\" или "	"	\"
'	'**	'	'	'
<	<	<	<	<
>	>	>	>	>
Новая строка (\n)	Не поддерживается*	Не поддерживается*	Не поддерживается*, ***	\n
{	{	{	&#x7B	{
}	}	}	&#x7D	}

* В данных контекстах последовательность новой строки `\n` автоматически преобразуется в сущность `
`.

** Чтобы включить символ ' в значение атрибута, отделенного символами ', используйте управляющую последовательность `'`.

*** Допускается применять последовательность `\n`, если значение элемента является вычисляемым. Например,

```
var val:String = "Newlines \n are \n okay \n here!";
var paragraph:XML = <p>{val}</p>;
```

**** В отличие от строк, в литерале XML символ обратного слэша (`\`) никогда не интерпретируется как начало управляющей последовательности.

Стоит отметить, что, хотя символы > и & могут быть использованы в любом литерале XML, когда среда Flash встречает их в текстовом узле при парсинге XML-документа, она автоматически преобразует эти символы в сущности `>` и `&` соответственно. Подобным образом, когда среда выполнения встречает символ & в значении атрибута при парсинге документа XML, она автоматически преобразует этот символ в сущность `&`. Однако при использовании в контексте строки эти сущности будут преобразованы обратно в исходные символы. Чтобы увидеть текстовый узел без преобразования его сущностей, используйте метод экземпляра `toXMLString( )` класса `XML`. Это демонстрирует следующий код:

```
var p:XML = <p>&></p>;
trace(p.toString( ));      // Выводит: &>
trace(p.toXMLString( ));  // Выводит: <p>&amp;gt;</p>
```

Наконец, обратите внимание, что, хотя символ ' может использоваться для отделения значения атрибута в литерале XML, в процессе парсинга он преобразуется в символ ". Это демонстрирует следующий код:

```
var p:XML = <p align='left'/>;
trace(p.toXMLString( ));  // Выводит: <p align="left"/>
```

Присваивание значений объекту XMLList

Как уже говорилось в подразд. «Изменение содержимого элемента», не существует разницы между присваиванием значения объекту XMLList, содержащему один-единственный экземпляр класса XML, и присваиванием значения непосредственно этому экземпляру класса XML. Тем не менее присваивание значения объекту XMLList, содержащему более одного экземпляра класса XML, может иметь множество различных результатов. В зависимости от типа присваиваемого значения и типа экземпляров класса XML, находящихся в списке, список может быть изменен или даже полностью заменен.

Единственный типичный сценарий использования операции присваивания значения экземпляру класса XMLList — замена ребенка элемента-родителя новым XML-элементом или списком элементов. Например, следующий код заменяет два ребенка <P> элемента <DOC> одним элементом <P>:

```
var doc:XML = <DOC TOPIC="Code Tips" AUTHOR="Colin">
    <P>Errors are your friends</P>
    <P>Backup often</P>
  </DOC>;

doc.* = <P>Practice coding everyday</P>;

// Выдает:
<DOC TOPIC="Code Tips" AUTHOR="Colin">
  <P>Practice coding everyday</P>
</DOC>
```

Присваивание значения объекту XMLList является редко используемой операцией, поэтому ее тщательное рассмотрение выходит за рамки этой книги. Читателям, интересующимся гротескными ситуациями в программировании, которой, например, является попытка присвоить список инструкций обработки списку атрибутов, предоставляется возможность самостоятельно познакомиться с ними.

Загрузка XML-данных

Для наглядности в большинстве примеров этой главы XML-данные были записаны в виде литералов. Однако в реальных приложениях они зачастую загружаются из внешнего источника.

Для загрузки XML-данных из внешнего источника в экземпляр класса XML используйте такую последовательность действий.

1. Создайте объект URLRequest, описав местоположение внешних данных XML (это может быть либо файл, либо серверный сценарий, возвращающий данные в формате XML).
2. Создайте объект URLLoader и используйте его метод load( ) для загрузки данных XML.
3. Подождите, пока загрузятся данные XML.

4. Передайте загруженные данные XML в конструктор нового экземпляра класса XML.

Хотя подробное рассмотрение классов `URLRequest` и `URLLoader` выходит за рамки этой главы, в листинге 18.13 представлен код, необходимый для загрузки XML-данных в экземпляр класса `XML`. Используемый в примере класс `XMLLoader` расширяет класс `Sprite`, чтобы его можно было откомпилировать в качестве основного класса приложения для тестирования. Информацию о классах `URLRequest` и `URLLoader` можно найти в справочнике по языку ActionScript корпорации Adobe. Информацию об обработке событий можно получить в гл. 12.

Листинг 18.13. Загрузка данных XML из внешнего источника

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.net.*;

  // Демонстрирует код, необходимый для загрузки XML-данных
  // из внешнего источника
  public class XMLLoader extends Sprite {
    // Переменная, которой будут присвоены загруженные XML-данные
    private var novel:XML;
    // Объект, используемый для загрузки XML-данных
    private var urlLoader:URLLoader;

    // Конструктор
    public function XMLLoader ( ) {
      // Указываем местоположение внешнего источника XML-данных
      var urlRequest:URLRequest = new URLRequest("novel.xml");
      // Создаем объект, который умеет загружать внешние текстовые данные
      urlLoader = new URLLoader( );
      // Регистрируем обработчик, чтобы получить событие об окончании
      // загрузки XML-данных
      urlLoader.addEventListener(Event.COMPLETE, completeListener);
      // Загружаем XML-данные
      urlLoader.load(urlRequest);
    }

    // Метод, вызываемый автоматически по окончании загрузки XML-данных
    private function completeListener(e:Event):void {
      // Строка, содержащая загруженные XML-данные, присваивается переменной
      // data объекта URLLoader (то есть urlLoader.data). Чтобы создать
      // новый экземпляр класса XML из этой загруженной строки, мы передаем
      // ее в конструктор класса XML
      novel = new XML(urlLoader.data);
      trace(novel.toXMLString( )); // Отображаем загруженные XML-данные,
                                   // преобразованные в объект XML
    }
  }
```

Обратите внимание, что все операции загрузки данных в языке ActionScript, включая операцию, представленную в листинге 18.13, попадают под действие ограничений безопасности приложения Flash Player. Полную информацию об ограничениях безопасности можно найти в гл. 19.

Использование пространств имен XML

В языке XML пространства имен используются для предотвращения конфликтов имен в разметке, а конечной целью их использования является возможность мирного сосуществования разметок из различных XML-словарей в одном документе.

В языке ActionScript пространства имен поддерживаются и как часть расширения E4X, и как базовый инструмент программирования. В этом разделе описываются приемы работы с пространствами имен при помощи синтаксиса расширения E4X, но предполагается, что вы знакомы с базовыми концепциями, изложенными в определении пространств имен консорциума W3C для языка XML. Данная информация изложена на следующих интернет-страницах:

- статья Рональда Бурета (Ronald Bouret) «XML Namespaces FAQ»: http://www.rpbourret.com/xml/NamespacesFAQ.htm;
- рекомендации консорциума W3C «Namespaces in XML 1.1»: http://www.w3.org/TR/xml-names11/;
- статья Дэвида Мартсона (David Martson) «Plan to use XML namespaces, Part 1»: http://www-128.ibm.com/developerworks/library/x-nmspace.html.

Информацию об использовании не относящихся к языку XML пространств имен в программировании на языке ActionScript можно найти в гл. 17.

Обращение к элементам и атрибутам, уточненным пространствами имен

Ранее рассказывалось, как обращаться к элементам и атрибутам, которые не уточняются пространством имен. Чтобы познакомиться с дополнительными способами, необходимыми для обращения к элементам и атрибутам, уточненным пространством имен, рассмотрим новый пример XML-фрагмента, приведенный в листинге 18.14. Фрагмент представляет собой часть гипотетического каталога мебели. Просматривая пример, обратите внимание на следующие детали, относящиеся к пространствам имен:

- идентификатор URI `http://www.example.com/furniture` и сопутствующий префикс `shop`;
- пространство имен по умолчанию `http://www.w3.org/1999/xhtml`;
- три элемента, которые квалифицированы пространством имен `http://www.example.com/furniture`: `<shop:table>`, `<shop:desc>` и `<shop:price>`;
- один атрибут, уточненный пространством имен `http://www.example.com/furniture`: `shop:id`.

Листинг 18.14. Использование пространств имен в каталоге мебели

```
var catalog:XML = <html xmlns:shop="http://www.example.com/furniture"
      xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Catalog</title>
  </head>
  <body>
    <shop:table shop:id="4875">
      <table border="1">
        <tr align="center">
          <td>Item</td>
          <td>Price</td>
        </tr>
        <tr align="left">
          <td><shop:desc>3-legged Coffee Table</shop:desc></td>
          <td><shop:price>79.99</shop:price></td>
        </tr>
      </table>
    </shop:table>
  </body>
</html>
```

В листинге 18.14 в основном представлен XHTML-документ, предназначенный для отображения в браузерах, но кроме этого он содержит разметку, представляющую предметы из каталога мебели. Разметка каталога мебели придает документу семантическую структуру, позволяя обрабатывать этот документ не только браузерами, но и другими клиентами. В каталоге используется пространство имен, чтобы устранить неоднозначность между разметкой XHTML и разметкой каталога мебели. Как результат, элемент `<table>` может представлять, с одной стороны, предмет мебели (стол), а с другой — графический элемент макета веб-страницы. При этом конфликты имен отсутствуют.

Для обращения к уточненным пространствам имен элементам и атрибутам из листинга 18.14 мы должны сначала получить ссылку на используемые пространства имен. Чтобы получить ссылку на пространство `http://www.example.com/furniture`, мы вызываем метод экземпляра `namespace( )` класса `XML` над корневым узлом документа, передавая в качестве аргумента префикс `"shop"`. В результате метод `namespace( )` возвращает объект `Namespace`, представляющий пространство имен `http://www.example.com/furniture`. Мы присваиваем этот объект переменной `shopNS` для дальнейшего использования.

```
var shopNS:Namespace = catalog.namespace("shop");
```

В качестве альтернативы, если известен идентификатор URI пространства имен, можно создать ссылку на объект `Namespace`, используя конструктор одноименного класса:

```
var shopNS:Namespace = new Namespace("http://www.example.com/furniture");
```

Чтобы получить ссылку на пространство имен, используемое по умолчанию, мы вызываем метод `namespace( )` над корневым узлом документа, не передавая префикс пространства имен:

```
var htmlNS:Namespace = catalog.namespace( );
```

В качестве альтернативы, если известен идентификатор URI пространства имен, мы можем создать ссылку на объект `Namespace`, используемый по умолчанию, с помощью конструктора одноименного класса:

```
var htmlNS:Namespace = new Namespace("http://www.w3.org/1999/xhtml");
```

Для обращения к пространствам имен документа могут использоваться и методы `inScopeNamespaces( )` и `namespaceDeclarations( )`. Дополнительную информацию можно найти в справочнике по языку ActionScript корпорации Adobe.

В расширении E4X атрибуты пространств имен XML не представляются в виде атрибутов (то есть к ним нельзя обратиться с помощью метода attributes( ) или выражения некийЭлемент.@*). Вместо этого для обращения к объявлениям пространств имен элемента применяется метод экземпляра namespaceDeclarations( ) класса XML.

Теперь, когда у нас есть ссылка на объект `Namespace`, мы можем обращаться к уточненным пространством имен элементам и атрибутам, используя уточненные имена в следующем общем формате:

пространствоИмен::*локальноеИмяЭлемента*
пространствоИмен::@*локальноеИмяАтрибута*

Например, вот так записывается уточненное имя элемента `<shop:price>` в коде на языке ActionScript:

```
shopNS::price
```

Обратите внимание на использование оператора уточнителя имени (`::`), который отделяет имя пространства имен от локального имени.

Вот так можно обратиться к элементу `<body>`, который уточняется пространством имен по умолчанию (`http://www.w3.org/1999/xhtml`):

```
catalog.htmlNS::body
```

Для обращения к элементу `<shop:table>`, который является ребенком элемента `<body>`, используется следующая запись:

```
catalog.htmlNS::body.shopNS::table
```

Для обращения к атрибуту `shop:id` используется такая запись:

```
catalog.htmlNS::body.shopNS::table.@shopNS::id
```

Для обращения к элементу `<shop:price>` мы можем использовать следующий «кошмарный» код:

```
catalog.htmlNS::body.shopNS::table.htmlNS::table.htmlNS::tr[
    1].htmlNS::td[1].shopNS::price
```

Однако мы будем спать спокойнее, если воспользуемся преимуществом оператора «потомок» (`..`) в двух местах, как показано в следующем коде:

```
catalog..shopNS::table..shopNS::price
```

И все равно повторное использование `shopNS::` слегка раздражает. Мы можем избежать нескольких лишних нажатий клавиш, попросив язык ActionScript автоматически уточнять все неуточненные имена элементов и атрибутов с помощью

выбранного пространства имен. Для этого мы используем инструкцию назначения пространства имен XML по умолчанию, которая имеет следующий вид:

```
default xml namespace = пространствоИменИлиСтрокаURI
```

Например, следующий код заставляет язык ActionScript автоматически уточнять все неуточненные имена элементов и атрибутов с помощью пространства имен `http://www.example.com/furniture`:

```
default xml namespace = shopNS;
```

После выполнения этой инструкции пространство имен `http://www.example.com/furniture` применяется ко всем неуточненным ссылкам на элементы и атрибуты, поэтому мы можем сократить следующий код:

```
catalog..shopNS::table..shopNS::price
```

до записи:

```
catalog..table..price
```

Из-за ошибки в приложении Flash Player 9 при выполнении предыдущего примера кода (catalog..table..price) в первый раз возвращается значение undefined.

В более полном примере документ каталога, скорее всего, будет содержать несколько элементов `<shop:table>`. Для обращения к определенному столу нам придется использовать фильтрующий предикат, как показано в следующем коде:

```
catalog..table.(@id == 4875)..price
```

В листинге 18.15 приведен код, который можно использовать для обращения и отображения информации обо всех столах из каталога.

Листинг 18.15. Отображение всех столов из каталога

```
var shopNS:Namespace = catalog.namespace("shop");
default xml namespace = shopNS;
for each (var table:XML in catalog..table) {
  trace(table..desc + ": " + table..price);
}
```

Как и в случае с именами элементов и атрибутов, мы можем использовать групповой символ свойств (`*`) с пространствами имен. Например, следующий код возвращает объект `XMLList`, представляющий все элементы `<table>` во всех пространствах имен:

```
catalog..*::table
```

Чтобы получить всех потомков на всех уровнях иерархии во всех пространствах имен или за пределами пространства имен, применяется следующий код:

```
объектXML..*::*   // элементы
объектXML..@*::*  // атрибуты
```

Чтобы получить всех детей во всех пространствах имен или за пределами пространства имен, используется такой код:

```
объектXML.*::*   // элементы
объектXML.@*::*  // атрибуты
```

Создание элементов и атрибутов, уточняемых пространствами имен

Для создания элементов и атрибутов, которые уточняются пространствами имен, мы используем синтаксис обращения к уточненным именам, рассмотренный в предыдущем разделе, вместе с методиками создания элементов и атрибутов, описанными в разд. «Изменение или создание нового содержимого XML».

Перед созданием имен, уточняемых пространствами имен, мы должны создать (или получить) ссылку на объект `Namespace`. Например, следующий код создает два объекта `Namespace` и присваивает их переменным `htmlNS` и `shopNS` для дальнейшего использования в уточненных именах:

```
var htmlNS:Namespace = new Namespace("html",
                                     "http://www.w3.org/1999/xhtml");
var shopNS:Namespace = new Namespace("shop",
                                     "http://www.example.com/furniture");
```

Когда создается не один элемент или атрибут, а целый документ, обычно (и это более удобно) применяется пространство имен по умолчанию, которое указывается в инструкции `default XML namespace`. Например, следующий код в качестве пространства имен по умолчанию устанавливает `http://www.w3.org/1999/xhtml`:

```
default xml namespace = htmlNS;
```

Как только будет установлено пространство имен по умолчанию, все вновь создаваемые элементы (но не атрибуты), для которых пространство имен не указано явным образом, будут автоматически уточняться с помощью пространства имен по умолчанию. Например, следующий код создает элемент с локальным именем `"html"`; для него явно не указано пространство имен, поэтому это имя автоматически уточняется с помощью пространства имен по умолчанию (`http://www.w3.org/1999/xhtml`):

```
var catalog:XML = <html/>;
```

Данная строка кода генерирует следующий исходный код XML:

```
<html xmlns="http://www.w3.org/1999/xhtml" />
```

Чтобы добавить объявление пространства имен к указанному элементу, используется метод экземпляра `addNamespace( )` класса `XML`. Например, следующий код добавляет объявление нового пространства имен к предыдущему элементу:

```
catalog.addNamespace(shopNS);
```

Результирующий исходный код XML выглядит так:

```
<html xmlns:shop="http://www.example.com/furniture"
      xmlns="http://www.w3.org/1999/xhtml" />
```

Возможно, вы узнали приведенный элемент — это первая строка кода из документа каталога, представленного в листинге 18.14. Достроим оставшуюся часть этого документа. Вот теги `<head>` и `<title>`. Их имена автоматически уточняются пространством имен по умолчанию (`http://www.w3.org/1999/xhtml`).

```
catalog.head.title = "Catalog";
```

Теперь создадим элемент <shop:table> и его атрибут shop:id. Их имена уточняются пространством имен http://www.example.com/furniture. Мы хотим получить исходный код XML, который выглядит следующим образом:

```
<shop:table shop:id="4875">
```

Для того чтобы создать данный элемент, воспользуемся следующим кодом на языке ActionScript:

```
catalog.body.shopNS::table = "";
catalog.body.shopNS::table.@shopNS::id = "4875";
```

Приведенный код вам должен быть знаком. За исключением синтаксиса уточнителя пространства имен shopNS::, он аналогичен коду, который мы использовали ранее для создания элементов и атрибутов. Уточнитель пространства имен просто задает пространство для локальных имен table и id. В листинге 18.16 применяется та же методика для создания оставшейся части документа каталога. В примере обратите внимание на следующую строку кода:

```
catalog.body.shopNS::table.table.tr.td[1] = "Price";
```

Она создает новый элемент с именем "td" непосредственно за существующим элементом catalog.body.shopNS::table.table.tr.td[0].

Листинг 18.16. Создание каталога мебели

```
// Создание пространств имен
var htmlNS:Namespace = new Namespace("html",
                                     "http://www.w3.org/1999/xhtml");
var shopNS:Namespace = new Namespace("shop",
                                     "http://www.example.com/furniture");
// Задание пространства имен по умолчанию
default xml namespace = htmlNS;

// Создание корневого элемента
var catalog:XML = <html/>;

// Добавление пространства имен furniture к корневому элементу
catalog.addNamespace(shopNS);

// Создание оставшейся части документа
catalog.head.title = "Catalog";
catalog.body.shopNS::table = "";
catalog.body.shopNS::table.@shopNS::id = "4875";
catalog.body.shopNS::table.table = "";
catalog.body.shopNS::table.table.@border = "1";
catalog.body.shopNS::table.table.tr.td = "Item";
catalog.body.shopNS::table.table.tr.td[1] = "Price";
catalog.body.shopNS::table.table.tr.@align = "center";
catalog.body.shopNS::table.table.tr[1] = "";
catalog.body.shopNS::table.table.tr[1].@align = "left";
catalog.body.shopNS::table.table.tr[1].td.shopNS::desc =
                                                  "3-legged Coffee Table";
catalog.body.shopNS::table.table.tr[1].td[1] = "";
catalog.body.shopNS::table.table.tr[1].td[1].shopNS::price = "79.99";
```

Вот мы и рассмотрели все важнейшие темы, касающиеся расширения E4X. В оставшейся части этой главы будут рассмотрены два дополнительных вопроса: преобразование и равенство данных XML.

Преобразование объектов XML и XMLList в строки

Как мы уже видели на протяжении этой главы, расширение E4X реализует собственные правила для преобразования экземпляров классов `XML` и `XMLList` в строку. Для информации в этом разделе описываются правила расширения E4X, применяемые для преобразования объектов `XML` и `XMLList` в строку. Не забывайте, что экземпляр класса `XML` может представлять пять различных типов содержимого: элемент, атрибут, текстовый узел, комментарий или инструкцию обработки. Мы рассмотрим правила преобразования для каждого типа в отдельности, но начнем с рассмотрения преобразования объекта `XMLList` в строку, чтобы подготовиться к дальнейшему обсуждению.

Преобразование объекта XMLList в строку

Когда объект `XMLList` содержит один-единственный экземпляр класса `XML`, результат вызова метода `toString( )` класса `XMLList` полностью совпадает с результатом вызова метода `toString( )` над этим единственным экземпляром класса `XML`. Например, в следующем коде переменная `title` ссылается на объект `XMLList`, чей единственный экземпляр класса `XML` представляет элемент `<TITLE>`. В результате преобразования значения переменной `title` в строку возвращается значение `Ulysses` — именно оно было бы получено в результате вызова метода `toString( )` непосредственно над единственным экземпляром класса `XML`:

```
var novel:XML = <BOOK ISBN="0141182806">
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  </BOOK>

// Создаем объект XMLList с одним-единственным экземпляром класса XML
var title:XMLList = novel.TITLE;
// Преобразуем объект XMLList в строку и отображаем ее.
trace(title);  // Отображает: Ulysses
```

Когда объект `XMLList` содержит несколько экземпляров класса `XML`, вызов метода `toString( )` класса `XMLList` возвращает строку, которая получается в результате вызова метода `toXMLString( )` над каждым экземпляром класса `XML` и объединения содержимого этих экземпляров, при этом содержимое каждого экземпляра размещается на отдельной строке. Например, в следующем коде объект `XMLList`, присваиваемый переменной `details`, содержит три экземпляра класса `XML`, которые представляют три элемента — `<TITLE>`, `<AUTHOR>` и `<PUBLISHER>`:

```
// Создаем объект XMLList с тремя экземплярами класса XML
var details:XMLList = novel.*;
```

Преобразование значения переменной `details` в строку возвращает исходный код XML для элементов <TITLE>, <AUTHOR> и <PUBLISHER>:

```
// Преобразуем объект XMLList в строку и отображаем ее
trace(details);  // Выводит:
                 // <TITLE>Ulysses</TITLE>
                 // <AUTHOR>Joyce, James</AUTHOR>
                 // <PUBLISHER>Penguin Books Ltd</PUBLISHER>
```

Преобразование элемента XML в строку

Для экземпляров класса XML, представляющих элементы, метод `toString( )` этого класса возвращает один из двух результатов в зависимости от содержимого данного элемента. Если элемент содержит детей, метод `toString( )` класса XML возвращает исходный XML-код для этого элемента и его детей, отформатированный в соответствии с настройками переменных `XML.ignoreWhitespace`, `XML.prettyPrinting` и `XML.prettyIndent`. Например, в следующем коде элемент <BOOK> имеет три элемента-ребенка (<TITLE>, <AUTHOR> и <PUBLISHER>):

```
var novel:XML = <BOOK ISBN="0141182806">
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  </BOOK>;
```

Преобразование этого элемента в строку вернет исходный XML-код:

```
trace(novel.toString( ));  // Отображает:
                           // <BOOK ISBN="0141182806">
                           //   <TITLE>Ulysses</TITLE>
                           //   <AUTHOR>Joyce, James</AUTHOR>
                           //   <PUBLISHER>Penguin Books Ltd</PUBLISHER>
                           // </BOOK>;
```

Для элемента, не имеющего детей, метод `toString( )` класса XML вернет текст, содержащийся в этом элементе, без открывающего и закрывающего тегов. Например, следующий код преобразует элемент <TITLE> в строку. Результатом является строка `Ulysses`, а не <TITLE>Ulysses</TITLE>.

```
trace(novel.TITLE.toString( ));  // Выводит: Ulysses
```

Если мы хотим получить строку, включающую текстовый узел и окружающие его теги, используется метод `toXMLString( )` класса XML, как показано в следующем коде:

```
trace(novel.TITLE.toXMLString( ));  // Выводит: <TITLE>Ulysses</TITLE>
```

Стоит отметить, что правила преобразования строк для элементов XML, определенные в расширении E4X, изменяют способ обращения к концевым текстовым узлам в языке ActionScript. В языках ActionScript версий 1.0 и 2.0 для обращения к текстовым узлам использовалась переменная экземпляра `firstChild` класса XML (которая теперь, в языке ActionScript 3.0, является переменной экземпляра `firstChild` класса `XMLDocument`). Например, устаревшим эквивалентом следующей инструкции расширения E4X:

```
trace(novel.TITLE.toString( ));
```

будет являться:

```
trace(novel.firstChild.firstChild.firstChild);
```

В расширении E4X к тексту элемента, не имеющего элементов-детей, в контексте строки можно обратиться непосредственно через имя этого элемента. Вот еще два примера кода с использованием расширения E4X (на этот раз мы опустили явный вызов метода `toString( )`, поскольку среда Flash автоматически вызывает этот метод над любым аргументом, передаваемым в функцию `trace( )`):

```
trace(novel.AUTHOR);     // Выводит: Joyce, James
trace(novel.PUBLISHER);  // Выводит: Penguin Books Ltd
```

Далее представлено прямое сравнение устаревшего метода доступа к текстовым узлам с аналогичным методом доступа расширения E4X:

```
// Доступ к текстовому узлу в расширении E4X
var msg:XML = <GREETING>
    <TO>World</TO>
    <FROM>J. Programmer</FROM>
    <MESSAGE>Hello</MESSAGE>
  </GREETING>
trace(msg.TO);       // Выводит: World
trace(msg.FROM);     // Выводит: J. Programmer
trace(msg.MESSAGE);  // Выводит: Hello

// Устаревший метод доступа к текстовому узлу
var msgDoc:XMLDocument = new XMLDocument("<GREETING>"
  + "<TO>World</TO>"
  + "<FROM>J. Programmer</FROM>"
  + "<MESSAGE>Hello</MESSAGE>"
  + "</GREETING>");
trace(msgDoc.firstChild.firstChild.firstChild);    // Выводит: World
trace(msgDoc.firstChild.childNodes[1].firstChild); // Выводит: J. Programmer
trace(msgDoc.firstChild.childNodes[2].firstChild); // Выводит: Hello
```

Преобразование атрибута в строку

Для экземпляров класса XML, которые представляют атрибуты, метод `toString( )` возвращает только значение этого атрибута, а не все его определение целиком. Например, следующий код преобразует атрибут `ISBN` предыдущего элемента <BOOK> в строку. Результатом преобразования является строка `0141182806`, а не `ISBN='0141182806'`.

```
trace(novel.@ISBN.toString( ));  // Выводит: 0141182806
```

Преобразование комментариев и инструкций обработки в строки

Когда метод `toString( )` класса XML вызывается над экземпляром класса XML, представляющим комментарий или инструкцию обработки, этот комментарий или инструкция обработки возвращается целиком:

```
XML.ignoreComments = false;
XML.ignoreProcessingInstructions = false;

// Создаем фрагмент XML, включающий и комментарий, и инструкцию обработки
// (выделены полужирным шрифтом)
var novel:XML = <BOOK ISBN="0141182806">
    <!--Это комментарий-->
    <?someTargetApp someData?>
    <TITLE>Ulysses</TITLE>
    <AUTHOR>Joyce, James</AUTHOR>
    <PUBLISHER>Penguin Books Ltd</PUBLISHER>
  </BOOK>;

// Преобразуем комментарий в строку.
// Выводит: <!-Это комментарий-->
trace(novel.comments( )[0].toString( ));

// Преобразуем инструкцию обработки в строку.
// Выводит: <?someTargetApp someData?>
trace(novel.processingInstructions( )[0].toString( ));
```

Определение равенства в расширении E4X

В этом разделе рассматриваются специальные правила языка ActionScript для определения равенства объектов `XML`, `XMLList`, `QName` и `Namespace`. Стоит отметить, однако, что представленная информация применима только к оператору равенства (==) — она не касается оператора строгого равенства (===). Расширение E4X не изменяет семантику оператора строгого равенства. В частности, этот оператор считает два экземпляра класса `XML`, `XMLList` — `QName` или `Namespace` — равными тогда, и только тогда, когда они указывают на одну и ту же объектную ссылку.

Равенство объектов XML

Оператор равенства (==) считает два экземпляра класса `XML`, представляющие элементы, равными, если они представляют одну и ту же иерархию XML. Например, в следующем коде переменные `x1` и `x2` указывают на различные объектные ссылки, но считаются равными, поскольку представляют одну и ту же иерархию XML:

```
var x1:XML = <a><b></b></a>;
var x2:XML = <a><b></b></a>;
trace(x1 == x2);  // Выводит: true
```

По умолчанию расширение E4X игнорирует пробельные узлы, поэтому два экземпляра класса `XML`, представляющие элементы, считаются равными, когда имеют одинаковую разметку, даже если используется различное форматирование. Например, в следующем коде исходный XML-код для экземпляра класса `XML`, хранящегося в переменной `x1`, не содержит пробельные узлы, в отличие от исходного XML-кода для экземпляра класса `XML`, хранящегося в переменной `x2`, который содержит два пробельных узла. Однако, несмотря на это различие, экземпляры по-прежнему

считаются равными, поскольку пробельные символы игнорируются — это значит, что иерархии XML остаются одинаковыми.

```
var x1:XML = <a><b></b></a>;
var x2:XML = <a>
               <b></b>
             </a>;
trace(x1 == x2);  // По-прежнему выводит: true
```

Если перед парсингом кода XML мы скажем среде Flash учитывать пробельные узлы, экземпляры класса XML из предыдущего примера больше не будут считаться равными, как показано в следующем коде:

```
XML.ignoreWhitespace = false;  // Не игнорировать пробельные узлы
var x1:XML = <a><b></b></a>;
var x2:XML = <a>
               <b></b>
             </a>;
trace(x1 == x2);  // Теперь выводит: false
```

Экземпляр класса XML, представляющий элемент, считается равным экземпляру класса XML, представляющему атрибут, если элемент не имеет элементов-детей и содержащийся в нем текст совпадает со значением атрибута. Например, в следующем коде атрибут QUANTITY считается равным элементу <COST>, поскольку элемент <COST> не имеет элементов-детей и содержит текст, который совпадает со значением атрибута QUANTITY:

```
var product:XML = <PRODUCT QUANTITY="1"><COST>1</COST></PRODUCT>;
trace(product.@QUANTITY == product.COST);  // Выводит: true
```

Подобным образом экземпляр класса XML, представляющий элемент, считается равным экземпляру класса XML, представляющему текстовый узел, если элемент не имеет элементов-детей и содержащийся в нем текст совпадает со значением текстового узла. Например, в следующем коде текстовый узел, содержащийся в элементе <COST>, считается равным элементу <QUANTITY>, поскольку второй элемент не имеет элементов-детей и содержит текст, который совпадает со значением текстового узла-ребенка первого элемента:

```
var product:XML = <PRODUCT>
                    <COST>1</COST>
                    <QUANTITY>1</QUANTITY>
                  </PRODUCT>;
trace(product.COST.*[0] == product.QUANTITY);  // Выводит: true
```

Во всех остальных случаях, если типы узлов двух экземпляров класса XML отличаются, эти два экземпляра считаются разными. Когда типы узлов совпадают, два экземпляра считаются равными, если типом обоих узлов является:

- «атрибут», а значения атрибутов совпадают;
- «текст», а текст узлов совпадает;
- «комментарий», а текст, расположенный между открывающими и закрывающими разделителями комментария (<!-- и -->), совпадает;
- «инструкция обработки», а текст, расположенный между открывающими и закрывающими разделителями инструкции обработки (<? и ?>), совпадает.

Равенство объектов XMLList

Чтобы определить, являются ли два экземпляра класса `XMLList` равными, среда выполнения Flash сравнивает по порядку все экземпляры в этих объектах, используя правила определения равенства объектов `XML`, которые были рассмотрены в предыдущем разделе. Если хотя бы один элемент из первого экземпляра класса `XMLList` не равен соответствующему элементу из второго экземпляра, два экземпляра класса `XMLList` считаются неравными. Например, в следующем коде объект `XMLList`, возвращаемый выражением `msg1.*`, считается равным объекту `XMLList`, возвращаемому выражением `msg2.*`, поскольку каждый экземпляр класса `XML` из результата выражения `msg1.*` считается равным экземпляру класса `XML` в соответствующей позиции из результата выражения `msg2.*`:

```
var msg1:XML = <GREETING>
    <TO>World</TO>
    <FROM>J. Programmer</FROM>
    <MESSAGE>Hello</MESSAGE>
  </GREETING>;

var msg2:XML = <GREETING>
    <TO>World</TO>
    <FROM>J. Programmer</FROM>
    <MESSAGE>Hello</MESSAGE>
  </GREETING>;

trace(msg1.* == msg2.*);  // Выводит: true
```

Экземпляр класса `XML` и объект `XMLList`, содержащий один-единственный экземпляр класса `XML`, сравниваются точно так же, как два экземпляра класса `XML`:

```
trace(msg1.FROM == msg2.*[1]);  // Выводит: true
```

Это значит, что оператор равенства (==) считает объект `XMLList` с одним-единственным экземпляром класса `XML` равным этому экземпляру!

```
trace(msg1.FROM == msg1.FROM[0]);  // Выводит: true
```

Чтобы отличить объект `XMLList`, содержащий один-единственный экземпляр класса `XML`, от этого экземпляра, используйте оператор строгого равенства (===):

```
trace(msg1.FROM === msg1.FROM[0]);  // Выводит: false
```

Равенство объектов QName

Класс `QName` представляет имя элемента или атрибута, уточняемое пространством имен. Два экземпляра класса `QName` считаются равными, если их названия пространств имен и локальные имена совпадают (то есть они имеют одинаковые значения переменных `uri` и `localName`). Например, следующий код создает объект `QName` с помощью конструктора класса `QName` и сравнивает этот объект с объектом `QName`, полученным из XML-документа. Оба объекта `QName` имеют одинаковые названия пространств имен и локальные имена, поэтому считаются равными.

```
var product:XML = <someCorp:PRODUCT
                    xmlns:someCorp="http://www.example.com/someCorp">
                      <someCorp:PRICE>99.99</someCorp:PRICE>
                  </someCorp:PRODUCT>;

var someCorp:Namespace = product.namespace("someCorp");
var qn1:QName = new QName("http://www.example.com/someCorp", "PRICE");
var qn2:QName = product.someCorp::PRICE.name( );

trace(qn1 == qn2);  // Выводит: true
```

Равенство объектов Namespace

Класс `Namespace` представляет уточняющую часть имени, которое уточняется с помощью пространства имен. Два объекта `Namespace` считаются равными тогда, и только тогда, когда они содержат одно и то же название пространства имен (то есть когда их переменные `uri` хранят одно и то же значение), независимо от префикса. Например, следующий код создает объект `Namespace` с помощью конструктора класса `Namespace` и сравнивает созданный объект с объектом `Namespace`, полученным из XML-документа. Оба объекта `Namespace` содержат один и тот же идентификатор URI, поэтому считаются равными, несмотря на то, что префиксы отличаются.

```
var product:XML = <someCorp:PRODUCT
                     xmlns:someCorp="http://www.example.com/someCorp">
                     <someCorp:PRICE>99.99</someCorp:PRICE>
                  </someCorp:PRODUCT>;
var ns1:Namespace = product.namespace("someCorp");
var ns2:Namespace = new Namespace("sc", "http://www.example.com/someCorp");
trace(ns1 == ns2);  // Выводит: true
```

Дальнейшее изучение

В этой главе была рассмотрена большая часть основной функциональности расширения E4X, однако исчерпывающее описание выходит за рамки данной книги. Для дальнейшего изучения посмотрите на описание методов и переменных классов `XML` и `XMLList` в справочнике по языку ActionScript корпорации Adobe. Все технические детали можно найти в спецификации расширения E4X по адресу http://www.ecma-international.org/publications/standards/Ecma-357.htm.

Далее по программе — знакомство с ограничениями безопасности приложения Flash Player. Если изучение этой темы не кажется вам приятным занятием, можете перейти сразу к части II, где будут рассматриваться вопросы отображения содержимого на экране. Просто помните, что если в процессе разработки вы столкнетесь с ошибками безопасности, глава 19 поможет вам решить появившиеся проблемы.

Ограничения безопасности Flash Player

Чтобы предотвратить передачу данных несанкционированным получателям без соответствующего разрешения, приложение Flash Player тщательно проверяет все запросы на загрузку или на доступ к внешним ресурсам либо на взаимодействие с другими SWF- или HTML-файлами. Любой запрос к внешнему ресурсу из SWF-файла (к ресурсу, который не является частью скомпилированного SWF-файла, осуществляющего запрос), отвергается или принимается на основании таких факторов, как:

- операция языка ActionScript, используемая для доступа к ресурсу;
- статус безопасности SWF-файла, выполняющего запрос;
- местоположение ресурса;
- набор явных прав доступа для ресурса, которые определены либо создателем ресурса, либо его распространителем;
- явные права доступа, предоставленные пользователем (например, разрешение на подключение к камере или микрофону пользователя);
- тип приложения Flash Player, выполняющего SWF-файл (например, версия встраиваемого расширения, автономная версия, отладочная версия среды разработки Flash).

В предыдущем списке и на протяжении всей этой главы будут использоваться следующие термины.

Распространитель ресурса — сторона, которая занимается распространением данного ресурса. Обычно это оператор сервера, например администратор сайта или сокетного сервера.

Создатель ресурса — сторона, которая фактически является автором ресурса. Для SWF-файлов создателем ресурса является разработчик на языке ActionScript, который скомпилировал данный SWF-файл.

Пользователь — пользователь компьютера, на котором выполняется приложение Flash Player.

В этой главе сначала рассматриваются ограничения безопасности приложения Flash Player в общих чертах, после чего описывается конкретное влияние безопасности на загрузку содержимого и на обращение к внешним данным.

В этой главе мы рассмотрим ограничения безопасности одной конкретной среды Flash: *приложения Flash Player (версии, реализованной в виде модуля расширения браузера,* и автономной версии). Информацию об ограничениях безопасности, налагаемых другими средами выполнения Flash (например, приложениями Adobe AIR и Flash Lite), можно найти в документации корпорации Adobe.

Чего нет в этой главе

Перед тем как перейти к изучению материала, проясним один момент: безопасность — это очень обширная тема. Всестороннее рассмотрение безопасности приложения Flash Player выходит за рамки этой книги. Более того, в этой главе рассматриваются функции безопасности, предназначенные лишь для защиты пользователей Flash-содержимого, но не обсуждаются вопросы разработки защищенных приложений, например сайтов для электронной коммерции. Гораздо более полную информацию о безопасности, включая вопросы разработки защищенных приложений, например использование протокола Secure Sockets Layer (SSL), создание собственных алгоритмов шифрования и защиту данных, передаваемых по протоколу RTMP, можно найти в таких ключевых ресурсах, как:

- документация корпорации Adobe, раздел Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Flash Player Security;
- «Центр по вопросам безопасности» (Security Topic Center) корпорации Adobe: http://www.adobe.com/devnet/security/;
- технический документ корпорации Adobe, посвященный безопасности: http://www.adobe.com/go/fp9_0_security;
- статья «Security Changes in Flash Player 8» Денеба Мекеты (Deneb Meketa), в основном посвященная локальной безопасности: http://www.adobe.com/devnet/flash/articles/fplayer8_security.html;
- статья «Security Changes in Flash Player 7» Денеба Мекеты, в основном посвященная файлам политики безопасности: http://www.adobe.com/devnet/flash/articles/fplayer_security.html;
- интерактивная справка приложения Flash Player корпорации Adobe, где описываются настройки безопасности, доступные пользователям: http://www.adobe.com/support/documentation/en/flashplayer/help/index.html.

Теперь посмотрим, как безопасность приложения Flash Player влияет на загрузку содержимого и на доступ к внешним данным.

Локальная область действия, удаленная область действия и удаленные регионы

Как будет видно на протяжении этой главы, ограничения безопасности среды выполнения Flash зачастую основываются на местоположении SWF-файлов и внешних ресурсов. Оценивая местоположение ресурса с точки зрения безопасности, среда Flash различает ресурсы в удаленных местоположениях и ресурсы в локальных местоположениях.

В этой главе мы будем использовать термин *«удаленная область действия»* в отношении логической группы всех возможных удаленных местоположений, например Интернета. Соответственно мы будем использовать термин *«локальная область действия»* в отношении логической группы всех возможных локальных местопо-

ложений. Локальное местоположение — это любое местоположение, к которому может обратиться пользователь компьютера, выполняющего приложение Flash Player либо с помощью протокола `file:` (обычно применяется для обращения к локальной файловой системе), либо с помощью пути, соответствующему универсальному соглашению об именах (UNC — universal naming convention) (обычно применяется для обращения к компьютерам в локальной сети).

Сама по себе удаленная область действия подразделяется на различные регионы, концептуально ограничиваемые распространителями ресурсов. Мы будем называть такие регионы *удаленными*. В частности, удаленный регион может являться:

- доменом Интернета;
- поддоменом Интернета;
- IP-адресом, который указывает на компьютер, находящийся в удаленной области действия.

Таким образом, в соответствии с предыдущим списком:

- sitea.com и siteb.com, games.example.com и finances.example.com, 192.150.14.120 и 205.166.76.26 — это разные удаленные регионы;
- 192.150.18.117 и adobe.com — это разные удаленные регионы, несмотря на то что IP-адрес 192.150.18.117 закреплен за доменом adobe.com (поскольку для приложения Flash Player IP-адреса, заданные в числовом виде, и их эквивалентные доменные имена считаются разными).

Термины «удаленная область действия», «локальная область действия» и «удаленный регион» в настоящий момент не являются частью официальной терминологии по безопасности корпорации Adobe. Они используются в этой книге исключительно в пояснительных целях.

Типы безопасности песочниц

Среда выполнения Flash присваивает статус безопасности, называемый *типом безопасности песочницы*, каждому SWF-файлу, открываемому или загружаемому в приложения Flash Player. Существует четыре возможных типа безопасности песочницы: *удаленный* (remote), *локальный с поддержкой файловой системы* (local-with-filesystem), *локальный с поддержкой сети* (local-with-networking) и *локальный с установленным доверием* (local-trusted). Каждый тип безопасности песочницы определяет специальный набор правил, который регламентирует возможности SWF-файла по выполнению операций с внешними источниками данных. В частности, к операциям с внешними источниками данных, которые потенциально могут быть запрещены типами безопасности песочниц, относятся:

- загрузка содержимого;
- обращение к содержимому в виде данных;
- кросс-скриптинг;
- загрузка данных;

- отправка данных на внешний адрес URL;
- обращение к камере и микрофону пользователя;
- обращение к совместно используемым локальным объектам;
- выгрузка и загрузка файлов, выбранных пользователем;
- скриптинг HTML-страницы из SWF-файла и наоборот;
- подключение к каналу LocalConnection.

В этой главе мы увидим, как каждый из перечисленных типов безопасности песочниц влияет на первые пять типов операций с внешними источниками данных из предыдущего списка. Чтобы узнать, как типы безопасности песочниц влияют на оставшиеся типы операций с внешними источниками данных, обратитесь к документации корпорации Adobe — раздел Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Flash Player Security.

Стоит отметить, что, если операция завершилась неудачно из-за ограничений безопасности Flash Player, среда Flash сгенерирует либо ошибку `SecurityError`, либо выполнит диспетчеризацию события `SecurityErrorEvent.SECURITY_ERROR`. Более подробную информацию по обработке ошибок безопасности можно найти в разд. «Обработка нарушений безопасности» ближе к концу этой главы.

Как присваиваются типы безопасности песочниц. Чтобы определить тип безопасности песочницы для конкретного SWF-файла, среда выполнения Flash сначала принимает во внимание местоположение, из которого этот SWF-файл был загружен или открыт. Всем SWF-файлам из удаленной области действия присваивается тип безопасности песочницы «удаленный». В отличие от этого, SWF-файлам из локальной области действия присваивается один из оставшихся типов безопасности песочниц — «локальный с установленным доверием», «локальный с поддержкой сети» или «локальный с поддержкой файловой системы». Конкретный тип безопасности песочницы, присваиваемый локальному SWF-файлу, зависит от двух факторов:

- была ли включена поддержка сети при компиляции SWF-файла (дополнительную информацию можно найти далее, в разд. «Выбор локального типа безопасности песочницы»);
- установлено ли явное доверие для SWF-файла (говорят, что для SWF-файла установлено явное доверие, если этот файл открыт из надежного локального местоположения; дополнительную информацию можно найти далее, в подразд. «Установка локального доверия» разд. «Выбор локального типа безопасности песочницы»).

Всем SWF-файлам из локальной области действия, для которых установлено явное доверие, присваивается тип безопасности песочницы «локальный с установленным доверием». Подобным образом, для исполняемых файлов проектора (то есть отдельных файлов, включающих SWF-файл и определенную версию приложения Flash Player) доверие установлено всегда.

Если для SWF-файлов из локальной области действия не установлено явное доверие, то им присваивается либо тип безопасности песочницы «локальный с поддержкой сети» (для SWF-файлов, которые были откомпилированы с поддержкой сети),

либо тип безопасности песочницы «локальный с поддержкой файловой системы» (для SWF-файлов, откомпилированных без поддержки сети).

Для краткости в этой главе мы будем называть SWF-файлы, которым присвоен тип безопасности песочницы «удаленный», *удаленными SWF-файлами*. Подобным образом мы будем использовать термины *локальный SWF-файл с поддержкой файловой системы*, *локальный SWF-файл с поддержкой сети* и *локальный SWF-файл с установленным доверием* в отношении SWF-файлов с типами безопасности песочниц «локальный с поддержкой файловой системы», «локальный с поддержкой сети» и «локальный с установленным доверием» соответственно.

Как правило, локальные SWF-файлы с поддержкой сети имеют больше возможностей для доступа к удаленной области действия, чем к локальной области действия. В отличие от этого, локальные SWF-файлы с поддержкой сети имеют больше возможностей для доступа к локальной области действия, чем к удаленной области действия.

Чтобы определить тип безопасности песочницы SWF-файла на этапе выполнения, используйте значение переменной flash.system.Security.sandboxType внутри этого SWF-файла.

Поскольку приложение Flash Player присваивает всем SWF-файлам из удаленной области действия тип безопасности песочницы «удаленный», интернет-разработчики должны всегда учитывать ограничения данного типа безопасности песочницы.

В отличие от этого, разработчики, создающие SWF-содержимое, которое предназначено для локальной загрузки или открытия, могут использовать настройки компилятора, файлы конфигурации, инсталляторы и инструкции для выбора одного из трех локальных типов безопасности песочниц. Выбирая тип безопасности песочницы, такие разработчики фактически определяют для своего содержимого логический набор возможностей для доступа к внешним источникам данных.

В следующих разделах мы узнаем, как типы безопасности песочниц SWF-файлов влияют на возможности доступа к внешним источникам данных, а затем подробно рассмотрим механизмы (и логическое обоснование) выбора одного из трех локальных типов безопасности песочниц.

Обобщения в вопросах безопасности вредны

В оставшейся части этой главы мы будем очень подробно рассматривать конкретные операции и ограничения безопасности. При описании правил безопасности в этой книге исключены обобщения в ущерб точности, поскольку обобщения в вопросах безопасности зачастую являются источником досадных недоразумений. Помните, что, если в документации или сторонних ресурсах делаются обобщения, касающиеся вопросов безопасности приложения Flash Player, важные исключения могут остаться без должного внимания. Например, следующее утверждение большей частью является справедливым, и поэтому делается заманчивое обобщение: SWF-файл, типом безопасности песочницы которого является «локальный с поддержкой файловой системы», имеет полный доступ к локальной области действия.

Однако из этого утверждения существует множество важных исключений, включая следующие:

- локальные SWF-файлы с поддержкой файловой системы не могут подключаться к сокетам;
- локальные SWF-файлы с поддержкой файловой системы не могут загружать другие локальные SWF-файлы с поддержкой сети;
- локальные SWF-файлы с поддержкой файловой системы не могут обращаться к данным локальных SWF-файлов с установленным доверием без разрешений создателя;
- для обращения к пользовательской камере и микрофону требуется разрешение пользователя;
- пользователи для любого SWF-файла могут полностью отключить или ограничить возможность сохранения данных в совместно используемых локальных объектах.

Чтобы избежать недоразумений, если в процессе разработки вы столкнетесь с проблемой безопасности, всегда фокусируйтесь на деталях. Определите операцию, которую вы желаете выполнить, тип безопасности песочницы вашего SWF-файла и конкретные ограничения, которые накладывает данный тип безопасности песочницы на выполняемую операцию. Имея на руках эту информацию, вы сможете уверенно работать с любыми ограничениями безопасности или найти способ, чтобы обойти данные ограничения.

В этой главе не рассматриваются абсолютно все ограничения безопасности, налагаемые средой выполнения Flash. Чтобы определить ограничения, которые налагает среда Flash на операции, не рассматриваемые в этой главе, обратитесь к разделам, посвященным данным операциям, в справочнике по языку ActionScript корпорации Adobe.

Теперь рассмотрим, как влияет каждый из четырех типов безопасности песочниц на загрузку содержимого, обращение к содержимому в виде данных, кросс-скриптинг и загрузку данных.

Ограничения на загрузку содержимого, обращение к содержимому в виде данных, кросс-скриптинг и загрузка данных

Для многих разработчиков первое знакомство с системой безопасности языка ActionScript происходит в тот момент, когда выполняемая операция блокируется из соображений безопасности. В этом разделе будет рассказано о четырех наиболее часто блокируемых внешних операциях: загрузке содержимого, обращении к содержимому в виде данных, кросс-скриптинге и загрузке данных. После этого мы рассмотрим условия, при которых происходит блокировка данных распространенных операций.

Загрузка содержимого

Загрузка содержимого подразумевает получение любого внешнего ресурса для его дальнейшего отображения или воспроизведения. По существу, операции загрузки содержимого позволяют разработчикам представлять внешнее содержимое пользователю даже в тех случаях, когда правила безопасности среды выполнения Flash запрещают программный доступ к данным этого содержимого.

Методы языка ActionScript, относящиеся к операциям «загрузки содержимого» с точки зрения безопасности, перечислены в табл. 19.1.

Таблица 19.1. Операции загрузки содержимого

Метод загрузки содержимого	Тип содержимого	Конкретные форматы файлов, поддерживаемые приложением Flash Player 9
flash.display.Loader.load( )	Изображение, файл Adobe Flash	JPEG, GIF, PNG, SWF
flash.media.Sound.load( )	Аудио	MP3
flash.net.NetStream.play( )	Прогрессивное видео	FLV

Для удобства в этой главе время от времени используется термин *«ресурсы содержимого»* в отношении ресурсов, загруженных с помощью одного из методов, представленных в табл. 19.1. Стоит отметить, однако, что внешняя операция становится операцией загрузки содержимого благодаря конкретному методу, используемому для загрузки, а не типу файла данного ресурса. Например, загрузка изображения в формате JPEG с помощью метода экземпляра `load( )` класса `Loader` считается операцией загрузки содержимого, однако загрузка того же JPEG-изображения через бинарный сокет или с помощью метода экземпляра `load( )` класса `URLLoader` уже не считается операцией загрузки содержимого. Это отличие существенно, поскольку к различным категориям операций применяются различные правила безопасности.

Обращение к содержимому в виде данных

Обращение к содержимому в виде данных подразумевает чтение внутренней информации ресурса содержимого, например чтение пикселов растрового изображения или спектра звука. В табл. 19.2 представлены методы языка ActionScript, которые считаются операциями «обращения к содержимому в виде данных» с точки зрения безопасности.

Таблица 19.2. Обращение к содержимому в виде данных, примеры операций

Операция	Описание
Обращение к изображению через переменную экземпляра content класса Loader	Получает объект Bitmap языка ActionScript, представляющий загруженное изображение
Вызов метода экземпляра draw( ) класса BitmapData	Копирует пикселы отображаемого элемента в объект BitmapData
Вызов метода экземпляра computeSpectrum( ) класса SoundMixer	Копирует данные текущей звуковой волны в объект ByteArray
Обращение к переменной экземпляра id3 класса Sound	Читает метаданные в формате ID3 звукового файла

Кросс-скриптинг

Кросс-скриптинг подразумевает обращение к загруженному SWF-файлу программным путем. Многие операции языка ActionScript могут быть применены для кросс-скриптинга SWF-файла, включая следующие, но не ограничиваясь ими:

- использование переменной экземпляра `content` класса `Loader` для получения объекта, представляющего загруженный SWF-файл;
- обращение к переменным загруженного SWF-файла;
- вызов методов загруженного SWF-файла;
- привязывание к классу, определенному в загруженном SWF-файле;
- использование метода экземпляра `draw( )` класса `BitmapData` для копирования пикселов загруженного SWF-файла в объект `BitmapData`.

Описание остальных операций кросс-скриптинга можно найти в справочнике по языку ActionScript корпорации Adobe, где явно указаны ограничения безопасности, которые применяются к каждой операции языка ActionScript.

Загрузка данных

В общем смысле термин «загрузка данных» может использоваться для описания широкого спектра операций загрузки приложения Flash Player, включая загрузку файлов с сервера через метод экземпляра `download( )` класса `FileReference`, загрузку объектов с помощью приложения Flash Remoting, загрузку бинарных данных через объект `Socket` и т. д. Однако для текущего обсуждения (и для оставшейся части данной главы) *загрузка данных* подразумевает следующее:

- загрузку внешнего текста, бинарных данных или переменных с помощью метода экземпляра `load( )` класса `URLLoader`;
- загрузку данных с помощью метода экземпляра `load( )` класса `URLStream`.

Чтобы узнать об ограничениях, налагаемых средой Flash на операции загрузки данных, которые не рассматриваются в этой главе, обратитесь к справочнику по языку ActionScript корпорации Adobe.

Для метода экземпляра `load( )` класса `URLLoader` формат загружаемых данных (текст, бинарные данные или переменные) задается переменной экземпляра `dataFormat` класса `URLLoader`. К обычным текстовым форматам файлов относятся форматы XML, TXT и HTML. К обычным форматам бинарных данных относятся изображения, SWF-файлы и сериализованные объекты в формате ActionScript Message Format (AMF). Тем не менее бинарными данными может быть любой файл или содержимое, загруженное в объект `ByteArray` для дальнейшей обработки в исходном бинарном формате. Переменные загружаются в одном-единственном формате: преобразованные в URL-строки переменные, которые были загружены в виде пар «имя/значение» из внешнего текстового файла или сценария.

Стоит отметить, что, как и в случае с загрузкой содержимого, внешняя операция становится операцией загрузки данных благодаря конкретному методу,

используемому для загрузки, а не типу файла данного ресурса. Например, загрузка SWF-файла с помощью метода экземпляра `load( )` класса `URLLoader` считается операцией *загрузки данных*; загрузка того же SWF-файла с помощью метода экземпляра `load( )` класса `Loader` считается операцией *загрузки содержимого*.

Ограничения на загрузку содержимого, обращение к содержимому в виде данных, загрузку данных и кросс-скриптинг

Теперь, когда мы разобрались, что представляют собой операции загрузки содержимого, обращения к содержимому в виде данных, кросс-скриптинга и загрузки данных, посмотрим, как каждый из четырех типов безопасности песочниц приложения Flash Player ограничивает эти операции.

В следующих четырех таблицах (табл. 19.3–19.6) перечислены условия, при которых каждый тип безопасности песочницы разрешает или запрещает выполнять операции загрузки содержимого, обращения к содержимому в виде данных, кросс-скриптинга и загрузки данных. В каждой таблице представлены конкретные правила, устанавливаемые отдельным типом безопасности песочницы, — разрешается или запрещается использовать внешние операции, перечисленные в крайнем левом столбце, для обращения к ресурсам, перечисленным в оставшихся столбцах таблицы.

Как видно из таблиц, некоторые операции допускаются только при наличии разрешения создателя или распространителя. *Разрешение создателя* подразумевает, что SWF-файл включает надлежащий вызов статического метода `allowDomain( )` класса `Security` (или, в редких случаях, вызов метода `allowInsecureDomain( )`). *Разрешение распространителя* подразумевает, что распространитель ресурса сделал доступным надлежащий файл междоменной политики безопасности. Более подробную информацию можно найти далее, в разд. «Разрешения создателя (allowDomain( ))» и «Разрешения распространителя (файлы политики безопасности)».

Как видно из табл. 19.3–19.6, загрузка содержимого разрешается в большем количестве ситуаций по сравнению с загрузкой содержимого в виде данных, кросс-скриптингом и загрузкой данных. Например, приложению может быть дано разрешение на загрузку и отображение растрового изображения, но запрещен доступ к исходным пиксельным данным этого изображения. Подобным образом приложению может быть дано разрешение на загрузку и отображение внешнего SWF-файла, но может требоваться разрешение на кросс-скриптинг этого SWF-файла.

Стоит отметить, что в табл. 19.3–19.6 рассмотрены разрешения только для одного направления взаимодействия, когда SWF-файл загружает или обращается к внешнему ресурсу. В этих таблицах не описано обратное направление взаимодействия, когда загруженный SWF-файл взаимодействует с SWF-файлом, загрузившим его. Информацию по двунаправленному взаимодействию между SWF-файлами можно найти в разделе Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Flash Player Security ▸ Cross-scripting документации корпорации Adobe.

В табл. 19.3 перечислены правила, устанавливаемые типом безопасности песочницы «удаленный». Здесь выражение «регион происхождения SWF-файла» подразумевает удаленный регион, из которого был открыт или загружен данный SWF-файл. Например, если файл `hiscores.swf` загружается из удаленного местоположения http://coolgames.com/hiscores.swf, то регионом происхождения файла `hiscores.swf` является `coolgames.com` (дополнительные сведения по удаленным регионам можно найти в разд. «Локальная область действия, удаленная область действия и удаленные регионы»).

В табл. 19.3 демонстрируются следующие ключевые правила типа безопасности песочницы «удаленный».

- Операции загрузки содержимого, обращения к содержимому в виде данных, кросс-скриптинга и загрузки данных не могут использоваться вместе с ресурсами из локальной области действия.
- Все ресурсы из всей удаленной области действия могут загружаться в виде содержимого.
- Операции загрузки содержимого, обращения к содержимому в виде данных, кросс-скриптинга и загрузки данных могут использоваться для всех ресурсов из региона происхождения SWF-файла.
- К ресурсам удаленной области действия, находящимся за пределами региона происхождения SWF-файла, можно обращаться как к данным или загружать их как данные, если установлено соответствующее разрешение распространителя.
- Для SWF-файлов из удаленной области действия, находящихся за пределами региона происхождения SWF-файла, может быть выполнен кросс-скриптинг, если установлено соответствующее разрешение создателя.

Таблица 19.3. Тип безопасности песочницы «удаленный», допустимые и запрещенные операции

Операция	Локальная область действия	Ресурсы удаленной области действия из региона происхождения SWF-файла	Ресурсы удаленной области действия, находящиеся за пределами региона происхождения SWF-файла
Загрузка содержимого	Запрещается	Допускается	Допускается
Обращение к содержимому в виде данных	Запрещается	Допускается	Допускается только по разрешению распространителя
Кросс-скриптинг	Запрещается	Допускается	Допускается только по разрешению создателя
Загрузка данных	Запрещается	Допускается	Допускается только по разрешению распространителя

В табл. 19.4 перечислены следующие ключевые правила, устанавливаемые типом безопасности песочницы «локальный с поддержкой файловой системы».

- Операции загрузки содержимого, обращения к содержимому в виде данных, кросс-скриптинга и загрузки данных не могут использоваться вместе с ресурсами из удаленной области действия.

- Операции загрузки содержимого, обращения к содержимому в виде данных и загрузки данных могут использоваться для всех ресурсов, не являющихся SWF-файлами, из локальной области действия.
- Загрузка локальных SWF-файлов с поддержкой сети строго запрещается.
- Допускается загрузка и кросс-скриптинг других локальных SWF-файлов с поддержкой файловой системы.
- Для кросс-скриптинга локальных SWF-файлов с установленным доверием требуется разрешение создателя.

Таблица 19.4. Тип безопасности песочницы «локальный с поддержкой файловой системы», допустимые и запрещенные операции

Операция	Ресурсы, не являющиеся SWF-файлами, в локальной области действия	Локальные SWF-файлы с поддержкой файловой системы	Локальные SWF-файлы с поддержкой сети	Локальные SWF-файлы с установленным доверием	Ресурсы в удаленной области действия
Загрузка содержимого	Допускается	Допускается	Запрещается	Допускается	Запрещается
Обращение к содержимому в виде данных	Допускается	—	—	—	Запрещается
Кросс-скриптинг	—	Допускается	Запрещается	Допускается только по разрешению создателя	Запрещается
Загрузка данных	Допускается	—	—	—	Запрещается

В табл. 19.5 перечислены следующие ключевые правила, устанавливаемые типом безопасности песочницы «локальный с поддержкой сети».

- Операции загрузки содержимого могут использоваться для ресурсов в удаленной области действия.
- Операции загрузки данных и обращения к содержимому в виде данных могут применяться для ресурсов в удаленной области действия, если установлено соответствующее разрешение распространителя.
- Операции загрузки содержимого могут использоваться для ресурсов, не являющихся SWF-файлами, в локальной области действия.
- Операции загрузки данных и обращения к содержимому в виде данных не могут применяться для ресурсов в локальной области действия.
- Загрузка локальных SWF-файлов с поддержкой файловой системы строго запрещается.
- Допускается загрузка и кросс-скриптинг других локальных SWF-файлов с поддержкой сети.
- Для кросс-скриптинга локальных SWF-файлов с установленным доверием или удаленных SWF-файлов требуется разрешение создателя.

Таблица 19.5. Тип безопасности песочницы «локальный с поддержкой сети», допустимые и запрещенные операции

Операция	Ресурсы, не являющиеся SWF-файлами, в локальной области действия	Локальные SWF-файлы с поддержкой файловой системы	Локальные SWF-файлы с поддержкой сети	Локальные SWF-файлы с установленным доверием	Ресурсы в удаленной области действия
Загрузка содержимого	Допускается	Запрещается	Допускается	Допускается	Допускается
Обращение к содержимому в виде данных	Запрещается	—	—	—	Допускается только по разрешению распространителя
Кросс-скриптинг	—	Запрещается	Допускается	Допускается только по разрешению создателя	Допускается только по разрешению создателя
Загрузка данных	Запрещается	—	—	—	Допускается только по разрешению распространителя

В табл. 19.6 перечислены следующие ключевые правила, устанавливаемые типом безопасности песочницы «локальный с установленным доверием».

- Операции загрузки содержимого, обращения к содержимому в виде данных, кросс-скриптинга и загрузки данных могут использоваться для любых ресурсов в локальной и удаленной области действия.
- Тип безопасности песочницы «локальный с установленным доверием» дает SWF-файлу максимально возможную степень свободы из всех, которые может предоставить приложение Flash Player.

Таблица 19.6. Тип безопасности песочницы «локальный с установленным доверием», допустимые и запрещенные операции

Операция	Ресурсы, не являющиеся SWF-файлами, в локальной области действия	Локальные SWF-файлы с поддержкой файловой системы	Локальные SWF-файлы с поддержкой сети	Локальные SWF-файлы с установленным доверием	Ресурсы в удаленной области действия
Загрузка содержимого	Допускается	Допускается	Допускается	Допускается	Допускается
Обращение к содержимому в виде данных	Допускается	—	—	—	Допускается
Кросс-скриптинг	—	Допускается	Допускается	Допускается	Допускается
Загрузка данных	Допускается	—	—	—	Допускается

Мы познакомились с правилами, которые устанавливаются каждым из четырех типов безопасности песочниц для четырех типов операций. Перед тем как перейти к другим вопросам, касающимся безопасности, рассмотрим еще один тип операции — подключение к сокету.

Безопасность сокетов

В языке ActionScript подключение к сокетам осуществляется посредством классов `XMLSocket`, `Socket` и `NetConnection`. В табл. 19.7 и 19.8 перечислены конкретные местоположения и порты, к которым можно открывать сокетные соединения с помощью методов классов `XMLSocket` и `Socket`. В таблицах не описан класс `NetConnection`, который используется в приложениях Adobe Flash Media Server и Adobe Flex. Информацию по безопасности класса `NetConnection` можно найти в документации по этим продуктам.

И в табл. 19.7, и в табл. 19.8 термин *«разрешение распространителя»* подразумевает, что оператор сервера сделал доступным надлежащий файл междоменной политики безопасности. Более подробную информацию можно найти далее, в разд. «Разрешения распространителя (файлы политики безопасности)».

В табл. 19.7 показано, может ли удаленный SWF-файл устанавливать сокетное соединение с четырьмя перечисленными местоположениями.

Таблица 19.7. Удаленный тип безопасности песочницы, допустимые и запрещенные сокетные соединения

Локальная область действия, любой порт	Удаленная область действия внутри региона происхождения SWF-файла, порт 1024 и выше	Удаленная область действия внутри региона происхождения SWF-файла, порт 1023 и ниже	Удаленная область действия за пределами области происхождения SWF-файла, любой порт
Допускается только по разрешению распространителя	Допускается	Допускается только по разрешению распространителя	Допускается только по разрешению распространителя

В табл. 19.8 показано, может ли SWF-файл, тип безопасности песочницы которого представлен в крайнем левом столбце, устанавливать сокетное соединение с местоположениями, перечисленными в остальных столбцах.

Таблица 19.8. Локальные типы безопасности песочниц, допустимые и запрещенные сокетные соединения

Тип безопасности песочницы	Локальная область действия, любой порт	Удаленная область действия, любой порт
Локальный с поддержкой файловой системы	Запрещается	Запрещается
Локальный с поддержкой сети	Допускается только по разрешению распространителя	Допускается только по разрешению распространителя
Локальный с установленным доверием	Допускается	Допускается

Примеры сценариев безопасности

Чтобы подкрепить практикой изученный материал, рассмотрим несколько примеров, которые демонстрируют, как система безопасности приложения Flash Player позволяет предотвратить получение данных посторонними лицами. В каждом сценарии сначала описывается способ, которым мог бы воспользоваться хакер для получения данных при отсутствии системы безопасности приложения Flash Player, а затем порядок действий системы безопасности при предотвращении попытки хакера завладеть целевыми данными.

Шпионское вложение в электронном письме — без системы безопасности Flash Player

Джо Хакер хочет похитить персональные данные Дэйва Юзера. Джо знает, что Дэйв заполняет свои налоговые декларации с помощью приложения ABCTax, работающего под управлением операционной системы Windows. Джо проводит небольшое исследование и обнаруживает, что приложение ABCTax хранит информацию о годовой налоговой декларации в XML-файле, который находится по адресу `c:\ABCTax\taxreturn.xml`. Если Джо сможет получить этот файл, он сможет воспользоваться содержащейся в нем информацией, чтобы открыть банковский счет и обратиться за получением кредитной карты от лица Дэйва. Итак, Джо отправляет по электронной почте письмо Дэйву, прикрепив к письму внешне безобидную анимацию `cartoon.swf`. Дэйв открывает электронное письмо и воспроизводит анимацию в браузере на своей локальной машине. Без ведома Дэйва файл `cartoon.swf` тайно использует метод `URLLoader.load( )` для получения файла `taxreturn.xml` из локальной файловой системы. После этого файл `cartoon.swf` использует метод `flash.net.sendToURL( )`, чтобы выгрузить файл `taxreturn.xml` на сайт Джо.

Джо получает кредитную карту на имя Дэйва и покупает приставку Nintendo Wii с множеством отличных игр.

Шпионское вложение в электронном письме — с системой безопасности Flash Player

Как и раньше, Джо отправляет Дэйву электронное письмо, к которому прикреплена внешне безобидная анимация `cartoon.swf`. Дэйв открывает электронное письмо и воспроизводит анимацию в браузере на своей локальной машине. Поскольку файл `cartoon.swf` открывается из локальной области действия, приложение Flash Player проверяет, была ли включена поддержка сети при компиляции файла `cartoon.swf`.

Сначала предположим, что файл `cartoon.swf` был скомпилирован без поддержки сети. В этом случае приложение Flash Player присваивает файлу `cartoon.swf` тип безопасности песочницы «локальный с поддержкой файловой системы». Как и раньше, файл `cartoon.swf` тайно использует метод `URLLoader.load( )` для получения файла `taxreturn.xml` из локальной файловой системы. В соот-

ветствии с табл. 19.4 файлу `cartoon.swf` разрешается загрузить эти локальные данные. После этого он пытается использовать метод `flash.net.sendToURL( )`, чтобы выгрузить файл `taxreturn.xml` на сайт Джо, однако эта попытка блокируется, поскольку локальным SWF-файлам с поддержкой файловой системы не разрешается выполнять операции `flash.net.sendToURL( )`.

В предыдущих таблицах не рассматривались ограничения безопасности конкретно для метода flash.net.sendToURL( ), однако, как отмечалось ранее, определить ограничения безопасности для любого метода интерфейса API приложения Flash Player можно в справочнике по языку ActionScript корпорации Adobe.

Теперь предположим, что файл `cartoon.swf` был скомпилирован с поддержкой сети. В данном случае приложение Flash Player присваивает ему тип безопасности песочницы «локальный с поддержкой сети». Как и раньше, файл `cartoon.swf` пытается тайно использовать метод `URLLoader.load( )` для получения файла `taxreturn.xml` из локальной файловой системы. Однако эта попытка блокируется, поскольку, в соответствии с табл. 19.5, локальные SWF-файлы с поддержкой сети не могут использовать операции загрузки данных для ресурсов в локальной области действия.

Джо вынужден покупать приставку Nintendo Wii за свои деньги.

Внутренняя корпоративная информация — без системы безопасности Flash Player

Джо Хакер хочет получить некую служебную информацию, чтобы совершить выгодную сделку на бирже. Джо раньше работал в корпорации WYZ, у которой есть общедоступный сайт www.wyzcorp.com. Джо остался в хороших отношениях с корпорацией WYZ, поэтому его наняли по контракту, чтобы обновить сведения о корпорации в файле `profile.swf` на сайте www.wyzcorp.com. Джо знает, что корпорация WYZ планирует выпустить важный продукт, что повлияет на стоимость акций компании. Джо также знает, что даты выпуска новых продуктов корпорации WYZ хранятся на внутреннем сайте, который находится за межсетевым экраном, по следующему адресу: strategy.wyzcorp.com/releasedates.html. Если Джо сможет тайно получить дату выпуска нового продукта, то сможет скупить акции корпорации WYZ за день до выхода этого продукта, а затем выгодно продать их.

Итак, Джо добавляет некий код в файл `profile.swf`, использующий метод `URLLoader.load( )`, чтобы попытаться загрузить файл: strategy.wyzcorp.com/releasedates.html. После этого сотрудник корпорации WYZ просматривает сведения о корпорации по адресу www.wyzcorp.com/profile.swf. Поскольку компьютер сотрудника находится за межсетевым экраном, он имеет доступ к сайту strategy.wyzcorp.com, поэтому попытка загрузить файл `releasedates.html` оказывается успешной! Без ведома сотрудника файл `profile.swf` использует метод `flash.net.sendToURL( )`, чтобы выгрузить файл `releasedates.html` на сайт Джо.

Акции корпорации WYZ взлетают в цене, а Джо теперь может позволить себе заниматься любимым делом.

Внутренняя корпоративная информация — с системой безопасности Flash Player

Как и раньше, Джо размещает файл `profile.swf` на сайте www.wyzcorp.com, который использует метод `URLLoader.load( )`, чтобы попытаться загрузить файл strategy.wyzcorp.com/releasedates.html. После этого сотрудник корпорации WYZ просматривает сведения о корпорации по адресу www.wyzcorp.com/profile.swf. Поскольку файл `profile.swf` открывается в удаленной области действия, среда выполнения Flash присваивает ему тип безопасности песочницы «удаленный». Когда файл www.wyzcorp.com/profile.swf пытается загрузить файл strategy.wyzcorp.com/releasedates.html, попытка блокируется, поскольку, в соответствии с табл. 19.3, удаленный SWF-файл не может использовать операции загрузки данных для ресурсов, находящихся за пределами удаленного региона происхождения этого файла.

Джо надеется получить гигабайт трафика за баннерную рекламу нового продукта корпорации WYZ, чтобы оплатить жилье в следующем месяце.

Межсайтовая информация — без системы безопасности Flash Player

Джо Хакер хочет похитить информацию о некоторых банковских счетах. Он работает в агентстве Hipster Ad Agency, занимающемся производством рекламы для банка ReallyHuge Bank. У этого банка есть система интернет-банкинга, доступная по адресу www.reallyhugebank.com/bank.swf. Приложение `bank.swf` загружает рекламу из файла www.hipsteradagency.com/ad.swf. Джо изучил структуру файла `bank.swf` с помощью декомпилятора SWF-файлов и теперь знает, в каких переменных приложение `bank.swf` хранит номера банковских счетов и пароли. С преступным намерением Джо добавляет в файл `ad.swf` код, который считывает значения этих переменных у своего родителя `bank.swf`, и затем использует метод `flash.net.sendToURL( )` для отправки похищенной информации на сайт Джо. Всякий раз, когда пользователь приложения `bank.swf` обращается к своему счету, Джо получает номер счета и пароль.

Джо жертвует баснословно большие суммы денег организации «Гринпис».

Межсайтовая информация — с системой безопасности Flash Player

Как и раньше, Джо добавляет в файл `ad.swf` код, который считывает значения нужных переменных у своего родителя `bank.swf`. Пользователь запускает приложение www.reallyhugebank.com/bank.swf, а оно, в свою очередь, загружает «зловредный» файл Джо www.hipsteradagency.com/ad.swf. Поскольку файл `ad.swf` открывается из удаленной области действия, среда выполнения Flash присваивает ему тип безопасности песочницы «удаленный». Когда файл `ad.swf` пытается прочитать значения переменных файла `bank.swf`, эта попытка блокируется, поскольку, в соответствии с табл. 19.3, удаленный SWF-файл не может использовать операции

загрузки данных для ресурсов, находящихся за пределами удаленного региона происхождения этого файла.

Джо жертвует скромную сумму денег организации «Гринпис».

Выбор локального типа безопасности песочницы

Вы получили хорошее представление о мерах безопасности, обеспечиваемых каждым типом безопасности песочниц. Кроме того, вы узнали, что SWF-файлам, открытым или загруженным из местоположений удаленной области действия, всегда присваивается тип безопасности песочницы «удаленный», а SWF-файлам, открытым или загруженным из местоположений локальной области действия, присваивается один из трех локальных типов безопасности песочниц.

Теперь поближе познакомимся с механизмами, применяемыми для выбора одного из трех локальных типов безопасности песочниц. В каждом из трех последующих разделов представлен сценарий, в котором разработчик использует один из трех локальных типов безопасности песочниц. Каждый сценарий описывает и логическое обоснование, и сам механизм для выбора каждого типа безопасности песочницы.

Компиляция локального SWF-файла с поддержкой файловой системы

Сьюзен разрабатывает приложение-календарь `calendar.swf`, которое будет размещаться на сайте отеля. Календарь загружает информацию о праздниках из внешнего XML-файла `holiday.xml`. Приложение практически готово, поэтому Сьюзен необходимо отправить его на утверждение заказчику. Несмотря на то что приложение `calendar.swf` со временем будет размещено на сайте, заказчик хочет показать его множеству людей в отеле. В процессе демонстрации заказчик не желает иметь постоянное подключение к Интернету. Итак, Сьюзен отправляет файлы `calendar.swf` и `holiday.xml` заказчику.

Чтобы в процессе демонстрации приложение `calendar.swf` могло загружать файл `holiday.xml` из локальной файловой системы, Сьюзен скомпилировала SWF-файл календаря с установленным в значение `false` флагом компилятора `-use-network`. В зависимости от используемой среды разработки применяются различные механизмы для установки флага компилятора `-use-network`. В приложении Flex Builder 2 Сьюзен выполняет следующие шаги, чтобы установить в значение `false` флаг компилятора `-use-network`.

1. На палитре **Navigator** (Навигатор) Сьюзен выбирает папку проекта для приложения-календаря.
2. В меню **Project** (Проект) она выбирает команду **Properties** (Свойства).
3. В окне **Properties** (Свойства) она выбирает раздел **ActionScript Compiler** (Компилятор ActionScript).

4. В поле ввода Additional compiler arguments (Дополнительные аргументы компилятора) она вводит `-use-network=false`.
5. Чтобы подтвердить установку, Сьюзен нажимает кнопку OK.

В среде разработки Flash Сьюзен выполняет следующие шаги, чтобы установить в значение `false` флаг компилятора `-use-network`:

1. В меню File (Файл) Сьюзен выбирает команду Publish Settings (Настройки публикации).
2. В окне Publish Settings (Настройки публикации) она переходит на вкладку Flash.
3. В раскрывающемся списке Local playback security (Безопасность локального воспроизведения) она выбирает пункт Access local files only (Обращение только к локальным файлам).
4. Чтобы подтвердить установку, Сьюзен нажимает кнопку OK.

При использовании консольного компилятора из бесплатного инструментария разработчика Flex SDK Сьюзен указывает значение флага `-use-network` в качестве параметра компилятора mxmlc. Вот команда, которую выполняет Сьюзен, работая в операционной системе Microsoft Windows:

```
mxmlc.exe -use-network=false -file-specs c:\projects\calendar\Calendar.as
-output c:\projects\calendar\bin\Calendar.swf
```

Компиляция локального SWF-файла с поддержкой сети

Дориан разрабатывает видеоигру `race.swf` для сайта своей компании. Посетители сайта могут играть в эту игру и отправлять свои лучшие результаты в режиме онлайн. Дориан хочет сделать загружаемую версию своей игры, в которую можно будет играть при отсутствии подключения к Интернету. Когда у пользователя отсутствует подключение к Интернету, загружаемая версия игры будет сохранять лучшие результаты в совместно используемом локальном объекте и отправит их на сервер, как только пользователь подключится к Интернету.

Дориан знает, что большая часть посетителей предпочитает не запускать исполняемые файлы из неизвестных источников, поэтому она решает сделать свою игру в виде загружаемого SWF-файла, а не в виде исполняемого файла проектора. Чтобы позволить игре, выполняемой в виде локального SWF-файла, подключаться к серверу, на котором хранятся лучшие результаты, Дориан компилирует приложение `race.swf` с установленным в значение `true` флагом компилятора `-use-network`.

Чтобы установить флаг `-use-network`, Дориан использует тот же механизм, который использовала Сьюзен в предыдущем сценарии с приложением `calendar.swf`, но при установке флага `-use-network` вместо значения `false` указывает значение `true`. При использовании среды разработки Flash Дориан выбирает для параметра Local playback security (Безопасность локального воспроизведения) значение Access network only (Обращение только к сети) вместо значения Access local files only (Обращение только к локальным файлам).

Установка локального доверия

Колин создает инструмент администрирования `admin.swf` для сокетного приложения-сервера. Файл `admin.swf` предназначен для работы либо в том же домене, в котором находится сокетный сервер, либо в локальной файловой системе администратора этого сервера. Инструмент администрирования подключается к удаленному сокетному серверу и не загружает никакие локальные файлы, поэтому Колин компилирует приложение как локальный SWF-файл с поддержкой сети.

Первый экран инструмента администрирования представляет собой простую форму для ввода имени сервера и пароля. При входе в систему инструмент администрирования предлагает сохранить введенный пароль сервера. Если пользователь соглашается, то инструмент администрирования сохраняет пароль в совместно используемом локальном объекте. В следующий раз при входе в систему инструмент администрирования автоматически заполнит текстовое поле для ввода пароля.

В процессе разработки Колин вдруг осознает, что, поскольку файл `admin.swf` является локальным SWF-файлом с поддержкой сети, другие локальные SWF-файлы с поддержкой сети, находящиеся на том же компьютере, на котором выполняется приложение `admin.swf`, смогут загрузить это приложение и прочитать пароль из текстового поля!

Чтобы закрыть эту потенциальную брешь в безопасности, Колин принимает разумное решение: файлу `admin.swf` должен присваиваться тип безопасности песочницы «локальный с установленным доверием». В результате другие локальные SWF-файлы с поддержкой сети не смогут прочитать значение из текстового поля для ввода пароля.

Чтобы превратить файл `admin.swf` в локальный SWF-файл с установленным доверием, Колин создает инсталлятор, который устанавливает доверие для местоположения файла `admin.swf` путем помещения конфигурационного файла в директорию **Global Flash Player Trust** на локальной машине. В соответствии с официальным форматом языка конфигурационный файл содержит одну-единственную текстовую строку: местоположение файла `admin.swf` в локальной файловой системе. В итоге, когда приложение Flash Player загружает файл `admin.swf` из указанного местоположения, этому файлу присваивается тип безопасности песочницы «локальный с установленным доверием».

Полную информацию, посвященную созданию и управлению устанавливающими доверие конфигурационными файлами, можно найти в разделе Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Flash Player Security ▸ Overview of permission controls ▸ Administrative user controls документации корпорации Adobe.

Колин также понимает, что пользователи инструмента администрирования могут захотеть переместить файл `admin.swf` в любое произвольное местоположение. Чтобы позволить пользователю перемещать файл `admin.swf` в новое местоположение, не теряя статус «локальный с установленным доверием», Колин включает инструкцию в документацию по инструменту администрирования (см. врезку выше).

Описанный сценарий с инструментом администрирования сокетного сервера демонстрирует два доступных механизма установления доверия для локального SWF-файла: конфигурационные файлы на компьютере, на котором выполняется приложение Flash Player (обеспечиваемые инсталлятором Колина), и инструмент Flash Player Settings Manager (к которому обращается пользователь). Подробное описание инструмента Flash Player Settings Manager можно найти в разделе Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Flash Player Security ▸ Overview of permission controls ▸ User controls документации корпорации Adobe.

Перенос файла admin.swf в произвольное местоположение. Перед тем как перенести файл `admin.swf` в новое местоположение, не забудьте зарегистрировать его как локальный SWF-файл с установленным доверием, выполнив такую последовательность действий.

1. Откройте инструмент Flash Player Settings Manager, загрузив в браузере следующую страницу: http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager04.html.
2. Щелкните кнопкой мыши на ссылке Edit locations (Управление местоположениями) параметра Always trust files in these locations (Всегда доверять файлам в следующих местоположениях) в разделе Global Security Settings (Глобальные настройки безопасности) и выберите команду Add location (Добавить местоположение).
3. Введите или выберите местоположение, для которого вы хотите установить доверие.

Стоит отметить, что, если приложение Flash Player выполняется в тот момент, когда устанавливается доверие (либо с помощью конфигурационных файлов, либо с помощью инструмента Flash Player Settings Manager), новый доверительный статус для указанного SWF-файла не вступит в силу до тех пор, пока приложение не будет перезапущено. Для версий Flash Player, реализованных в виде дополнительных модулей или элементов управления ActiveX, «перезапуск» подразумевает выключение всех экземпляров приложения Flash Player — даже тех, которые выполняются в других окнах браузера!

Для разработчиков доверие устанавливается автоматически

Чтобы упростить тестирование локального содержимого, предназначенного для публикации в Интернете, приложение Flex Builder 2 корпорации Adobe автоматически устанавливает доверие для разрабатываемых проектов. Для этого оно добавляет запись, описывающую путь для выходной папки каждого проекта (обычно /bin/), в файл `flexbuilder.cfg`, находящийся в директории User Flash Player Trust. Подобным образом отладочная версия приложения Flash Player среды разработки Flash автоматически устанавливает доверие для всех открываемых или загружаемых SWF-файлов в локальной области действия.

Вследствие этого при загрузке элементов в процессе разработки вы можете не столкнуться с проблемами безопасности, которые возникнут у конечного пользователя. Например, SWF-файл, находящийся в папке проекта /bin/, сможет загрузить локальный файл, даже если у него есть только разрешение для доступа к сети.

Чтобы протестировать приложение в тех условиях, в которых будет находиться ваш конечный пользователь, выполняйте его в соответствующей целевой среде. Например, для веб-приложений тестирование необходимо проводить в Интернете. Для локальных приложений без установленного доверия, разрабатываемых в приложении Flex Builder 2, тестирование проводите из локальной папки, для которой не установлено доверие (Рабочий стол операционной системы обычно является именно такой папкой). Для локальных приложений без установленного доверия, разрабатываемых в среде разработки Flash, используйте команду меню File ▸ Publish Preview ▸ HTML (Файл ▸ Просмотр публикации ▸ HTML), чтобы просмотреть приложение в браузере (среда разработки Flash не устанавливает автоматически доверие для содержимого, просматриваемого в браузере).

В процессе отладки вы должны всегда убеждаться в том, что тип безопасности песочницы вашего приложения соответствует типу безопасности песочницы, который будет использоваться при внедрении приложения.

Чтобы проверить тип безопасности песочницы SWF-файла на этапе выполнения, получите значение переменной `flash.system.Security.sandboxType` внутри этого SWF-файла.

Чтобы проверить, для каких папок на данном компьютере установлено доверие, просмотрите конфигурационные файлы, находящиеся в директориях User Flash Player Trust и Global Flash Player Trust, или воспользуйтесь онлайн-инструментом Flash Player Settings Manager, который доступен по адресу http://www.adobe.com/support/documentation/en/flashplayer/help/index.html.

Чтобы удалить доверие для проекта приложения Flex Builder 2 (это позволит имитировать работу конечного пользователя с приложениями без установленного доверия), удалите соответствующий путь из файла `flexbuilder.cfg`, находящегося в директории User Flash Player Trust. Стоит отметить, однако, что, поскольку приложение Flex Builder 2 автоматически восстанавливает файл `flexbuilder.cfg` при создании нового проекта, вам придется удалять данный путь из файла после каждого создания или импорта проекта. Узнать местоположение директорий User Flash Player Trust и Global Flash Player Trust можно в разделе Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Flash Player Security ▸ Overview of permission controls документации корпорации Adobe.

Используемый по умолчанию тип безопасности песочницы

И приложение Flex Builder 2, и консольный компилятор mxmlc устанавливают флаг компилятора `-use-network` в значение `true`, если это значение не указано явно. Таким образом, по умолчанию, если SWF-файл, скомпилированный с помощью приложения Flex Builder 2 или компилятора mxmlc, запускается в локальной области действия из любого местоположения, для которого не установлено доверие, ему будет присвоен тип безопасности «локальный с поддержкой сети».

Каким бы странным это ни казалось, но значением по умолчанию для параметра Local playback secutiry (Безопасность локального воспроизведения) среды разработки Flash является Access local files only (Обращение только к локальным файлам).

Поэтому по умолчанию, если SWF-файл, скомпилированный с помощью среды разработки Flash, запускается в локальной области действия из любого местоположения, для которого не установлено доверие, ему будет присвоен тип безопасности песочницы «локальный с поддержкой файловой системы».

Чтобы избежать недоразумений, всегда явно указывайте желаемое значение для флага компилятора `-use-network` и параметра публикации Local playback secutiry (Безопасность локального воспроизведения) среды разработки Flash.

Разрешения распространителя (файлы политики безопасности)

На протяжении этой главы мы неоднократно видели, как система безопасности приложения Flash Player ограничивает доступ SWF-файла к внешним ресурсам. Теперь рассмотрим, как в некоторых случаях распространитель ресурса может использовать разрешения распространителя, чтобы обойти эти ограничения.

Напомним, что «распространитель ресурса» — это сторона, которая предоставляет ресурс из некоторого удаленного региона. Например, и администратор сайта, и администратор сокетного сервера являются распространителями ресурсов.

Как сторона, отвечающая за ресурсы из конкретного удаленного региона, распространитель ресурса может устанавливать доверие для SWF-файлов из внешних источников, чтобы они могли обращаться к этим ресурсам. Чтобы установить доверие для SWF-файлов на доступ к определенному набору ресурсов, распространитель ресурса использует особый механизм разрешений, называемый *файлом политики безопасности*. Файл политики безопасности — это простой XML-документ, который содержит список доверенных источников SWF-файлов. Вообще говоря, файл политики безопасности предоставляет SWF-файлам из своего списка доверенных источников доступ к ресурсам, которые в обычной ситуации оказываются недоступными из-за ограничений безопасности приложения Flash Player.

К типам операций, которые потенциально могут быть разрешены файлом политики безопасности, относятся:

- загрузка содержимого в виде данных;
- загрузка данных;
- подключение к сокету;
- импортирующая загрузка (рассматривается далее, в разд. «Импортирующая загрузка»).

С помощью файла политики безопасности невозможно разрешить операции кросс-скриптинга. Узнать подробнее о разрешении операции кросс-скриптинга можно в разд. «Разрешения создателя (allowDomain( ))».

Обычно файлы политики безопасности используются для разрешения взаимодействия между различными удаленными регионами. Например, файл политики

безопасности может дать файлу http://site-a.com/map.swf разрешение на чтение пикселов из файла http://site-b.com/satellite-image.jpg или разрешение на загрузку файла http://siteb.com/map-data.xml.

В соответствии с данными из табл. 19.3, 19.5, 19.7 и 19.8 файл политики безопасности позволяет предоставить SWF-файлу доступ к ресурсам, которые оказываются недоступными, когда:

- удаленный SWF-файл пытается выполнить операцию доступа к содержимому в виде данных для ресурса в удаленной области действия, находящегося за пределами его региона происхождения;
- удаленный SWF-файл пытается выполнить операцию загрузки данных для ресурса в удаленной области действия, находящегося за пределами его региона происхождения;
- локальный SWF-файл с поддержкой сети пытается выполнить операцию доступа к содержимому в виде данных для ресурса в удаленной области действия;
- локальный SWF-файл с поддержкой сети пытается выполнить операцию загрузки данных для ресурса в удаленной области действия;
- удаленный SWF-файл пытается подключиться к сокету внутри его региона происхождения, но к порту ниже 1024;
- удаленный SWF-файл пытается подключить к сокету за пределами его региона происхождения;
- локальный SWF-файл с поддержкой сети пытается подключиться к сокету в удаленной области действия.

В следующих разделах описываются механизмы использования файлов политики безопасности распространителями ресурсов для разрешения доступа к ресурсам в каждой из перечисленных ситуаций.

Дополнительную информацию по файлам политики безопасности можно найти в разделе Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Flash Player Security ▸ Overview of permission controls ▸ Web Site controls (cross-domain policy files) документации корпорации Adobe.

В приложении Flash Player 6 файлы политики безопасности применялись исключительно для разрешения междоменного взаимодействия и поэтому назывались файлами междоменной политики безопасности. С момента появления приложения Flash Player версии 7.0.19.0 файлы политики безопасности также начали использоваться для разрешения сокетных подключений к портам из нижнего диапазона. Чтобы как-то отразить это расширенное назначение, в данной книге используется более короткий термин «файл политики безопасности», однако в других источниках вы будете встречать первоначальный термин — «файл междоменной политики безопасности».

Разрешение операций загрузки данных и обращения к содержимому в виде данных

Чтобы предоставить SWF-файлам из определенного подмножества источников разрешение на выполнение операций загрузки данных или обращения к содержимому

в виде данных для заданного набора удаленных ресурсов, используйте следующую общую последовательность действий.

1. Создайте файл политики безопасности.
2. Разместите созданный файл политики безопасности в том же удаленном регионе (то есть в том же домене или для того же IP-адреса), где находится ресурс, для которого устанавливается разрешение.

Предыдущие шаги будут подробно описаны в следующих двух разделах. Познакомившись с тем, как создаются и размещаются файлы политики безопасности, мы рассмотрим процесс получения SWF-файлом разрешения из файла политики безопасности для выполнения операций загрузки данных и обращения к содержимому в виде данных.

Создание файла политики безопасности

Для создания файла политики безопасности используйте такую последовательность действий.

1. Создайте новый текстовый файл.
2. Добавьте в файл список желаемых разрешенных источников, используя официальный синтаксис корпорации Adobe для файлов политики безопасности.
3. Сохраните текстовый файл.

Официальный синтаксис корпорации Adobe для файлов политики безопасности основан на языке XML и имеет следующую структуру:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="доменИлиIP"/>
</cross-domain-policy>
```

Здесь *доменИлиIP* определяет доменное имя или IP-адрес разрешенного источника. SWF-файлу, загруженному из разрешенного источника, разрешается выполнять операции загрузки данных и обращения к содержимому в виде данных для заданного набора ресурсов. Как вы узнаете из следующего раздела, конкретный набор ресурсов, доступ к которым разрешает файл политики безопасности, определяется местоположением этого файла.

Файл политики безопасности может содержать любое количество тегов `<allow-access-from>`. Например, следующий файл политики безопасности определяет три разрешенных источника: example1.com, example2.com и example3.com.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="example1.com"/>
  <allow-access-from domain="example2.com"/>
  <allow-access-from domain="example3.com"/>
</cross-domain-policy>
```

Когда символ * используется в значении атрибута domain, он обозначает подстановочный символ. Например, следующий файл политики безопасности разрешает доступ для сайта example1.com и его любого поддомена, независимо от уровня вложенности (например, games.example1.com, driving.games.example1.com и т. д.):

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="*.example1.com"/>
</cross-domain-policy>
```

Сам по себе символ * разрешает доступ для всех источников:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="*"/>
</cross-domain-policy>
```

Чтобы включить локальную область действия в список разрешенных источников, файл политики безопасности должен явным образом установить доверие для всех источников. Для этого должен быть указан символ * (любой источник) в качестве значения атрибута domain. Таким образом, чтобы локальные SWF-файлы с поддержкой сети могли загружать XML-файлы с сайта, атрибуту domain файла политики безопасности этого сайта *должно быть* установлено значение *.

Размещение файла политики безопасности

После создания файл политики безопасности должен быть размещен в том же удаленном регионе (то есть домене или IP-адресе), где находится ресурс, к которому разрешается доступ. Например, если файл политики безопасности разрешает доступ к содержимому сайта www.example.com, файл политики безопасности должен быть также размещен на сайте www.example.com.

Набор ресурсов, к которым файл политики безопасности разрешает доступ, определяется конкретным местоположением этого файла. Если файл политики безопасности размещается в корневой директории сайта, то доступ будет разрешен ко всему сайту. Например, файл политики безопасности, размещенный по адресу http://www.example.com, разрешает доступ ко всему содержимому сайта www.example.com.

Когда файл политики безопасности размещается в поддиректории сайта, он разрешает доступ только к этой директории и к ее поддиректориям. Например, файл политики безопасности, размещенный по адресу http://www.example.com/assets, разрешает доступ ко всему содержимому директории /assets/ и ее поддиректорий, но не разрешает доступ ни к корневой директории сайта www.example.com, ни к любой другой поддиректории этого сайта.

С целью автоматизации процесса загрузки файлов политики безопасности язык ActionScript определяет используемое по умолчанию имя и местоположение для файлов политики безопасности. Любой файл политики безопасности с именем crossdomain.xml, размещенный в корневой директории сайта, считается

размещаемым в *используемом по умолчанию местоположении файла политики безопасности* и называется *используемым по умолчанию файлом политики безопасности* этого сайта. Как вы узнаете в следующих двух разделах, размещение файла политики безопасности в используемом по умолчанию местоположении файла политики безопасности сокращает объем кода, необходимого для получения разрешений из этого файла политики безопасности.

Получение разрешения на загрузку данных из файла политики безопасности

Когда на сайте размещается используемый по умолчанию файл политики безопасности, устанавливающий разрешение для определенного удаленного региона, SWF-файлы из этого региона могут загружать данные с сайта, просто выполняя желаемую операцию загрузки данных. Предположим, что на сайте site-a.com размещен следующий используемый по умолчанию файл политики безопасности, который разрешает доступ для источников site-b.com и www.site-b.com:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.site-b.com"/>
  <allow-access-from domain="site-b.com"/>
</cross-domain-policy>
```

Чтобы загрузить файл http://site-a.com/assets/file.xml, любой SWF-файл из источника www.site-b.com или site-b.com может использовать следующий код:

```
var urlloader:URLLoader = new URLLoader( );
urlloader.load(new URLRequest("http://site-a.com/assets/file.xml"));
```

Поскольку файл политики безопасности сайта site-a.com размещается в используемом по умолчанию местоположении, приложение Flash Player находит его автоматически и разрешает загрузить файл `file.xml`.

С другой стороны, если файл политики безопасности сайта размещается не в используемом по умолчанию местоположении, SWF-файлы из разрешенных удаленных регионов должны вручную загрузить этот файл перед тем, как попытаться загрузить данные с этого сайта. Чтобы загрузить файл политики безопасности вручную, используется статический метод `loadPolicyFile( )` класса `Security`, который имеет следующий общий вид:

```
Security.loadPolicyFile("http://доменИлиIP/путьКФайлуПолитикиБезопасности");
```

В приведенном обобщенном коде *доменИлиIP* — это домен или IP-адрес сайта, на котором размещается файл политики безопасности, а *путьКФайлуПолитикиБезопасности* — местоположение файла политики безопасности на этом сервере. Как уже упоминалось ранее, IP-адреса, заданные в числовом виде, и их эквивалентные доменные имена для приложения Flash Player считаются разными.

Предположим, что сайт site-c.com размещает следующий файл политики безопасности по адресу http://site-c.com/assets/policy.xml. Этот файл политики безопасности разрешает доступ для источников site-d.com и www.site-d.com.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.site-d.com"/>
  <allow-access-from domain="site-d.com"/>
</cross-domain-policy>
```

Чтобы загрузить файл **http://site-c.com/assets/file.xml**, любой SWF-файл из источников **www.site-d.com** или **site-d.com** может использовать следующий код:

```
// Сначала загружаем файл политики безопасности
Security.loadPolicyFile("http://site-c.com/assets/policy.xml");
// Затем выполняем операцию загрузки
var urlloader:URLLoader = new URLLoader( );
urlloader.load(new URLRequest("http://site-c.com/assets/file.xml"));
```

Обратите внимание, что в приведенном коде вызов команды загрузки данных происходит сразу после вызова команды загрузки файла политики безопасности. Приложение Flash Player автоматически дожидается загрузки файла политики безопасности перед тем, как перейти к выполнению операции загрузки данных.

После того как файл политики безопасности будет загружен с помощью метода `Security.loadPolicyFile( )`, его разрешения будут действовать для всех последующих операций загрузки данных, осуществляемых этим SWF-файлом. Например, следующий код вручную загружает файл политики безопасности, после чего выполняет две операции загрузки, которые зависят от разрешений этого файла политики безопасности:

```
// Один раз загружаем файл политики безопасности
Security.loadPolicyFile("http://site-c.com/assets/policy.xml");
// Выполняем две разрешенные операции загрузки
var urlloader1:URLLoader = new URLLoader( );
urlloader1.load(new URLRequest("http://site-c.com/assets/file1.xml"));
var urlloader2:URLLoader = new URLLoader( );
urlloader2.load(new URLRequest("http://site-c.com/assets/file2.xml"));
```

Рассмотрим практический пример, который демонстрирует, как файл политики безопасности, размещенный в поддиректории сайта, может быть использован в реальной ситуации. Предположим, Грейм поддерживает сайт с бесплатной информацией о котировках акций **stock-feeds-galore.com**. Он сохраняет самую последнюю полученную информацию в XML-файле, который размещается по следующему адресу:

```
stock-feeds-galore.com/latest/feed.xml
```

Грейм хочет сделать так, чтобы содержимое директории **/latest/** было доступно всем SWF-файлам из любого источника, но не хочет делать доступным весь сайт. Таким образом, Грейм размещает следующий файл политики безопасности с именем `policy.xml` в директории **/latest/** (обратите внимание на использование подстановочного символа `*` в значении атрибута `domain`):

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
```

```
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="*"/>
</cross-domain-policy>
```

После этого Грейм размещает на сайте stock-feeds-galore.com сообщение, которое информирует разработчиков на языке ActionScript о местоположении файла политики безопасности:

```
stock-feeds-galore.com/latest/policy.xml
```

Тем временем Джеймс создает приложение `stockticker.swf` для отображения котировок акций, которое он собирается разместить на своем сайте www.some-news-site.com. Приложение Джеймса загружает последнюю информацию об акциях с сайта Грейма. Поскольку файл политики безопасности сайта www.stock-feeds-galore.com не находится в местоположении, используемом по умолчанию, Джеймс вынужден загрузить файл политики безопасности перед тем, как выполнить операцию загрузки информации об акциях. Вот код, который использует Джеймс для загрузки файла политики безопасности Грейма:

```
Security.loadPolicyFile("http://stock-feeds-galore.com/latest/policy.xml")
```

Инициировав запрос на загрузку файла политики безопасности, Джеймс использует объект `URLLoader` для загрузки файла `feed.xml`, как показано в следующем коде:

```
var urlLoader:URLLoader = new URLLoader( );
urlLoader.load(new URLRequest("http://stock-feeds-galore.com/latest/feed.xml"));
```

В результате выполнения предыдущего кода приложение Flash Player загружает файл http://stock-feeds-galore.com/latest/policy.xml, находит необходимое разрешение в этом файле политики безопасности и переходит к загрузке файла `feed.xml`.

Теперь, когда мы познакомились с механизмом получения разрешения из файла политики безопасности на загрузку данных, рассмотрим механизм получения из файла политики безопасности разрешения на выполнение операции обращения к содержимому в виде данных.

Получение разрешения на доступ к содержимому в виде данных из файла политики безопасности

Код, используемый для получения разрешения на доступ к содержимому в виде данных из файла политики безопасности, зависит от типа данных, к которым происходит обращение. Чтобы получить разрешение на обращение к *изображению* в виде данных из файла политики безопасности, выполняйте следующие шаги.

1. Если файл политики безопасности не находится в местоположении, используемом по умолчанию, загрузите его методом `Security.loadPolicyFile( )` (как описывалось в предыдущем разделе).
2. Создайте объект `LoaderContext` и присвойте значение `true` его переменной `checkPolicyFile`.
3. Загрузите нужное изображение с помощью метода `Loader.load( )`; в качестве параметра `context` метода `Loader.load( )` передайте объект `LoaderContext`, созданный на шаге 2.

4. После загрузки изображения выполните операцию обращения к содержимому в виде данных.

Предположим, что сайт site-a.com размещает следующий файл политики безопасности по адресу http://site-a.com/assets/policy.xml. Файл разрешает доступ для источников site-b.com и www.site-b.com.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.site-b.com"/>
  <allow-access-from domain="site-b.com"/>
</cross-domain-policy>
```

Чтобы обратиться к файлу http://site-a.com/assets/image.jpg в виде данных, любой SWF-файл из источника www.site-b.com или site-b.com может использовать следующий код:

```
// Шаг 1: Файл политики безопасности не находится в местоположении,
//        используемом по умолчанию, поэтому загружаем его вручную.
Security.loadPolicyFile("http://site-a.com/assets/policy.xml");

// Шаг 2: Создаем объект LoaderContext и присваиваем его переменной
//        checkPolicyFile значение true.
var loaderContext = new LoaderContext( );
loaderContext.checkPolicyFile = true;

// Шаг 3: Загружаем изображение. Передаем объект LoaderContext в метод
//        Loader.load( ).
theLoader.load(new URLRequest("http://site-a.com/assets/image.jpg"),
                              loaderContext);

// Шаг 4: Позднее, когда приложение убедится, что загрузка изображения
//        завершена, обращаемся к содержимому изображения в виде данных.
trace(theLoader.content);
```

Чтобы получить разрешение из файла политики безопасности на обращение к внешнему *звуковому файлу* в виде данных, выполняйте следующие шаги.

1. Если файл политики безопасности не находится в местоположении, используемом по умолчанию, загрузите его методом `Security.loadPolicyFile( )` (как описывалось в предыдущем разделе).
2. Создайте объект `SoundLoaderContext` и присвойте значение `true` его переменной `checkPolicyFile`.
3. Загрузите желаемый звуковой файл с помощью метода экземпляра `load( )` класса `Sound`. В качестве параметра `context` метода `load( )` передайте объект `SoundLoaderContext`, созданный на шаге 2.
4. После успешного завершения загрузки звукового файла (определить окончание загрузки можно с помощью событий класса `Sound`, информирующих о ходе выполнения загрузки) выполните разрешенную операцию обращения к содержимому в виде данных.

Предположим, что на сайте site-c.com размещен следующий используемый по умолчанию файл политики безопасности, который разрешает доступ для источников site-d.com и www.site-d.com.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.site-d.com"/>
  <allow-access-from domain="site-d.com"/>
</cross-domain-policy>
```

Чтобы обратиться к файлу http://site-c.com/sounds/song.mp3 в виде данных, любой SWF-файл из источника www.site-d.com или site-d.com может использовать следующий код:

```
// Шаг 1: Файл политики безопасности находится в местоположении,
//        используемом по умолчанию, поэтому загружать его вручную не нужно

// Шаг 2: Создаем объект SoundLoaderContext и присваиваем его переменной
//        checkPolicyFile значение true.
var soundLoaderContext = new SoundLoaderContext( );
soundLoaderContext.checkPolicyFile = true;

// Шаг 3: Загружаем звуковой файл. Передаем объект SoundLoaderContext
//        в метод Loader.load( ).
theSound.load(new URLRequest("http://example.com/sounds/song.mp3");

// Шаг 4: Позднее, когда приложение убедится, что загрузка метаданных
//        в формате ID3 звукового файла завершена (это можно определить
//        по событию Event.ID3), обращаемся к содержимому звукового файла
//        в виде данных.
trace(theSound.id3);
```

Следует отметить, что присваивание значения `true` переменной экземпляра `checkPolicyFile` классов `LoaderContext` или `SoundLoaderContext` не влияет на загрузку элемента. При выполнении метода `load( )` классов `Loader` или `SoundLoader` элемент загружается всегда, даже если файл политики безопасности не разрешает доступ для региона происхождения данного SWF-файла. Тем не менее, если код в этом SWF-файле попытается обратиться к загруженному элементу в виде данных, приложение Flash Player сгенерирует исключение `SecurityError`.

Рассмотрим реальный пример, который демонстрирует, как используемый по умолчанию файл политики безопасности сайта может быть применен для разрешения операции обращения к содержимому в виде данных.

Помните сайт Грейма stock-feeds-galore.com? Сайт работает так хорошо, что у Грейма появляется свободное время. Он решает поэкспериментировать с программированием растровой графики на языке ActionScript и создает приложение для распознавания лиц, которое может автоматически добавлять забавную шляпу для вечеринок на любую фотографию лица человека. Грейм очень доволен собой.

Друг Грейма Энди управляет компанией, организующей лотереи, у которой есть рекламный сайт www.lotterylotterylottery.com. Энди увидел приложение Грейма и решил, что оно позволит провести хорошую рекламную кампанию. Кампания заключается в том, что победители лотерей размещают свои фотографии на сайте photos.lotterylotterylottery.com. После этого основной сайт www.lotterylotterylottery.com выбирает случайную фотографию для домашней страницы, отображая победителя лотереи в шляпе для вечеринок. Энди нанимает Грейма, чтобы реализовать программный код для этой кампании.

Грейм размещает свое приложение для распознавания лиц `partyhat.swf` на сайте www.lotterylotterylottery.com. После этого он пишет сценарий на языке Perl `randompic.pl`, который возвращает случайную фотографию (JPG-файл) с сайта photos.lotterylotterylottery.com. Он размещает сценарий `randompic.pl` в директории photos.lotterylotterylottery.com/cgi-bin.

Файлу `partyhat.swf` с сайта www.lotterylotterylottery.com требуется доступ к пикселам фотографий, загруженных с сайта photos.lotterylotterylottery.com. Чтобы разрешить этот доступ, Грейм размещает следующий файл политики безопасности в корневой директории сайта photos.lotterylotterylottery.com и присваивает ему имя `crossdomain.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.lotterylotterylottery.com"/>
  <allow-access-from domain="lotterylotterylottery.com"/>
</cross-domain-policy>
```

Обратите внимание, что Грейм добавил оба источника — www.lotterylotterylottery.com и lotterylotterylottery.com — в файл политики безопасности. Таким образом, приложение `partyhat.swf` будет функционировать правильно независимо от того, с какого из указанных URL-адресов оно будет загружено. Грейм также не позабыл исключить домен «*», поскольку его политика безопасности применяется только к конкретным доменам, а не ко всему миру.

Чтобы загрузить фотографию, Грейм использует следующий код (обратите внимание, что вызывать метод `Security.loadPolicyFile( )` не обязательно, поскольку Грейм разместил файл политики безопасности в используемом по умолчанию местоположении файла политики безопасности).

```
var loaderContext = new LoaderContext( );
loaderContext.checkPolicyFile = true;
loader.load(
   new URLRequest("http://photos.lotterylotterylottery.com/randompic.pl"),
   loaderContext);
```

В результате выполнения приведенного кода приложение Flash Player загружает файл http://photos.lotterylotterylottery.com/crossdomain.xml, находит в нем требуемое разрешение, загружает фотографию, возвращаемую сценарием `randomic.pl`, и после этого разрешает приложению `partyhat.swf` обратиться к пикселам загруженной фотографии.

После загрузки фотографии приложение `partyhat.swf` благополучно обращается к ее данным. Например, вот код, используемый Греймом для вызова метода приложения `partyhat.swf`, который добавляет шляпу для вечеринок к загруженной фотографии (стоит отметить, что для обращения к объекту `Bitmap` загруженного изображения `loader.content` требуется разрешение):

```
addHat(loader.content);
```

Теперь, когда мы познакомились с механизмами использования файлов политики безопасности для разрешения операций загрузки данных и обращения к содержимому в виде данных, рассмотрим, как можно использовать файлы политики безопасности для разрешения сокетных соединений.

Использование файла политики безопасности для разрешения сокетных соединений

Чтобы разрешить сокетные соединения с помощью файла политики безопасности, используйте следующую общую последовательность действий.

1. Создайте файл политики безопасности.
2. Сделайте так, чтобы этот файл был доступен через сокетный сервер или HTTP-сервер, запущенный в том же домене или для того же IP-адреса, с которым планируется установить сокетное соединение.

Описанные шаги подробно рассматриваются в следующих трех разделах.

Создание файла политики безопасности

Файлы политики безопасности, разрешающие установку сокетных соединений, в основном имеют такой же синтаксис, как и файлы политики безопасности, разрешающие выполнение операций загрузки данных и обращения к содержимому в виде данных. Однако в файлах первого типа тег `<allow-access-from>` содержит дополнительный атрибут `to-ports`, как показано в следующем коде:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="доменИлиIP" to-ports="порты"/>
</cross-domain-policy>
```

Атрибут `to-ports` определяет порты, к которым может подключаться SWF-файл из источника *доменИлиIP*. Порты можно задавать по отдельности (разделяя значения запятыми) или диапазонами (разделяя значения символом -). Например, приведенный далее файл политики безопасности устанавливает следующие разрешения:

- SWF-файлы из источника example1.com могут подключаться к портам 9100 и 9200;
- SWF-файлы из источника example2.com могут подключаться к портам в диапазоне от 10 000 до 11 000.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="example1.com" to-ports="9100,9200"/>
  <allow-access-from domain="example2.com" to-ports="10000-11000"/>
</cross-domain-policy>
```

В значении атрибута `to-ports` символ * является подстановочным символом. Когда файл политики безопасности загружается через сокет на порте меньшем чем 1024, символ * означает, что доступ разрешен к любому порту. Когда файл политики безопасности загружается через сокет на порте, большем или равном 1024, символ * означает, что можно обращаться к любому порту, большему или равному 1024.

Поскольку порты ниже 1024 считаются привилегированными, файл политики безопасности, получаемый через порт 1024 или выше, не сможет разрешить доступ к портам ниже 1024, даже если они будут указаны явно.

Например, если получение следующего файла политики безопасности осуществляется через порт 2000, SWF-файлам с сайта example3.com будет дано разрешение на подключение ко всем портам выше или равным 1024.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="example3.com" to-ports="*"/>
</cross-domain-policy>
```

Но если получение того же файла политики безопасности осуществляется через порт 1021 (который меньше чем 1024), то SWF-файлам с сайта example3.com будет дано разрешение на подключение к *любому* порту.

Таким образом, чтобы SWF-файлы из *любого* местоположения могли подключаться к *любому* порту, получение следующего файла политики безопасности должно осуществляться через порт ниже 1024:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="*" to-ports="*"/>
</cross-domain-policy>
```

Когда получение файла политики безопасности осуществляется через сокет, атрибут `to-ports` является обязательным. Если этот атрибут не указан, то доступ будет запрещен ко всем портам.

Теперь, когда мы знаем, как создавать файл политики безопасности, разрешающий сокетное соединение, рассмотрим, как SWF-файл может получить разрешение из этого файла политики безопасности.

Получение файла политики безопасности через сокет

Файлы политики безопасности, разрешающие сокетные соединения, могут передаваться либо непосредственно через сокет, либо по протоколу HTTP. Файлы политики безопасности, передаваемые через сокет, должны размещаться в том же домене или на том же IP-адресе, с которым планируется устанавливать соединение. При этом можно использовать либо порт, с которым устанавливается соединение, либо другой порт. В любом случае сервер, обслуживающий порт, через который передается файл политики безопасности, должен общаться с приложением Flash Player с помощью очень простого протокола получения файла политики безопасности. Протокол состоит из одного тега `<policy-file-request/>`, который отправляется приложением Flash Player через сокет, когда оно желает загрузить файл политики безопасности, разрешающий сокетное соединение. В ответ сокетный сервер должен отправить приложению Flash Player текст файла политики безопасности в формате ASCII вместе с нулевым байтом (то есть пустым символом таблицы ASCII) и после этого закрыть соединение.

Таким образом, пользовательские серверы, желающие обрабатывать запросы на получение файла политики безопасности и осуществлять нормальное взаимодействие на одном и том же порте, должны реализовать и код, отвечающий на запросы на получение файла политики безопасности, и код, который управляет обычным сокетным взаимодействием. Если сервер обрабатывает запросы на получение файла политики безопасности и осуществляет обычное взаимодействие на одном и том же порте, то SWF-файлы из разрешенных регионов могут подключаться к этому серверу, выполняя желаемую операцию сокетного соединения.

Предположим, что многопользовательский игровой сервер, размещенный на сайте site-a.com, обрабатывает игровые запросы и запросы на получение файла политики безопасности на порте 3000. Файл политики безопасности этого сервера разрешает доступ для источников www.site-b.com и site-b.com, как показано в следующем коде:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.site-b.com" to-ports="3000"/>
  <allow-access-from domain="site-b.com" to-ports="3000"/>
</cross-domain-policy>
```

Чтобы подключиться к порту 3000 сайта site-a.com, любой SWF-файл, загруженный из источника www.site-b.com или site-b.com, может использовать следующий код:

```
var socket:Socket = new Socket( );
try {
  socket.connect("site-a.com", 3000);
} catch (e:SecurityError) {
  trace("Connection problem!");
  trace(e.message);
}
```

При выполнении предыдущего кода перед тем, как будет разрешено требуемое подключение к порту 3000, приложение Flash Player автоматически создает отдельное

подключение к порту 3000 и отправляет сообщение `<policy-file-request/>` игровому серверу. Игровой сервер отправляет в ответ файл политики безопасности сайта site-a.com и после этого закрывает соединение. В списке разрешенных регионов данного файла политики безопасности содержится источник SWF-файла, пытающегося установить соединение, что позволяет разрешить исходное сокетное соединение. Вообще, создается два различных подключения: одно для получения файла политики безопасности и затем еще одно для выполнения исходного запроса на установление сокетного соединения.

В некоторых ситуациях для сервера не представляется возможным ответить на запрос приложения Flash Player на получение файла политики безопасности. Например, SWF-файл может захотеть подключиться к существующему почтовому SMTP-серверу, который не понимает назначения инструкции `<policy-file-request/>`. Чтобы разрешить это соединение, администратор почтового сервера должен сделать файл политики безопасности доступным через другой порт того же домена или того же IP-адреса, где находится этот почтовый сервер. Сервер, прослушивающий другой порт, может быть чрезвычайно простым сокетным сервером, который просто ожидает подключения, получает инструкции `<policy-file-request/>`, в ответ возвращает файл политики безопасности и затем закрывает подключение.

Когда получение файла политики безопасности осуществляется через порт, отличный от порта желаемого сокетного соединения (как в нашем примере с почтовым сервером), SWF-файлы из разрешенных регионов должны загружать этот файл политики безопасности вручную перед тем, как выполнить запрос на установление желаемого сокетного соединения. Чтобы вручную загрузить файл политики безопасности через произвольный порт, применяется следующий обобщенный код:

```
Security.loadPolicyFile("xmlsocket://доменИлиIP:номерПорта");
```

Здесь *доменИлиIP* — это доменное имя или IP-адрес сервера, а *номерПорта* — номер порта, через который будет получен файл политики безопасности. Повторим, что IP-адреса, заданные в числовом виде, и их эквивалентные доменные имена для приложения Flash Player считаются разными. В предыдущем коде обратите внимание на обязательное использование протокола `xmlsocket://`. Название протокола описывает тип подключения, используемого для получения файла политики безопасности, а не тип подключения, который разрешает этот файл политики безопасности.

Файл политики безопасности, загруженный с помощью протокола xmlsocket://, разрешает подключения с использованием обоих классов Socket и XMLSocket, а не только с использованием класса XMLSocket.

Отправив инициированный вручную запрос на получение файла политики безопасности, можно сразу же отправлять последующий запрос на подключение к желаемому порту. Предположим, что на порте 1021 сайта site-c.com запущен простой сервер, обслуживающий запросы на получение файла политики безопасности, и что файл политики безопасности сайта site-c.com разрешает установление соединений к порту 25 из источников site-d.com и www.site-d.com. Вот файл политики безопасности:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.site-d.com" to-ports="25"/>
  <allow-access-from domain="site-d.com" to-ports="25"/>
</cross-domain-policy>
```

Чтобы подключиться к порту 25 сайта site-c.com, любой SWF-файл, загруженный из источников site-d.com или www.site-d.com, может использовать следующий код. Обратите внимание, что SWF-файл запрашивает сокетное соединение с портом 25 сразу после отправки запроса на получение файла политики безопасности через порт 1021. Приложение Flash Player терпеливо дожидается окончания загрузки файла политики безопасности перед тем, как продолжить подключение к порту 25.

```
// Загружаем файл политики безопасности вручную
Security.loadPolicyFile("xmlsocket://site-c.com:1021");
var socket:Socket = new Socket( );
try {
  // Пытаемся установить соединение (сразу после того, как был запрошен файл
  // политики безопасности)
  socket.connect("site-c.com", 25);
} catch (e:SecurityError) {
  trace("Connection problem!");
  trace(e.message);
}
```

При выполнении предыдущего кода приложение Flash Player перед тем, как разрешить запрашиваемое подключение к порту 25, устанавливает отдельное подключение к порту 1021 и отправляет сообщение `<policy-file-request/>` серверу, прослушивающему этот порт. Сервер отправляет в ответ файл политики безопасности сайта site-c.com и после этого закрывает подключение. В списке разрешенных регионов этого файла политики безопасности содержится источник SWF-файла, устанавливающего соединение, поэтому установление соединения с портом 25 может быть продолжено.

Теперь рассмотрим альтернативный способ для разрешения сокетного соединения: файл политики безопасности, передаваемый по протоколу HTTP.

Получение файла политики безопасности по протоколу HTTP

Приложение Flash Player до версии 7.0.19.0 требовало, чтобы файлы политики безопасности, разрешающие сокетные соединения, передавались по протоколу HTTP. Язык ActionScript 3.0, в основном для обратной совместимости, продолжает поддерживать механизм разрешения сокетных соединений посредством файлов политики безопасности, передаваемых по протоколу HTTP. Тем не менее, чтобы разрешить сокетное соединение, файл политики безопасности, передаваемый по протоколу HTTP, должен удовлетворять следующим требованиям:

- называться `crossdomain.xml`;
- размещаться в корневой директории веб-сервера;
- передаваться через порт 80 домена или IP-адреса, с которым устанавливается желаемое сокетное соединение;

- в языке ActionScript 3.0 файл должен загружаться вручную с помощью метода `Security.loadPolicyFile( )`.

Более того, в файлах политики безопасности, передаваемых по протоколу HTTP, не используется атрибут `to-ports`. Вместо этого доступ предоставляется ко всем портам, большим или равным 1024.

Файл политики безопасности, передаваемый по протоколу HTTP, не может разрешать сокетные подключения к портам ниже 1024 (однако стоит отметить, что до появления приложения Flash Player версии 9.0.28.0 это правило не действовало из-за ошибки).

Чтобы получить разрешение на установление заданного сокетного соединения из файла политики безопасности, передаваемого по протоколу HTTP, необходимо вручную загрузить этот файл до того, как будет предпринята попытка установить соединение, как показано в следующем обобщенном коде:

```
Security.loadPolicyFile("http://доменИлиIP/crossdomain.xml");
```

В предыдущем коде *доменИлиIP* — это точное имя домена или точный IP-адрес желаемого сокетного соединения.

Отправив запрос на получение файла политики безопасности по протоколу HTTP, можно сразу же отправлять последующий запрос на подключение к желаемому порту. Предположим, что у сайта **site-a.com** есть следующий файл политики безопасности, размещенный на веб-сервере по адресу **http://site-a.com/crossdomain.xml**. Этот файл политики безопасности разрешает доступ для источников **site-b.com** и **www.site-b.com**:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.site-b.com"/>
  <allow-access-from domain="site-b.com"/>
</cross-domain-policy>
```

Чтобы подключиться к порту 9100 сайта **site-a.com**, любой SWF-файл, загруженный из источников **site-b.com** или **www.site-b.com**, может использовать следующий код.

```
// Запрашиваем файл политики безопасности по протоколу HTTP перед попыткой
// установления соединения
Security.loadPolicyFile("http://site-a.com/crossdomain.xml");
var socket:Socket = new Socket( );
try {
  // Пытаемся установить соединение (сразу после того, как был отправлен
  // запрос на получение файла политики безопасности
  socket.connect("site-a.com", 9100);
} catch (e:SecurityError) {
  trace("Connection problem!");
  trace(e.message);
}
```

При выполнении предыдущего кода приложение Flash Player перед тем, как разрешить запрашиваемое подключение к порту 9100, загружает файл политики

безопасности сайта site-c.com по протоколу HTTP. В списке разрешенных регионов этого файла политики безопасности содержится источник SWF-файла, устанавливающего соединение, поэтому подключение к порту 9100 может быть продолжено.

Мы познакомились со способами, с помощью которых распространитель ресурса может предоставить внешним SWF-файлам разрешение на загрузку данных, обращение к содержимому в виде данных и подключение к сокетам. В следующем разделе мы продолжим изучение механизмов разрешений приложения Flash Player, рассмотрев, как создатель SWF-файла может разрешить операцию кросс-скриптинга для SWF-файлов из внешних источников.

Разрешения создателя (allowDomain( ))

Как уже говорилось, разрешения распространителя позволяют выполнять операции обращения к содержимому в виде данных, загрузки данных и сокетных соединений. Разрешения распространителя называются так потому, что они должны быть установлены распространителем ресурса, к которому предоставляется доступ.

В отличие от этого, *разрешения создателя* — это разрешения, устанавливаемые создателем SWF-файла, а не его распространителем. Разрешения создателя более ограниченны, чем разрешения распространителя; они разрешают только операции кросс-скриптинга и скриптинга SWF-файлов из HTML-файлов.

В этой книге не рассматриваются операции скриптинга SWF-файлов из HTML-файлов. Подробную информацию о безопасности и операциях скриптинга SWF-файлов из HTML-файлов можно найти в разделах, описывающих статические методы allowDomain( ) и allowInsecureDomain( ), справочника по языку ActionScript корпорации Adobe.

В отличие от разрешений распространителя, не зависящих от содержимого, к которому предоставляется доступ, разрешения создателя устанавливаются из SWF-файлов. Вызывая метод `Security.allowDomain( )` из SWF-файла, разработчик может разрешить SWF-файлам из внешних источников выполнять операции кросс-скриптинга над этим SWF-файлом. Например, файл `app.swf` содержит следующую строку кода:

```
Security.allowDomain("site-b.com");
```

В этом случае любой SWF-файл, загруженный из источника site-b.com, может выполнять кросс-скриптинг файла `app.swf`. Более того, поскольку вызов метода `allowDomain( )` происходит внутри SWF-файла, предоставляемые разрешения остаются действительными независимо от размещения этого SWF-файла.

В отличие от разрешений распространителя, разрешения создателя передаются вместе с SWF-файлом, для которого они установлены.

Метод `allowDomain( )` имеет следующий обобщенный вид:

```
Security.allowDomain("доменИлиIP1", "доменИлиIP2",..."доменИлиIPn")
```

где "*доменИлиIP1*", "*доменИлиIP2*"... "*доменИлиIPn*" — это список строк, содержащих доменные имена или IP-адреса разрешенных источников. SWF-файл, загруженный из разрешенного источника, может выполнять операции кросс-скриптинга над SWF-файлом, вызвавшим метод `allowDomain( )`.

Как и в случае с файлами политики безопасности, символ * обозначает подстановочный символ. Например, следующий код разрешает доступ для всех источников (то есть любой SWF-файл из любого источника может выполнять кросс-скриптинг SWF-файла, который содержит следующую строку кода):

```
Security.allowDomain("*");
```

Чтобы включить локальную область действия в список разрешенных источников, в качестве разрешаемого домена в метод `allowDomain( )` должен передаваться символ * (любой источник). Например, SWF-файл, желающий разрешить операцию кросс-скриптинга для локальных SWF-файлов с поддержкой сети, должен указать символ * в качестве разрешаемого домена.

Тем не менее, когда символ * используется в параметрах метода `allowDomain( )`, он не может применяться в качестве подстановочного символа поддомена (это слегка необычное поведение отличается от использования группового символа в файлах политики безопасности). Например, следующий код не разрешает доступ для всех поддоменов сайта **example.com**:

```
// Внимание: Не используйте этот код! Подстановочные символы поддоменов
// не поддерживаются.
Security.allowDomain("*.example.com");
```

Сразу после вызова метода `allowDomain( )` любой SWF-файл из разрешенного источника может немедленно приступать к выполнению разрешенных операций. Предположим, что телевизионная сеть использует универсальное приложение для воспроизведения анимации, которое размещается на сайте **www.sometvnetwork.com**. Проигрыватель загружает анимационные ролики в формате SWF с сайта **animation.sometvnetwork.com**. Для управления воспроизведением загруженных роликов проигрыватель вызывает основные методы класса `MovieClip` (`play( )`, `stop( )` и т. д.) над этими роликами. Поскольку проигрыватель и анимационные ролики загружаются из различных поддоменов, проигрыватель должен получить разрешение на вызов методов класса `MovieClip` над роликами. Таким образом, каждый конструктор основного класса анимационного ролика включает следующую строку кода, которая предоставляет необходимое разрешение проигрывателю:

```
Security.allowDomain("www.sometvnetwork.com", "sometvnetwork.com");
```

Обратите внимание, что, поскольку проигрыватель может быть открыт из источника **www.sometvnetwork.com** или **sometvnetwork.com**, анимационные файлы разрешают доступ для обоих доменов. Для загрузки анимационных роликов проигрыватель использует следующий код:

```
var loader:Loader = new Loader( );
loader.load(
 new URLRequest("http://animation.sometvnetwork.com/названиеАнимации.swf"));
```

Сразу после выполнения конструктора основного класса анимационного ролика проигрыватель может приступать к управлению загруженным роликом.

Чтобы гарантировать, что разрешения на выполнение операций кросс-скриптинга будут предоставлены сразу после инициализации SWF-файла, вызывайте метод Security.allowDomain( ) внутри метода-конструктора основного класса этого SWF-файла.

SWF-файл может определить, есть ли у него разрешение на выполнение операций кросс-скриптинга над загруженным SWF-файлом, проверив значение переменной `childAllowsParent` объекта `LoaderInfo` загруженного файла.

Дополнительную информацию о загрузке SWF-файлов можно найти в гл. 28. Информацию по вызову методов клипа над загруженными SWF-файлами можно получить в разд. «Проверка типов на этапе компиляции для динамически загружаемых элементов» гл. 28.

Разрешение операций кросс-скриптинга над SWF-файлами, загружаемыми по протоколу HTTPS, для файлов, загружаемых по протоколу HTTP. Когда SWF-файл загружается по протоколу HTTPS, приложение Flash Player запрещает вызов метода `allowDomain( )`, разрешающий доступ для источников, не использующих протокол HTTPS. Тем не менее разработчики, желающие разрешить доступ для источников, не использующих протокол HTTPS, из SWF-файла, загружаемого по протоколу HTTPS, могут, соблюдая должную осторожность, использовать метод `Security.allowInsecureDomain( )`.

Разрешение доступа для источника, не использующего протокол HTTPS, из SWF-файла, загружаемого по протоколу HTTPS, считается угрожающе опасным и категорически не рекомендуется.

Синтаксис и использование метода `allowInsecureDomain( )` такие же, как и у метода `allowDomain( )`, рассмотренного в предыдущем разделе. Единственное отличие заключается в том, что метод `allowInsecureDomain( )` позволяет разрешить доступ для источников, не использующих протокол HTTPS, из SWF-файла, загружаемого по протоколу HTTPS. В подавляющем большинстве ситуаций для предоставления разрешений создателя следует использовать метод `allowDomain( )` вместо `allowInsecureDomain( )`. Найти описание особых ситуаций, требующих использования метода `allowInsecureDomain( )`, можно в разделе, посвященном методу `Security.allowInsecureDomain( )`, справочника по языку ActionScript корпорации Adobe.

Импортирующая загрузка

В гл. 28 будет рассказано, как SWF-файл родителя может особым образом загрузить SWF-файл ребенка, что позволит родителю непосредственно использовать классы ребенка так, будто они определены в родителе. Эта методика требует, чтобы SWF-файл родителя импортировал классы SWF-файла ребенка в свой *домен приложения*. Рассмотрим базовый код, который должен включать SWF-файл родителя (обратите внимание на использование переменной экземпляра `applicationDomain` класса `LoaderContext`):

```
var loaderContext:LoaderContext = new LoaderContext( );
loaderContext.applicationDomain = ApplicationDomain.currentDomain;
var loader:Loader = new Loader( );
loader.load(new URLRequest("ребенок.swf"), loaderContext);
```

При выполнении приведенного кода попытка импортировать классы ребенка в домен приложения родителя будет заблокирована системой безопасности приложения Flash Player в следующих ситуациях:

- если SWF-файл родителя и SWF-файл ребенка загружены из различных удаленных регионов удаленной области действия;
- если SWF-файл родителя загружен из локальной области действия и имеет тип безопасности песочницы, отличный от типа безопасности песочницы SWF-файла ребенка.

В первом случае распространитель SWF-файла ребенка может использовать файл политики безопасности, чтобы предоставить разрешение SWF-файлу родителя на импорт классов SWF-файла ребенка. Ниже представлены шаги, которые должны выполнить распространитель SWF-файла ребенка и создатель SWF-файла родителя.

1. Распространитель SWF-файла ребенка должен разместить файл политики безопасности, разрешающий доступ для источника SWF-файла родителя, как было описано в разд. «Разрешения распространителя (файлы политики безопасности)».
2. Если файл политики безопасности находится не в применяемом по умолчанию местоположении, родитель должен загрузить его вручную, используя метод `Security.loadPolicyFile( )`, опять же как было описано в разд. «Разрешения распространителя (файлы политики безопасности)».
3. При загрузке SWF-файла ребенка SWF-файл родителя должен передать в метод `load( )` объект `LoaderContext`, переменной `securityDomain` которого присвоено значение `flash.system.SecurityDomain.currentDomain`.

Предположим, что на сайте site-a.com размещен следующий используемый по умолчанию файл политики безопасности, который разрешает доступ для источников site-b.com и www.site-b.com.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.site-b.com"/>
  <allow-access-from domain="site-b.com"/>
</cross-domain-policy>
```

Теперь предположим следующее: файл site-b.com/parent.swf желает импортировать классы файла site-a.com/child.swf в свой домен приложения. Для этого файл site-b.com/parent.swf использует следующий код (обратите внимание, что метод `Security.loadPolicyFile( )` не используется, поскольку файл политики безопасности находится в используемом по умолчанию местоположении файла политики безопасности):

```
var loaderContext:LoaderContext = new LoaderContext( );
loaderContext.applicationDomain = ApplicationDomain.currentDomain;
loaderContext.securityDomain = SecurityDomain.currentDomain;
loader.load(new URLRequest("http://site-a.com/child.swf"), loaderContext);
```

Использование переменной `securityDomain` для получения разрешения распространителя на импорт классов SWF-файла в домен приложения (как показано в предыдущем коде) называется *импортирующей загрузкой*.

Стоит отметить, что, когда некоторый файл `a.swf` применяет импортирующую загрузку для загрузки другого файла `b.swf`, приложение Flash Player считает, будто файл `b.swf` был сначала скопирован, а затем загружен непосредственно с сайта файла `a.swf`. Таким образом, к файлу `b.swf` применяются привилегии безопасности файла `a.swf`, а исходные привилегии безопасности файла `b.swf`, связанные с его реальным источником, аннулируются. Например, файл `b.swf` теряет возможность обращаться к ресурсам из своего реального источника с помощью относительных URL-адресов. Следовательно, при использовании импортирующей загрузки всегда проверяйте функционирование загруженного SWF-файла.

Импортирующая загрузка не требуется в следующих ситуациях, поскольку SWF-файлу родителя, по сути, разрешается импортировать классы SWF-файла ребенка в свой домен приложения:

- локальный SWF-файл импортирует классы из другого локального SWF-файла с таким же типом безопасности песочницы;
- удаленный SWF-файл импортирует классы из другого удаленного SWF-файла, находящегося в том же регионе.

Механизмы обращения к классам в загруженных SWF-файлах рассматриваются в гл. 28 и 31.

Обработка нарушений безопасности

В этой главе мы рассмотрели несколько правил безопасности, которые влияют на способность SWF-файла выполнять различные операции языка ActionScript. Когда операция не может быть выполнена из-за нарушения правила безопасности, среда выполнения либо генерирует исключение `SecurityError`, либо осуществляет диспетчеризацию события `SecurityErrorEvent.SECURITY_ERROR`.

Исключение `SecurityError` генерируется в том случае, когда можно сразу определить, что операция нарушает правило безопасности. Например, если локальный SWF-файл с поддержкой файловой системы пытается открыть сокетное соединение, среда выполнения ActionScript немедленно выявляет факт нарушения безопасности и генерирует исключение `SecurityError`.

В отличие от этого, диспетчеризация события `SecurityErrorEvent.SECURITY_ERROR` происходит в том случае, когда после ожидания завершения некоторой асинхронной задачи среда Flash определяет, что было нарушено правило безопасности. Например, когда локальный SWF-файл с поддержкой сети использует метод экземпляра `load( )` класса `URLLoader` для загрузки файла из удаленной области действия,

среда Flash должна асинхронно проверить наличие подходящего файла политики безопасности, разрешающего операцию загрузки. Если операция проверки файла политики безопасности завершится неудачей, среда Flash выполнит диспетчеризацию события `SecurityErrorEvent.SECURITY_ERROR` (заметьте — не исключение `SecurityError`).

В отладочной версии приложения Flash Player выявить необработанные исключения `SecurityError` и события `SecurityErrorEvent.SECURITY_ERROR` очень просто. Всякий раз, когда возникает данное исключение или событие, приложение Flash Player выводит окно, в котором описывается возникшая проблема. Отладочная версия приложения Flash Player абсолютно отличается от рабочей версии, в которой никакая информация о необработанных исключениях `SecurityError` и событиях `SecurityErrorEvent.SECURITY_ERROR` не отображается, поэтому выявить проблему может оказаться чрезвычайно сложно.

Чтобы убедиться, что никакие нарушения безопасности не остались незамеченными, всегда проверяйте код в отладочной версии приложения Flash Player.

Для обработки ошибок безопасности используется инструкция `try/catch/finally`. Для обработки событий `SecurityErrorEvent.SECURITY_ERROR` применяются приемники событий. Например, следующий код генерирует исключение `SecurityError`, пытаясь установить сокетное подключение к порту выше 65 535. При возникновении ошибки код добавляет сообщение о причине ее возникновения в объект `TextField`, отображаемый на экране.

```
var socket:Socket = new Socket( );
try {
  socket.connect("example.com", 70000);
} catch (e:SecurityError) {
  output.appendText("Connection problem!\n");
  output.appendText(e.message);
}
```

Подобным образом, пытаясь загрузить файл данных с сайта, не имеющего файла политики безопасности, локальный SWF-файл с поддержкой сети, содержащий следующий код, вызовет диспетчеризацию события `SecurityErrorEvent.SECURITY_ERROR`. Перед тем как попытаться осуществить операцию загрузки, код регистрирует приемник события, который выполняется при диспетчеризации события `SecurityErrorEvent.SECURITY_ERROR`.

```
var urlloader:URLLoader = new URLLoader( );
// Регистрируем приемник события
urlloader.addEventListener(SecurityErrorEvent.SECURITY_ERROR,
                           securityErrorListener);
// Вызываем нарушение безопасности
urlloader.load(new URLRequest("http://www.example.com/index.xml"));
```

На момент издания этой книги на сайте example.com не было файла политики безопасности, размещенного в местоположении, используемом по умолчанию, поэтому предыдущий код приводил к диспетчеризации события SecurityErrorEvent.SECURITY_ERROR.

Приемник для предыдущего события SecurityErrorEvent.SECURITY_ERROR, код которого приведен ниже, добавляет сообщение о причине возникновения ошибки в объект TextField, отображаемый на экране, — output:

```
private function securityErrorListener (e:SecurityErrorEvent):void {
  output.appendText("Loading problem!\n");
  output.appendText(e.text);
}
```

Чтобы определить, может ли эта операция генерировать исключение SecurityError или приводить к диспетчеризации события SecurityErrorEvent.SECURITY_ERROR, обратитесь к соответствующему разделу справочника по языку ActionScript корпорации Adobe. Описание каждой операции содержит список возможных исключений SecurityError под заголовком Throws (Генерирует) и список возможных событий SecurityErrorEvent.SECURITY_ERROR под заголовком Events (События).

В большинстве случаев класс, который определяет операцию, генерирующую событие SecurityErrorEvent.SECURITY_ERROR, одновременно является классом, в котором должны регистрироваться приемники событий. Например, класс URLLoader определяет операцию load( ), которая может приводить к диспетчеризации событий SecurityErrorEvent.SECURITY_ERROR. Приемники событий, обрабатывающие события SecurityErrorEvent.SECURITY_ERROR, вызванные методом экземпляра load( ) класса URLLoader, регистрируются в том экземпляре класса URLLoader, над которым вызывается метод load( ). Это демонстрирует следующий код:

```
// При использовании класса URLLoader регистрируйтесь на события
// в экземпляре класса URLLoader.
var urlloader:URLLoader = new URLLoader( );
urlloader.addEventListener(SecurityErrorEvent.SECURITY_ERROR,
                           securityErrorListener);
```

Однако в некоторых случаях класс, определяющий операцию, которая генерирует событие SecurityErrorEvent.SECURITY_ERROR, не является одновременно классом, в котором должны регистрироваться приемники событий. Например, класс Loader определяет операцию load( ), которая может приводить к диспетчеризации событий SecurityErrorEvent.SECURITY_ERROR. Но приемники событий, обрабатывающие данные события, должны регистрироваться в экземпляре класса LoaderInfo, ассоциированном с операцией load( ), а не в экземпляре класса Loader, над которым вызывается метод load( ). Как и раньше, это демонстрирует следующий код:

```
// При использовании класса Loader регистрируйтесь на события
// в экземпляре класса LoaderInfo.
var loader:Loader = new Loader( );
loader.contentLoaderInfo.addEventListener(SecurityErrorEvent.SECURITY_ERROR,
                                          securityErrorListener);
```

Чтобы определить класс, в котором должны регистрироваться приемники событий SecurityErrorEvent.SECURITY_ERROR для данной операции, обратитесь к справочнику по языку ActionScript корпорации Adobe. В частности, эту информа-

цию можно найти в описании класса, определяющего операцию, которая приводит к диспетчеризации события `SecurityErrorEvent.SECURITY_ERROR`.

Домены безопасности

В этом разделе вы получите общее представление о термине *«домен безопасности»* и его обычном эквиваленте *«песочница безопасности»*, который часто применяется в документации корпорации Adobe и других сторонних источниках. В данной книге ни один из этих терминов не используется. Причины этого будут описаны далее.

Определенный SWF-файл и логический набор ресурсов, к которым этот SWF-файл может свободно обращаться, используя операции обращения к содержимому в виде данных, загрузки данных и кросс-скриптинга (в соответствии с табл. 19.3–19.6), вместе концептуально формируют группу, называемую *доменом безопасности*. В табл. 19.9 перечислены составляющие домена безопасности SWF-файла каждого из четырех типов безопасности песочниц.

Не путайте понятия «тип безопасности песочницы» и «домен безопасности». Тип безопасности песочницы — это общий статус безопасности SWF-файла, а домен безопасности — это логический набор ресурсов. Тип безопасности песочницы SWF-файла на самом деле определяет домен безопасности этого файла во многом аналогично тому, как уровень доступа сотрудника компании может определять доступные зоны в здании компании.

Таблица 19.9. Домены безопасности по типу безопасности песочницы

Тип безопасности песочницы SWF-файла	Составляющие домена безопасности
Удаленный	Ресурсы в формате, отличном от формата SWF, из региона происхождения данного SWF-файла. SWF-файлы из региона происхождения данного SWF-файла
Локальный с поддержкой файловой системы	Ресурсы в формате, отличном от формата SWF, в локальной области действия. Локальные SWF-файлы с поддержкой файловой системы в локальной области действия
Локальный с поддержкой сети	Локальные SWF-файлы с поддержкой сети в локальной области действия
Локальный с установленным доверием	Любые ресурсы в формате, отличном от формата SWF, в локальной и удаленной области действия. Локальные SWF-файлы с установленным доверием в локальной области действия

При обсуждении домены безопасности часто описываются в терминах регионов — метафорических безопасных зон. Таким образом, можно сказать, что SWF-файл принадлежит домену безопасности, находится в нем или помещается в свой домен безопасности. Подобным образом можно сказать, что ресурс доступен для SWF-файла, поскольку он принадлежит домену безопасности этого SWF-файла.

Существует всего четыре типа безопасности песочницы, однако для каждого из них есть множество доменов безопасности. Например, любой SWF-файл в удаленной

области действия имеет один и тот же тип безопасности песочницы — «удаленный». Но удаленный SWF-файл из источника site-a.com и удаленный SWF-файл из источника site-b.com являются частями двух различных доменов безопасности (один — для источника site-a.com, а другой — для источника site-b.com). Подобным образом любой SWF-файл из местоположения с установленным доверием в локальной области действия имеет один и тот же тип безопасности песочницы — «локальный с установленным доверием». Однако два локальных SWF-файла с установленным доверием из различных корпоративных локальных сетей являются частями двух различных доменов безопасности (каждый для своей локальной сети).

В документации корпорации Adobe при описании ресурсов, к которым имеет или не имеет доступ SWF-файл, часто используется термин «домен безопасности» (и его обычный эквивалент *«песочница безопасности»*).

Двусмысленное использование термина «песочница». Как уже известно из предыдущего раздела, и в неофициальной литературе, посвященной безопасности приложения Flash Player, и в документации корпорации Adobe часто используется термин «песочница безопасности» и даже просто «песочница» в качестве обычного эквивалента формального термина «домен безопасности». Более того, в некоторых редких случаях, в неофициальной литературе и в документации корпорации Adobe также применяется термин «песочница безопасности» в качестве обычного эквивалента термина «тип безопасности песочницы».

При чтении документации, посвященной безопасности, помните, что термин «песочница» обычно используется для обозначения домена безопасности, а в некоторых случаях — для обозначения типа безопасности песочницы.

Чтобы избежать подобной путаницы, в этой книге обычные термины «песочница безопасности» и «песочница» не используются вообще, а официальный термин «домен безопасности» используется только в случае крайней необходимости (например, при обсуждении внутреннего класса `SecurityDomain`). Кроме того, статус безопасности SWF-файла всегда описывается по отношению к *типу безопасности песочницы* этого SWF-файла. Ресурсы, к которым может обращаться SWF-файл, в этой книге перечисляются явным образом вместо использования общего термина *«домен безопасности»* для описания логической группы доступных ресурсов.

Например, рассмотрим такое предложение из раздела Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Flash Player Security ▸ Security Sandboxes ▸ Local sandboxes документации корпорации Adobe:

```
Local files that are registered as trusted are placed in the local-trusted
sandbox.
```

Если использовать термины, которым отдается предпочтение в этой книге, предыдущая цитата будет звучать так: Локальным файлам, зарегистрированным в качестве файлов с установленным доверием, присваивается тип безопасности песочницы «локальный с установленным доверием».

Далее рассмотрим следующее предложение, на этот раз из раздела Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Flash Player Security ▸ Accessing loaded media as data документации корпорации Adobe:

By default, a SWF file from one *security sandbox* cannot obtain pixel data or audio data from graphic or audio objects rendered or played by loaded media in another *sandbox*.

Если использовать термины, которым отдается предпочтение в этой книге, приведенная цитата будет звучать так: По умолчанию SWF-файл из одного домена безопасности не может получать данные о пикселах либо аудиоданные из графических или звуковых объектов, отображаемых или воспроизводимых загрузившим их элементом за пределами данного домена безопасности.

Если значение термина «песочница» в документации корпорации Adobe или в неофициальных источниках кажется двусмысленным, сосредоточьте внимание на изученных типах безопасности песочниц, а также на разрешенных и запрещенных операциях. В противном случае просто попытайтесь выполнить желаемую операцию и используйте сообщения об ошибках, связанных с нарушением безопасности, которые генерируются компилятором или на этапе выполнения, чтобы определить, разрешена ли данная операция. Тем не менее, чтобы гарантировать, что будут отображены все сообщения обо всех возможных ошибках, связанных с нарушениями безопасности, следуйте рекомендациям, представленным ранее в подразд. «Для разработчиков доверие устанавливается автоматически» разд. «Выбор локального типа безопасности песочницы», и проводите тестирование в отладочной версии приложения Flash Player. Помните также, что вы можете проверить тип безопасности песочницы SWF-файла на этапе выполнения, получив значение переменной `flash.system.Security.sandboxType`. Знание типа безопасности песочницы SWF-файла поможет вам определить ограничения, накладываемые на этот SWF-файл приложением Flash Player.

Две распространенные проблемы разработки, связанные с безопасностью

При изучении этой главы мы рассмотрели множество ограничений безопасности и механизмов предоставления разрешений. Чтобы завершить наше изучение системы безопасности приложения Flash Player, рассмотрим два сценария, связанных с безопасностью, которые часто встречаются в типичном процессе разработки приложений на языке ActionScript: обращение к поддоменам Интернета и обращение к переменной экземпляра `context` класса `Loader`. В каждом сценарии описывается ограничение и соответствующий обходной путь для него.

Обращение к поддоменам Интернета

Из представленной ранее табл. 19.3 известно, что удаленный SWF-файл может загружать данные только из своего удаленного региона происхождения. Из разд. «Локальная область действия, удаленная область действия и удаленные регионы» вы также узнали, что два различных поддомена Интернета, например www.example.com и games.example.com, считаются разными удаленными регионами. Таким образом, SWF-файл, загруженный из источника http://example.com, может загружать любой

файл данных, размещенный в источнике http://example.com, но не может загружать файлы данных, размещенные в любом другом домене, включая поддомены, например games.example.com. Как ни удивительно, но это означает, что SWF-файл, загруженный из источника http://example.com, не может использовать абсолютный URL-адрес для обращения к файлу, размещенному на сайте www.example.com. Для предоставления SWF-файлу, загруженному из источника example.com, разрешения на загрузку элементов с сайта www.example.com используется файл политики безопасности — этот механизм был рассмотрен ранее в разд. «Разрешения распространителя (файлы политики безопасности)».

Следующая последовательность шагов описывает действия владельца сайта example.com, желающего создать файл политики безопасности, который разрешает SWF-файлам, загружаемым из источника example.com, загружать файлы данных с сайта www.example.com и наоборот.

1. Создайте новый текстовый файл с именем `crossdomain.xml`.
2. Откройте файл `crossdomain.xml` в текстовом редакторе.
3. Добавьте следующий XML-код в этот файл:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
  SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="www.example.com" />
  <allow-access-from domain="example.com" />
</cross-domain-policy>
```

4. Сохраните файл `crossdomain.xml`.
5. Выгрузите файл `crossdomain.xml` в корневую директорию сайта example.com (чтобы этот файл был доступен по адресу http://example.com/crossdomain.xml).

Обращение к переменной экземпляра content класса Loader

Когда отображаемый элемент из внешнего источника загружается с помощью объекта `Loader`, экземпляр загруженного элемента помещается в переменную `content` объекта `Loader`. Как было сказано ранее в подразд. «Обращение к содержимому в виде данных» разд. «Ограничения на загрузку содержимого, обращение к содержимому в виде данных, кросс-скриптинг и загрузку данных», обращение к значению переменной `content` считается либо операцией обращения к содержимому в виде данных (если объект, содержащийся в переменной `content`, является изображением), либо операцией кросс-скриптинга (если объект, содержащийся в переменной `content`, является экземпляром основного класса SWF-файла). Таким образом, в соответствии с ограничениями типов безопасности песочниц, представленными в табл. 19.3–19.5, обращение к загруженному элементу с помощью переменной `content` без необходимого разрешения приведет к возникновению ошибки безопасности в ситуациях, когда:

- удаленный SWF-файл использует переменную `content` для обращения к ресурсу, загружаемому из другого удаленного региона;

- локальный SWF-файл с поддержкой сети использует переменную `content` для обращения к ресурсу, загружаемому из удаленной области действия;
- локальный SWF-файл с поддержкой сети использует переменную `content` для обращения к локальному SWF-файлу с установленным доверием;
- локальный SWF-файл с поддержкой файловой системы использует переменную `content` для обращения к локальному SWF-файлу с установленным доверием.

Если вы столкнетесь с любой из описанных ситуаций в своем коде, то должны рассмотреть возможность полностью избежать использования переменной `content`. Если ваше приложение должно только отображать загруженный элемент на экране, то обращаться к переменной `content` совершенно не обязательно. Чтобы отобразить загруженный элемент на экране, не обращаясь к переменной `content`, просто добавьте объект `Loader` этого элемента — а не значение переменной `content` — прямо в список отображения. Дополнительную информацию и пример кода можно найти в подразд. «Отображение загруженного элемента на экране» разд. «Использование класса Loader для загрузки отображаемых элементов на этапе выполнения» гл. 28.

Стоит отметить, однако, что в следующих ситуациях использование переменной `content` обязательно, и, чтобы избежать ситуаций нарушения безопасности, должны быть установлены соответствующие разрешения создателя или распространителя:

- SWF-файлу, загружающему элемент, требуется доступ к данным элемента, например, чтобы прочитать пикселы растрового изображения;
- SWF-файлу, загружающему элемент, необходимо выполнить операцию кросс-скриптинга для загруженного элемента;
- к загруженному элементу необходимо обращаться непосредственно в виде объекта, например, когда объект, представляющий загруженный элемент, должен передаваться в метод, аргументом которого является объект `Bitmap`.

Дополнительную информацию о классе `Loader` можно найти в гл. 28, про список отображения вы прочитаете в гл. 20.

К части II!

В первой части вы познакомились с большинством основополагающих концепций языка ActionScript. В части II мы сосредоточимся на конкретной составляющей интерфейса API среды выполнения Flash, называемой *API отображения*. Впереди нас ожидает множество примеров кода и реальных сценариев программирования, поэтому будьте готовы применить свои знания базового языка ActionScript, заработанные таким тяжелым трудом!

Отображение и интерактивность

В этой части представлены методы отображения содержимого на экране и обработки событий ввода. К рассматриваемым темам относятся API отображения среды выполнения Flash, иерархическая обработка событий, реализация интерактивности с использованием клавиатуры и мыши, анимация, векторная и растровая графика, текст и операции загрузки содержимого.

Прочитав часть II, вы сможете добавлять графическое содержимое и интерактивные возможности в ваши собственные приложения.

- Глава 20 «API отображения и список отображения».
- Глава 21 «События и иерархии отображения».
- Глава 22 «Интерактивность».
- Глава 23 «Обновления экрана».
- Глава 24 «Программная анимация».
- Глава 25 «Рисование с помощью векторов».
- Глава 26 «Программирование растровой графики».
- Глава 27 «Отображение и ввод текста».
- Глава 28 «Загрузка внешних отображаемых элементов».

API отображения и список отображения

Одной из основных задач в программировании на языке ActionScript является отображение объектов на экране. В этой связи платформа Flash предлагает широкий набор инструментов для создания и управления графическим содержимым. Данные инструменты можно разделить на две общие категории.

- *API отображения* среды выполнения Flash — набор классов для работы с интерактивными графическими объектами, растровыми изображениями и векторным содержимым.
- Готовые компоненты пользовательского интерфейса:
  - *набор компонентов пользовательского интерфейса платформы разработки Flex* — развитый набор настраиваемых компонентов пользовательского интерфейса, построенный на базе API отображения;
  - *набор компонентов пользовательского интерфейса среды разработки Flash* — набор компонентов пользовательского интерфейса, благодаря которым уменьшается размер файла приложения и снижается потребление памяти, однако данные компоненты обладают более ограниченными возможностями по сравнению с набором компонентов пользовательского интерфейса платформы разработки Flex.

Интерфейс API отображения встроен непосредственно во все среды выполнения Flash и, как результат, доступен для всех SWF-файлов. Интерфейс API отображения предназначен для создания максимально настраиваемых пользовательских интерфейсов или визуальных эффектов наподобие тех, которые часто встречаются в мультипликационной графике и играх. В этой главе все внимание сосредоточено на интерфейсе API отображения.

Набор компонентов пользовательского интерфейса платформы разработки Flex является частью платформы разработки Flex — внешней библиотеки классов, поставляемой вместе с приложением Adobe Flex Builder. Эта библиотека также доступна для бесплатной загрузки по адресу http://www.adobe.com/go/flex2_sdk. Набор компонентов графического интерфейса платформы разработки Flex разработан для построения приложений с относительно стандартными элементами управления пользовательского интерфейса (полосами прокрутки, раскрывающимися меню, таблицами данных и т. д.). Элементы интерфейса платформы разработки Flex обычно используются в приложениях MXML, но могут включаться и в приложения, разработанные в основном с использованием языка ActionScript. Подробности использования платформы разработки Flex в приложениях на языке ActionScript можно найти в гл. 30.

Набор компонентов пользовательского интерфейса среды разработки Flash предназначен для использования в SWF-файлах, созданных в среде разработки Flash, а также, когда размер файла и низкое потребление памяти оказываются гораздо важнее расширенных возможностей компонента, например связывания данных и улучшенных возможностей стилизации. Набор компонентов пользовательского интерфейса среды разработки Flash и набор компонентов пользовательского интерфейса платформы разработки Flex используют очень похожий интерфейс API, что позволяет разработчикам применять полученные знания при переходе от одного набора компонентов к другому.

В приложении Flash Player 8 и более старых версиях язык ActionScript предоставлял следующие четыре основных строительных блока для создания и управления графическим содержимым.

- *Клип* — контейнер для графического содержимого, реализующий интерактивные возможности, простейшее рисование, иерархическое размещение элементов и анимационные возможности.
- *Текстовое поле* — прямоугольная область, содержащая отформатированный текст.
- *Кнопка* — элемент ввода, представляющий собой очень простую интерактивную кнопку.
- *Растровое изображение* (появилось в приложении Flash Player 8) — графическое изображение в растровом формате.

Перечисленные элементы остаются доступными в интерфейсе API отображения, однако представляющие их классы в языке ActionScript 3.0 (`MovieClip`, `TextField`, `SimpleButton` и `Bitmap`) были улучшены, переработаны и приобрели больше возможностей.

Обзор API отображения

В языке ActionScript все графическое содержимое создается и управляется с помощью классов интерфейса API отображения. Даже элементы интерфейса платформы разработки Flex и среды разработки Flash используют интерфейс API отображения в качестве графической основы. Многие классы API отображения непосредственно представляют конкретный тип графического содержимого, выводимого на экран. Например, класс `Bitmap` представляет растровые изображения, класс `Sprite` — интерактивную графику, а класс `TextField` — отформатированный текст.

При обсуждении мы будем называть классы, которые непосредственно представляют отображаемое на экране содержимое (и суперклассы этих классов), *базовыми классами отображения*. Остальные классы API отображения определяют вспомогательную графическую информацию и функциональность, но сами не представляют содержимое на экране. Например, классы `CapStyle` и `JointStyle` определяют константы, представляющие настройки для рисования линий, в то время как классы `Graphics` и `BitmapData` задают множество примитивных операций рисования. Мы будем называть такие классы, не отображающие информацию на

экране, *вспомогательными классами отображения*. Большинство базовых и вспомогательных классов API отображения находятся в пакете `flash.display`.

Базовые классы отображения, представленные на рис. 20.1, образуют иерархию, которая отражает три базовых уровня функциональности: отображение, пользовательскую интерактивность и содержание. Соответственно тремя важнейшими классами API отображения являются: `DisplayObject`, `InteractiveObject` и `DisplayObjectContainer`. Непосредственно создать экземпляры этих классов невозможно, однако они определяют абстрактную функциональность, которая реализуется различными конкретными подклассами.

Как уже говорилось в гл. 6, ActionScript 3.0 не поддерживает настоящие абстрактные классы. Поэтому на рис. 20.1 классы `DisplayObject`, `InteractiveObject` и `DisplayObjectContainer` показаны не как абстрактные классы, а как классы абстрактного типа. Тем не менее, несмотря на эту техническую деталь, для краткости в оставшейся части этой главы мы будет использовать термин «абстрактный» при описании архитектурной роли, которую играют классы `DisplayObject`, `InteractiveObject` и `DisplayObjectContainer`.

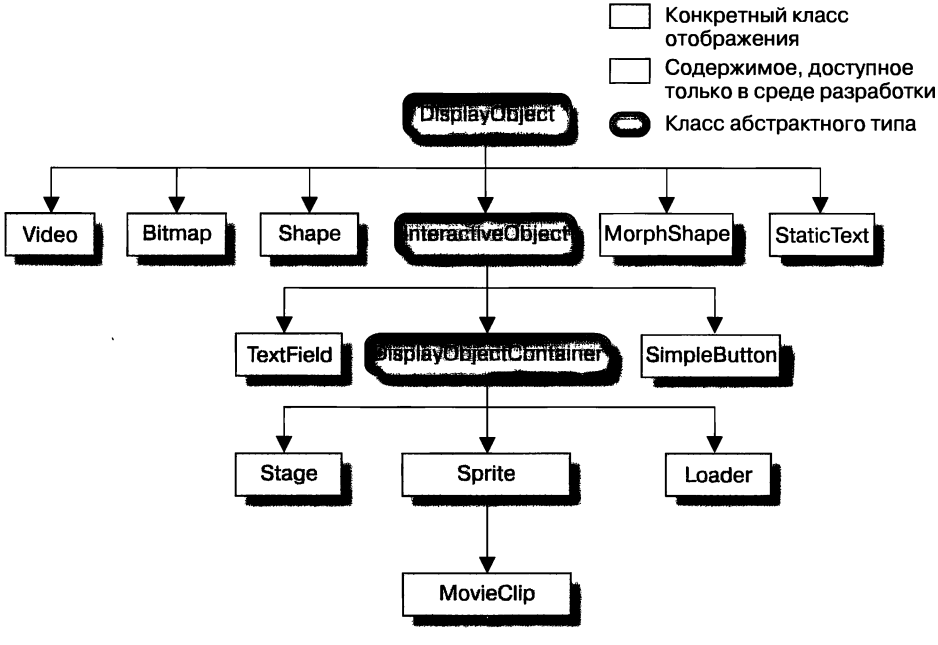

Рис. 20.1. Иерархия базовых классов отображения

Класс `DisplayObject` — корневой элемент в иерархии базовых классов отображения, он определяет первый уровень графической функциональности API отображения — вывод на экран. Все классы, унаследованные от `DisplayObject`, получают общий набор основных графических характеристик и возможностей. Например, для любого потомка класса `DisplayObject` можно изменить его позицию, размеры и угол поворота с помощью переменных `x`, `y`, `width`, `height` и `rotation`. Класс

DisplayObject — это не просто базовый класс; это источник многих расширенных возможностей в API отображения, включая следующие, но не ограничиваясь ими:

- преобразование координат (ознакомьтесь с описанием методов экземпляра localToGlobal( ) и globalToLocal( ) класса DisplayObject в справочнике по языку ActionScript корпорации Adobe);
- определение пересечений между объектами и точками (ознакомьтесь с описанием методов экземпляра hitTestObject( ) и hitTestPoint( ) класса DisplayObject в справочнике по языку ActionScript);
- применение фильтров, трансформаций и масок (ознакомьтесь с описанием переменных экземпляра filters, transform и mask класса DisplayObject в справочнике по языку ActionScript);
- непропорциональное изменение масштаба для «растягиваемых» графических элементов (ознакомьтесь с описанием переменной экземпляра scale9grid класса DisplayObject в справочнике по языку ActionScript корпорации Adobe).

Обратите внимание, что иногда в этой книге используется неофициальный термин «отображаемый объект», который обозначает любой экземпляр класса, унаследованного от класса DisplayObject.

Конкретные непосредственные подклассы класса DisplayObject — Video, Bitmap, Shape, MorphShape и StaticText — представляют простейший тип отображаемого содержимого: базовую графику, выводимую на экран, которая не может получать вводимые данные или содержать другое вложенное графическое содержимое. Класс Video представляет потоковое видео. Класс Bitmap визуализирует растровые изображения, созданные и управляемые вспомогательным классом BitmapData. Класс Shape предоставляет простой, облегченный холст для векторного рисования. Специальные классы MorphShape и StaticText представляют соответственно фигуры с преобразуемой формой и статический текст, созданные в среде разработки Flash. Ни класс MorphShape, ни класс StaticText не могут быть созданы на языке ActionScript.

Единственный абстрактный подкласс InteractiveObject класса DisplayObject образует второй уровень функциональности в API отображения — интерактивность. Все унаследованные от него классы получают возможность отвечать на события ввода от мыши и клавиатуры пользователя.

Конкретные непосредственные подклассы TextField и SimpleButton класса InteractiveObject представляют два различных типа интерактивного графического содержимого. Класс TextField представляет прямоугольную область для отображения отформатированного текста и получения вводимых пользователем данных. Класс SimpleButton представляет символы Button (Кнопка), создаваемые в среде разработки Flash, а также позволяет быстро создавать интерактивные кнопки из кода на языке ActionScript. Обрабатывая события ввода, передаваемые классами TextField или SimpleButton, программист может добавлять интерактивность в приложение. Например, экземпляр класса TextField может быть запрограммирован так, чтобы в ответ на событие FocusEvent.FOCUS_IN изменялся цвет фона этого экземпляра, а экземпляр класса SimpleButton может быть запрограммирован так, чтобы в ответ на событие MouseEvent.CLICK форма отправлялась на сервер.

Абстрактный подкласс `DisplayObjectContainer` класса `InteractiveObject` представляет основу третьего и последнего функционального уровня в API отображения — содержание. Все унаследованные от него классы получают возможность содержать любой другой экземпляр класса `DisplayObject`.

Контейнеры используются для объединения нескольких графических объектов, что позволяет обрабатывать их вместе. Всякий раз, когда контейнер перемещается, поворачивается или трансформируется, все содержащиеся в нем объекты наследуют это перемещение, вращение или трансформацию. Подобным образом всякий раз, когда контейнер удаляется с экрана, содержащиеся в нем объекты удаляются вместе с ним. Более того, контейнеры могут быть вложены внутрь других контейнеров для создания иерархических групп отображаемых объектов.

При описании объектов в иерархии отображения в этой книге применяется стандартная терминология для описания древовидных структур. Например, объект, который содержит другой объект в иерархии отображения, называется *родителем* этого объекта, а содержащийся объект называется *ребенком* этого родителя. В многоуровневой иерархии отображения объекты, расположенные выше данного объекта в иерархии, называются *предками* этого объекта. С другой стороны, объекты, расположенные ниже данного объекта в иерархии, называются *потомками* этого объекта. Наконец, объект самого верхнего уровня в иерархии (объект, потомками которого являются все остальные объекты) называется *корневым* объектом.

Не путайте объекты-предки и объекты-потомки в иерархии отображения с классами-предками и классами-потомками в иерархии наследования. Для ясности в этой книге иногда используются термины «предки отображения» и «потомки отображения» для описания объектов-предков и объектов-потомков в иерархии отображения.

Каждый из подклассов класса `DisplayObjectContainer` — `Sprite`, `MovieClip`, `Stage` и `Loader` — обеспечивает уникальный тип пустой содержащей структуры, ожидающей заполнения содержимым. Класс `Sprite` — это важнейший элемент среди классов-контейнеров. Являясь потомком классов `InteractiveObject` и `DisplayObjectContainer`, класс `Sprite` предоставляет отличную основу для построения собственных элементов пользовательского интерфейса с нуля. Класс `MovieClip` — это расширенный тип класса `Sprite`, представляющий анимированное содержимое, которое было создано в среде разработки Flash. Класс `Stage` представляет основную *область отображения* (видимую область внутри границ окна приложения) среды выполнения Flash. Наконец, класс `Loader` используется для загрузки внешнего графического содержимого из локального источника или через Интернет.

До появления языка ActionScript 3.0 класс MovieClip использовался как универсальный графический контейнер (подобно использованию класса Sprite в языке ActionScript 3.0). С появлением версии 3.0 ActionScript класс MovieClip применяется только для управления экземплярами символов клипов, созданных в среде разработки Flash. Поскольку язык ActionScript 3.0 не предоставляет способ для создания элементов временной шкалы, например кадров и фигур с изменяемой формой, нет необходимости создавать пустые клипы на этапе выполнения приложения, написанного на языке ActionScript 3.0. Вме-

сто этого любое графическое содержимое, создаваемое программным путем, должно являться экземпляром одного из подходящих базовых классов отображения (Bitmap, Shape, Sprite, TextField и т. д.).

Интерфейс API отображения предоставляет широкие возможности, которые доступны через сотни методов и переменных. Многие из них рассматриваются в этой книге. В следующих же разделах мы сосредоточимся не на систематическом рассмотрении каждого метода и переменной, а на фундаментальных концепциях. Описание всех методов и переменных API отображения можно найти в справочнике по языку ActionScript корпорации Adobe.

Расширение иерархии базовых классов отображения. Хотя в большинстве случаев базовые классы отображения могут эффективно использоваться без внесения каких-либо изменений, большинство нетривиальных программ расширяют функциональность базовых классов отображения, создавая подклассы для собственных целей. Например, программа, рисующая геометрические фигуры, может определить классы `Ellipse`, `Rectangle` и `Triangle`, расширяющие класс `Shape`. Подобным образом, в приложении для просмотра новостей может быть определен класс `Heading`, который расширяет класс `TextField`, а в гоночной игре может быть определен класс `Car`, который расширяет класс `Sprite`. На самом деле элементы пользовательского интерфейса в платформе разработки Flex являются потомками класса `Sprite`. В последующих главах мы увидим множество примеров пользовательских классов отображения. По мере изучения базовых классов отображения задумайтесь о том, как можно расширить их функциональность. Зачастую программисты на языке ActionScript расширяют и улучшают базовые классы отображения с помощью собственного кода. Дополнительную информацию можно найти далее, в разд. «Пользовательские графические классы».

Список отображения

Как уже говорилось, базовые классы отображения представляют типы графического содержимого, доступного в языке ActionScript. Чтобы создать реальную графику из этих теоретических типов, мы создаем экземпляры базовых классов отображения, а затем добавляем эти экземпляры в *список отображения*. Список отображения — это иерархия всех графических объектов, отображаемых средой выполнения Flash в настоящий момент. Когда отображаемый объект добавляется в список отображения и помещается в видимую область, среда Flash визуализирует содержимое этого объекта на экране.

Корневым элементом списка отображения является экземпляр класса `Stage`, который создается автоматически при запуске среды выполнения Flash. Этот особый, автоматически создаваемый экземпляр класса `Stage` выполняет две задачи. Во-первых, он выступает в качестве внешнего контейнера для всего графического содержимого, отображаемого средой выполнения Flash (то есть является корнем списка отображения). Во-вторых, он предоставляет информацию о глобальных характеристиках области отображения и позволяет управлять ими. Например, переменная экземпляра `quality` класса `Stage` определяет качество визуализации всей

отображаемой графики; переменная `scaleMode` задает способ масштабирования графики при изменении размеров области отображения; переменная `frameRate` определяет текущую предпочтительную скорость (кадров в секунду) для всех анимационных роликов. Как будет рассказано в этой главе, обращение к экземпляру класса `Stage` всегда осуществляется через некоторый объект в списке отображения с помощью переменной экземпляра `stage` класса `DisplayObject`. Например, если `output_txt` — экземпляр класса `TextField`, который в настоящий момент находится в списке отображения, обратиться к экземпляру класса `Stage` можно выражением `output_txt.stage`.

До появления языка ActionScript 3.0 класс `Stage` не содержал объекты из списка отображения. Более того, ко всем его методам и переменным можно было обращаться непосредственно через сам класс, как показано в следующем коде:

```
trace(Stage.align);
```

В языке ActionScript 3.0 к методам и переменным класса `Stage` нельзя обратиться через класс `Stage`, кроме того, не существует глобальной точки, ссылающейся на экземпляр класса `Stage`. В языке ActionScript 3.0 приведенная строка кода вызовет следующую ошибку:

```
Access of possibly undefined property 'align' through a reference with static
type 'Class'
```

На русском языке она будет звучать так: Обращение к возможно неопределенному свойству 'align' через ссылку на статический класс 'Class'.

Чтобы избежать появления этой ошибки, обращайтесь к экземпляру класса `Stage` с помощью следующего подхода:

```
trace(некийОтображаемыйОбъект.stage.align);
```

Здесь *некийОтображаемыйОбъект* — это объект, который в настоящий момент находится в списке отображения. Архитектура класса `Stage` в языке ActionScript 3.0 позволяет в будущем иметь несколько экземпляров класса `Stage`, а также вносит вклад в систему безопасности приложения Flash Player (поскольку неавторизованные объекты, загруженные из внешних источников, не имеют глобальной точки для обращения к экземпляру класса `Stage`).

На рис. 20.2 представлено состояние списка отображения для пустой среды выполнения Flash до момента открытия SWF-файла. Левая часть рисунка демонстрирует символическое представление среды выполнения Flash, а правая часть — соответствующую иерархию списка отображения. Когда среда Flash пуста, иерархия списка отображения содержит всего один элемент (единственный экземпляр класса `Stage`). Но скоро появятся другие элементы!

Когда пустая среда Flash открывает новый SWF-файл, она находит основной класс этого SWF-файла, создает экземпляр данного класса и добавляет созданный экземпляр в список отображения в качестве первого ребенка экземпляра класса `Stage`.

Напомним, что основной класс SWF-файла должен быть унаследован либо от класса Sprite, либо от класса MovieClip, каждый из которых является потомком класса DisplayObject. Методики определения основного класса SWF-файла рассматриваются в гл. 7.

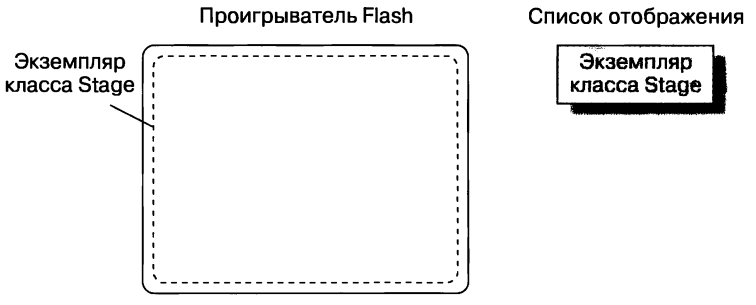

Рис. 20.2. Список отображения для пустой среды выполнения Flash

Экземпляр основного класса SWF-файла является точкой входа программы и первым графическим объектом, отображаемым на экране. Даже если экземпляр основного класса не создает никакую графику, он все равно добавляется в список отображения, готовый содержать любую графику, создаваемую приложением в будущем. Экземпляр основного класса первого SWF-файла, открываемого средой выполнения Flash, играет особую роль в языке ActionScript; он определяет конкретные глобальные настройки среды, например разрешение относительных URL-адресов и тип ограничений безопасности, применяемых к операциям с внешними источниками.

В честь особой роли, которую играет экземпляр основного класса первого SWF-файла, открываемого средой выполнения Flash, он иногда называется владельцем сцены.

Рассмотрим пример, который демонстрирует, как создается владелец сцены. Предположим, мы запустили автономную версию приложения Flash Player и открыли файл с именем `GreetingApp.swf`, основным классом которого является `GreetingApp`. Если файл `GreetingApp.swf` содержит только этот класс и класс `GreetingApp` не создает никакой графики, то список отображения приложения Flash Player будет включать всего два элемента: экземпляр класса `Stage` и экземпляр класса `GreetingApp` (содержащийся в экземпляре класса `Stage`). Это показано на рис. 20.3.

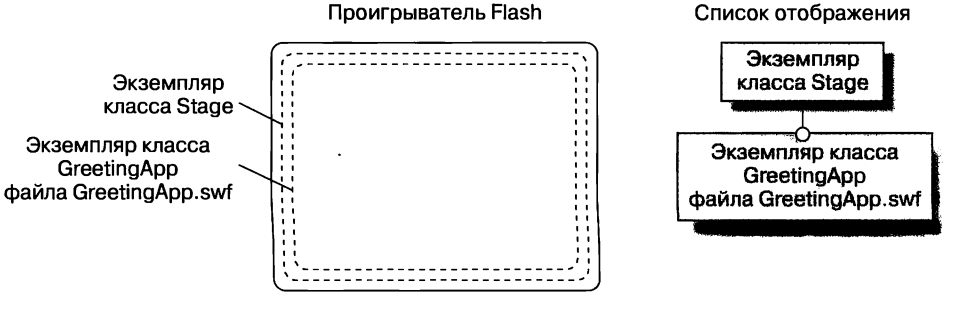

Рис. 20.3. Список отображения для файла GreetingApp.swf

Как только экземпляр основного класса SWF-файла будет добавлен в экземпляр класса `Stage`, программа сможет добавлять новое содержимое на экран в такой последовательности.

1. Создать отображаемый объект (то есть экземпляр любого базового класса отображения или любого класса, который расширяет базовый класс отображения).
2. Вызвать метод экземпляра `addChild( )` класса `DisplayObjectContainer` либо над экземпляром класса `Stage`, либо над экземпляром основного класса и передать в метод `addChild( )` отображаемый объект, созданный на шаге 1.

Попробуем выполнить вышеописанные общие шаги, создав класс `GreetingApp`, а затем добавив прямоугольник, круг и текстовое поле в список отображения с помощью метода `addChild( )`. Сначала приведем основу класса `GreetingApp`:

```
package {
  import flash.display.*;
  import flash.text.TextField;

  public class GreetingApp extends Sprite {
    public function GreetingApp ( ) {
    }
  }
}
```

В нашем классе `GreetingApp` будут использованы классы `Shape` и `Sprite`, поэтому мы импортируем целиком весь пакет `flash.display`, который содержит эти классы. Подобным образом в классе `GreetingApp` будет использован класс `TextField`, поэтому мы импортируем пакет `flash.text.TextField`.

Обратите внимание на то, что класс `GreetingApp` по необходимости расширяет класс `Sprite`. `GreetingApp` должен расширять либо класс `Sprite`, либо класс `MovieClip`, поскольку он является основным классом программы.

В языке ActionScript 3.0 основной класс SWF-файла должен расширять либо класс Sprite, либо класс MovieClip, либо их любой подкласс.

Когда основной класс представляет корневую временную шкалу FLA-файла, он должен расширять класс `MovieClip`. Во всех остальных случаях основной класс должен расширять класс `Sprite`. В нашем примере класс `GreetingApp` расширяет класс `Sprite`, поскольку он не ассоциирован с FLA-файлом. В результате компиляции этого класса должно получиться автономное приложение ActionScript.

Теперь создадим прямоугольник и круг в методе-конструкторе класса `GreetingApp`. И прямоугольник, и круг мы нарисуем внутри одного объекта `Shape`. Объекты `Shape` (и все графические объекты) создаются с помощью оператора `new`, как и объекты любого другого типа. Вот код, который используется для создания нового объекта `Shape`:

```
new Shape( )
```

Безусловно, в дальнейшем нам придется обращаться к этому объекту, чтобы нарисовать в нем элементы, поэтому присвоим его переменной `rectAndCircle`:

```
var rectAndCircle:Shape = new Shape( );
```

Для рисования векторов в языке ActionScript применяется вспомогательный класс отображения `Graphics`. Каждый объект `Shape` хранит свой собственный экземп-

ляр класса Graphics в переменной экземпляра graphics. Таким образом, чтобы нарисовать прямоугольник и круг внутри нашего объекта Shape, мы вызываем соответствующие методы над объектом rectAndCircle.graphics. Вот этот код:

```
// Задаем толщину линии, равную одному пикселу
rectAndCircle.graphics.lineStyle(1);

// Рисуем синий прямоугольник
rectAndCircle.graphics.beginFill(0x0000FF, 1);
rectAndCircle.graphics.drawRect(125, 0, 150, 75);

// Рисуем красный круг
rectAndCircle.graphics.beginFill(0xFF0000, 1);
rectAndCircle.graphics.drawCircle(50, 100, 50);
```

Дополнительную информацию о векторном рисовании в языке ActionScript 3.0 можно найти в гл. 25.

Операции векторного рисования не ограничиваются классом Shape. Класс Sprite также предоставляет ссылку на объект Graphics через свою переменную экземпляра graphics, поэтому, чтобы нарисовать прямоугольник и круг, вместо объекта Shape мы могли бы создать объект Sprite. Тем не менее, поскольку для хранения объекта Sprite требуется больше памяти, чем для объекта Shape, лучше использовать объект Shape при создании векторной графики, не содержащей дочерних объектов и для которой не требуется интерактивности.

Строго говоря, если бы мы хотели добиться минимального расходования памяти в примере приложения GreetingApp, мы бы рисовали наши фигуры непосредственно внутри экземпляра класса GreetingApp (помните, что класс GreetingApp расширяет класс Sprite, поэтому он поддерживает векторное рисование). Наш код выглядел бы следующим образом:

```
package {
  import flash.display.*;
  public class GreetingApp extends Sprite {
    public function GreetingApp ( ) {
      graphics.lineStyle(1);

      // Прямоугольник
      graphics.beginFill(0x0000FF, 1);
      graphics.drawRect(125, 0, 150, 75);

      // Круг
      graphics.beginFill(0xFF0000, 1);
      graphics.drawCircle(50, 100, 50);
    }
  }
}
```

Этот код успешно рисует прямоугольник и круг на экране, но данный подход менее гибок, чем подход с помещением фигур в отдельный объект Shape. Помещение

рисунков в объект Shape позволяет переносить их, размещать по слоям, модифицировать и удалять независимо от остального графического содержимого в приложении. Например, возвращаясь к нашему предыдущему подходу с рисованием в экземпляре класса Shape (rectAndCircle), для перемещения фигур в новую позицию мы могли бы использовать следующий код:

```
// Перемещаем объект rectAndCircle вправо на 125 пикселов
// и вниз на 100 пискелов
rectAndCircle.x = 125;
rectAndCircle.y = 100;
```

Отметим, что в настоящий момент у нас есть отображаемый объект rectAndCircle, который пока не добавлен в список отображения. Обращение и управление отображаемыми объектами, которые не добавлены в список отображения, является допустимой и распространенной практикой. Отображаемые объекты в любой момент жизненного цикла программы могут быть добавлены в список отображения или удалены из него, а программное взаимодействие с этими объектами может осуществляться независимо от того, находятся они в списке отображения или нет.

Обратите внимание, что предыдущее позиционирование объекта rectAndCircle осуществляется до того, как этот объект будет помещен в список отображения! Каждый отображаемый объект сохраняет свое собственное состояние, независимо от родителя, с которым он связан, — фактически независимо от того, добавлен объект в список отображения или нет. Когда объект rectAndCircle будет добавлен в контейнер отображения, он будет автоматически помещен в позицию (125; 100) координатного пространства этого контейнера. Если в дальнейшем объект rectAndCircle будет удален из этого контейнера и добавлен в другой контейнер, он будет помещен в позицию (125; 100) координатного пространства нового контейнера.

Любой отображаемый объект сохраняет свои характеристики при перемещении из одного контейнера в другой и даже при удалении из списка отображения.

Теперь наступает момент, которого мы с нетерпением ожидали. Чтобы отобразить фигуры на экране, мы вызываем метод addChild( ) над экземпляром класса GreetingApp внутри конструктора GreetingApp и передаем в этот метод ссылку на экземпляр класса Shape, хранящуюся в переменной rectAndCircle.

```
// Отображаем объект rectAndCircle на экране, добавляя его
// в список отображения
addChild(rectAndCircle);
```

В результате приложение Flash Player добавляет объект rectAndCircle в список отображения в качестве ребенка экземпляра класса GreetingApp.

Как подкласс класса Sprite, класс GreetingApp является потомком класса DisplayObjectContainer и, следовательно, наследует метод addChild( ) и возможность содержать детей. Чтобы вспомнить иерархию классов API отображения, вернитесь к рис. 20.1.

Отображение объектов на экране — это забавно! Проделаем это еще раз. Добавление следующего кода в конструктор класса `GreetingApp` приведет к отображению на экране текста «Hello world»:

```
// Создаем объект TextField, который будет содержать некий текст
var greeting_txt:TextField = new TextField( );

// Указываем отображаемый текст
greeting_txt.text = "Hello world";

// Задаем позицию объекта TextField
greeting_txt.x = 200;
greeting_txt.y = 300;

// Отображаем текст на экране, добавляя объект greeting_txt
// в список отображения
addChild(greeting_txt);
```

Когда объект будет добавлен в контейнер отображения, обратиться к этому контейнеру можно будет через переменную экземпляра `parent` класса `DisplayObject`. Например, в конструкторе класса `GreetingApp` следующее выражение является допустимой ссылкой на экземпляр класса `GreetingApp`:

```
greeting_txt.parent
```

Если отображаемый объект еще не был добавлен в список отображения, его переменная `parent` имеет значение `null`.

Листинг 20.1 демонстрирует код приложения `GreetingApp` целиком.

Листинг 20.1. Графическое приложение «Hello world»

```
package {
  import flash.display.*;
  import flash.text.TextField;

  public class GreetingApp extends Sprite {
    public function GreetingApp( ) {
      // Создаем объект Shape
      var rectAndCircle:Shape = new Shape( );

      // Задаем толщину линии один пиксел
      rectAndCircle.graphics.lineStyle(1);

      // Рисуем синий прямоугольник
      rectAndCircle.graphics.beginFill(0x0000FF, 1);
      rectAndCircle.graphics.drawRect(125, 0, 150, 75);

      // Рисуем красный круг
      rectAndCircle.graphics.beginFill(0xFF0000, 1);
      rectAndCircle.graphics.drawCircle(50, 100, 50);

      // Перемещаем фигуры вправо на 125 пикселов и вниз на 100 пискелов
      rectAndCircle.x = 125;
      rectAndCircle.y = 100;
```

```
        // Выводим объект rectAndCircle на экране, добавляя его
        // в список отображения
        addChild(rectAndCircle);

        // Создаем объект  TextField, который будет содержать некий текст
        var greeting_txt:TextField = new TextField( );

        // Указываем отображаемый текст
        greeting_txt.text = "Hello world";

        // Задаем позицию текста
        greeting_txt.x = 200;
        greeting_txt.y = 300;

        // Выводим текст на экране, добавляя объект greeting_txt
        // в список отображения
        addChild(greeting_txt);
    }
  }
}
```

На рис. 20.4 в графическом виде показаны результаты выполнения кода из листинга 20.1. Как и на двух предыдущих рисунках, графика, отображаемая на экране, показана слева, а соответствующая иерархия списка отображения приложения Flash Player — справа.

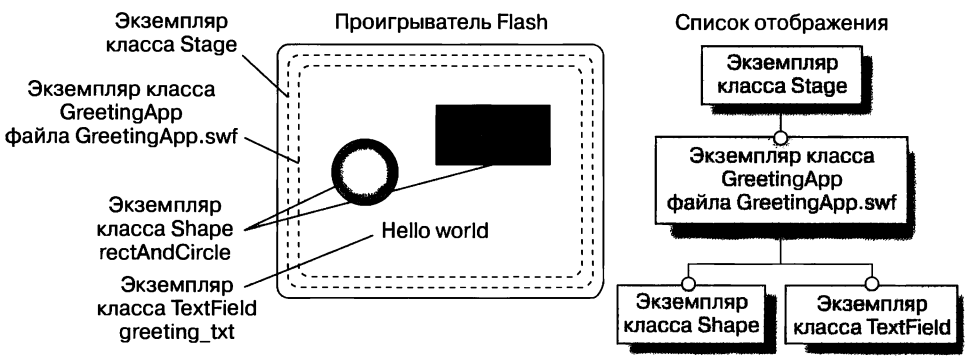

Рис. 20.4. Список отображения для приложения GreetingApp

Контейнеры и глубины

В предыдущем разделе мы добавили в приложение `GreetingApp` два отображаемых ребенка (`rectAndCircle` и `greeting_txt`). На экране они были размещены так, чтобы не перекрывать друг друга. Если бы они перекрывались, один объект закрывал бы другой в соответствии с *глубинами* этих двух объектов.

Глубина отображаемого объекта — это целочисленное значение, которое определяет, как данный объект перекрывает другие объекты в одном и том же контейнере

отображаемых объектов. Когда два объекта перекрываются, тот объект, у которого позиция глубины больше («высший» из двух), закрывает другой объект («низший» из двух). Следовательно, можно считать, что все отображаемые объекты в контейнере размещаются по порядку в визуальном стеке наподобие колоды уложенных в стопку игральных карт, счет которых начинается с нуля. Самый нижний объект в стеке имеет позицию глубины, равную 0, а самый верхний — позицию глубины, равную количеству объектов-детей в контейнере отображаемых объектов минус один (по аналогии самая нижняя карта в колоде имеет позицию глубины, равную 0, а самая верхняя — позицию глубины, равную количеству карт в колоде минус один).

В API управления глубиной отображаемых объектов языка ActionScript 2.0 допускалось существование «незаполненных» глубин. Например, в контейнере, содержащем всего два объекта, первый объект мог иметь глубину, равную 0, а другой объект — глубину, равную 40, оставляя незаполненными глубины от 1 до 39. В API управления глубиной отображаемых объектов языка ActionScript 3.0 появление незаполненных глубин не допускается и вообще невозможно.

Отображаемым объектам, добавляемым в контейнер с помощью метода `addChild( )`, позиции глубины присваиваются автоматически. Если взять пустой контейнер, то первый ребенок, добавленный через метод `addChild( )`, помещается на глубину 0, второй — на глубину 1, третий — на глубину 2 и т. д. Таким образом, объект, добавленный последним через метод `addChild( )`, всегда отображается поверх остальных детей.

В качестве примера продолжим работу с программой `GreetingApp` из предыдущего раздела. На этот раз мы нарисуем круг и прямоугольник в отдельных экземплярах класса `Shape`, чтобы их можно было размещать в стеке независимо друг от друга. Кроме того, изменим позиции круга, прямоугольника и текста таким образом, чтобы они перекрывались. Рассмотрим измененный код (этот код и другие примеры данного раздела взяты из метода-конструктора класса `GreetingApp`):

```
// Прямоугольник
var rect:Shape = new Shape( );
rect.graphics.lineStyle(1);
rect.graphics.beginFill(0x0000FF, 1);
rect.graphics.drawRect(0, 0, 75, 50);

// Круг
var circle:Shape = new Shape( );
circle.graphics.lineStyle(1);
circle.graphics.beginFill(0xFF0000, 1);
circle.graphics.drawCircle(0, 0, 25);
circle.x = 75;
circle.y = 35;

// Текстовое сообщение
var greeting_txt:TextField = new TextField( );
greeting_txt.text = "Hello world";
greeting_txt.x = 60;
greeting_txt.y = 25;
```

Теперь попытаемся добавить прямоугольник, круг и текст в качестве детей экземпляра класса `GreetingApp`, используя различные последовательности. Следующий код сначала добавляет прямоугольник, затем круг и после него — текст:

```
addChild(rect);          // Глубина 0
addChild(circle);        // Глубина 1
addChild(greeting_txt);  // Глубина 2
```

Как показано на рис. 20.5, прямоугольник был добавлен первым, поэтому он отображается под кругом и текстом. Затем был добавлен круг, поэтому он отображается поверх прямоугольника, но под текстом. Текст был добавлен последним, поэтому он отображается поверх прямоугольника и круга.

Следующий код изменяет последовательность, добавляя сначала круг, затем прямоугольник и после этого — текст. На рис. 20.6 показан результат. Обратите внимание, как простое изменение последовательности, с учетом которой добавляются объекты, влияет на конечный результат.

```
addChild(circle);        // Глубина 0
addChild(rect);          // Глубина 1
addChild(greeting_txt);  // Глубина 2
```

Рассмотрим еще один пример. Следующий код сначала добавляет текст, затем окружность и после нее — прямоугольник. На рис. 20.7 показан результат.

```
addChild(greeting_txt);  // Глубина 0
addChild(circle);        // Глубина 1
addChild(rect);          // Глубина 2
```

Рис. 20.5. Прямоугольник, круг, текст

Рис. 20.6. Круг, прямоугольник, текст

Рис. 20.7. Текст, круг, прямоугольник

Для того чтобы получить позицию глубины любого объекта в контейнере отображаемых объектов, нужно использовать метод экземпляра `getChildIndex( )` класса `DisplayObjectContainer`:

```
trace(getChildIndex(rect)); // Выводит: 2
```

Чтобы добавить новый объект в определенную позицию глубины, используется метод экземпляра `addChildAt( )` класса `DisplayObjectContainer` (обратите внимание: `addChildAt( )`, а не `addChild( )`). Этот метод принимает следующий вид:

контейнер`.addChildAt(`*отображаемыйОбъект*`,` *позицияГлубины*`)`

Значение параметра *позицияГлубины* должно находиться в пределах от 0 до значения *контейнер*`.numChildren` включительно.

Если указанная позиция *позицияГлубины* уже занята существующим ребенком, объект *отображаемыйОбъект* будет помещен позади этого существующего объекта (то есть позиции глубины всех отображаемых объектов, которые равны указанному значе-

нию или выше его, увеличиваются на единицу, чтобы освободить место для нового ребенка).

Повторяйте про себя это мнемоническое правило метода addChildAt( ): «Если глубина занята, новый ребенок помещается позади».

Для добавления нового объекта поверх всех существующих детей используется следующий код:

```
контейнер.addChildAt(отображаемыйОбъект, контейнер.numChildren)
```

что аналогично следующей записи:

```
контейнер.addChild(отображаемыйОбъект)
```

Обычно метод `addChildAt( )` применяется совместно с методом экземпляра `getChildIndex( )` класса `DisplayObjectContainer` для добавления объекта ниже существующего ребенка в заданном контейнере. Вот общий формат:

```
контейнер.addChildAt(новыйРебенок,
                     контейнер.getChildIndex(существующийРебенок))
```

Испытаем этот подход, добавив новый треугольник позади круга в приложении `GreetingApp`, используя самое последнее воплощение данного приложения, изображенное на рис. 20.7.

Рассмотрим код, который создает треугольник:

```
var triangle:Shape = new Shape( );
triangle.graphics.lineStyle(1);
triangle.graphics.beginFill(0x00FF00, 1);
triangle.graphics.moveTo(25, 0);
triangle.graphics.lineTo(50, 25);
triangle.graphics.lineTo(0, 25);
triangle.graphics.lineTo(25, 0);
triangle.graphics.endFill( );
triangle.x = 25;
triangle.y = 10;
```

Теперь рассмотрим код, который делает объект `triangle` новым ребенком экземпляра класса `GreetingApp`, размещая его ниже существующего объекта `circle` (обратите внимание, что методы `addChildAt( )` и `getChildIndex( )` неявно вызываются над текущим объектом `GreetingApp`). На рис. 20.8 показаны результаты.

```
addChildAt(triangle, getChildIndex(circle));
```

Рис. 20.8. Новый ребенок-треугольник

Как недавно было сказано, когда новый объект добавляется в позицию глубины, занятую существующим ребенком, позиции глубины существующего ребенка и всех

детей, расположенных выше его, увеличиваются на единицу. Новый объект затем занимает позицию глубины, которая была освобождена существующим ребенком. Например, перед добавлением объекта `triangle` глубины детей экземпляра класса `GreetingApp` выглядели следующим образом:

```
greeting_txt          0
circle                1
rect                  2
```

После добавления объекта `triangle` позиция глубины объекта `circle` изменилась с 1 на 2, позиция глубины объекта `rect` — с 2 на 3, а объект `triangle` занял глубину 1 (предыдущая глубина объекта `circle`). Между тем позиция глубины объекта `greeting_txt` не изменилась, поскольку с самого начала она была меньше глубины объекта `circle`. Вот измененные глубины после добавления объекта `triangle`:

```
greeting_txt          0
triangle              1
circle                2
rect                  3
```

Чтобы изменить глубину существующего ребенка, мы можем поменять местами позиции глубины этого ребенка и другого существующего ребенка с помощью методов экземпляра `swapChildren( )` или `swapChildrenAt( )` класса `DisplayObjectContainer`. Можно также непосредственно задать глубину для этого ребенка, используя метод экземпляра `setChildIndex( )` класса `DisplayObjectContainer`.

Метод `swapChildren( )` принимает следующий вид:

контейнер`.swapChildren(`*существующийРебенок1*`, `*существующийРебенок2*`);`

Здесь *существующийРебенок1* и *существующийРебенок2* — дочерние объекты контейнера *контейнер*. Метод `swapChildren( )` меняет местами глубины объектов *существующийРебенок1* и *существующийРебенок2*. На обычном языке приведенный код означает следующее: «Поместить объект *существующийРебенок1* на глубину, занимаемую объектом *существующийРебенок2*, и поместить объект *существующийРебенок2* на глубину, занимаемую объектом *существующийРебенок1*».

Метод `swapChildrenAt( )` принимает следующий вид:

контейнер`.swapChildrenAt(`*существующаяГлубина1*`, `*существующаяГлубина2*`);`

Здесь *существующаяГлубина1* и *существующаяГлубина2* — глубины, занимаемые детьми контейнера *контейнер*. Метод `swapChildrenAt( )` меняет местами глубины детей, находящихся на глубинах *существующаяГлубина1* и *существующаяГлубина2*. На обычном языке предыдущий код означает следующее: «Поместить ребенка, который в настоящий момент находится на глубине *существующаяГлубина1*, на глубину *существующаяГлубина2*, а ребенка, который в настоящий момент находится на глубине *существующаяГлубина2*, поместить на глубину *существующаяГлубина1*».

Метод `setChildIndex( )` принимает следующий вид:

контейнер`.setChildIndex(`*существующийРебенок*`, `*новаяПозицияГлубины*`);`

Здесь *существующийРебенок* — ребенок контейнера *контейнер*. Позиция *новаяПозицияГлубины* должна быть позицией глубины, в настоящий момент занимаемой объектом-ребенком контейнера *контейнер*. Иными словами, метод `setChildIndex( )` позволяет только изменять позиции существующих объектов-детей, но не позволяет добавлять новые позиции глубины. Значение параметра *новаяПозицияГлубины* метода `setChildIndex( )` обычно получают вызовом метода `getChildIndex( )` над существующим ребенком, как показано в следующем коде:

```
контейнер.setChildIndex(существующийРебенок1,
                        контейнер.getChildIndex(существующийРебенок2));
```

Это означает: «Поместить ребенка *существующийРебенок1* на глубину, которая в настоящий момент занята ребенком *существующийРебенок2*».

Стоит отметить, что, когда глубина объекта увеличивается на новую позицию с помощью метода `setChildIndex( )` (то есть объект перемещается выше), глубина всех объектов, находящихся между старой и новой позициями, уменьшается на 1, тем самым заполняя освободившуюся позицию в результате перемещения объекта. Таким образом, перемещенный объект появляется перед объектом, который раньше находился в указанной новой позиции. Например, продолжая работу над последней версией приложения `GreetingApp` (которая была показана на рис. 20.8), изменим позицию глубины объекта `greeting_txt` с 0 на 2. До выполнения следующего кода позицию глубины 2 занимает объект `circle`.

```
setChildIndex(greeting_txt, getChildIndex(circle));
```

Когда объект `greeting_txt` перемещается в позицию глубины 2, позиции глубины объектов `circle` и `triangle` уменьшаются до 1 и 0 соответственно, поэтому объект `greeting_txt` отображается перед этими объектами. Результаты показаны на рис. 20.9.

Рис. 20.9. Перемещение текста вверх

В отличие от этого, когда глубина объекта уменьшается на новую позицию с помощью метода `setChildIndex( )` (то есть объект перемещается ниже), позиция глубины всех объектов, находящихся в новой позиции или выше ее, увеличивается на 1, тем самым освобождая пространство для нового объекта. Таким образом, перемещенный объект появляется позади объекта, который раньше находился в указанной новой позиции (как если бы объект был добавлен с помощью метода `addChildAt( )`). Обратите внимание на важное различие между перемещением объекта на высокую глубину и перемещением объекта на низкую глубину.

Объект, перемещенный на большую глубину, отображается перед объектом, который находился в целевой позиции, однако объект, перемещенный на меньшую глубину, отображается позади объекта, который находился в целевой позиции.

Например, продолжая работать над кодом, результаты выполнения которого показаны на рис. 20.9, изменим позицию глубины объекта `rect` с 3 на 1 (1 — это глубина, в настоящий момент занимаемая объектом `circle`):

```
setChildIndex(rect, getChildIndex(circle));
```

Когда объект `rect` перемещается в позицию глубины 1, позиции глубины объектов `circle` и `greeting_txt` увеличиваются до 2 и 3 соответственно, поэтому объект `rect` отображается позади этих объектов (как показано на рис. 20.10).

Чтобы поместить объект поверх всех объектов в данном контейнере, используйте следующий код:

```
контейнер.setChildIndex(существующийРебенок, контейнер.numChildren-1)
```

Например, следующий код помещает треугольник поверх всех дочерних объектов экземпляра класса `GreetingApp` (код используется в классе `GreetingApp`, поэтому объект *контейнер* в предыдущем выражении опускается и неявно преобразуется в `this` — текущий объект):

```
setChildIndex(triangle, numChildren-1);
```

На рис. 20.11 показаны результаты выполнения кода.

Рис. 20.10. Перемещение прямоугольника вниз

Рис. 20.11. Треугольник перемещен вперед

Вы сможете легко разобраться с поведением метода `setChildIndex( )`, если представите детей объекта `DisplayObjectContainer` в виде колоды карт, как уже говорилось ранее. Если вы перемещаете нижнюю карту колоды вверх, остальные карты смещаются вниз (то есть карта, которая находилась поверх нижней карты, сама становится новой нижней картой). Если вы перемещаете верхнюю карту колоды вниз, остальные карты смещаются вверх (то есть карта, которая была нижней, теперь будет находиться поверх новой нижней карты).

Удаление элементов из контейнеров

Для удаления объекта из контейнера объектов отображения используется метод экземпляра `removeChild( )` класса `DisplayObjectContainer`, который принимает следующий вид:

```
контейнер.removeChild(существующийРебенок)
```

Здесь *контейнер* — это контейнер, который в настоящий момент содержит объект *существующийРебенок*. Например, чтобы удалить треугольник из экземпляра класса `GreetingApp`, мы бы использовали следующий код:

```
removeChild(triangle);
```

В качестве альтернативы мы можем удалить ребенка, используя его глубину, с помощью метода `removeChildAt( )`, который принимает следующий вид:

```
контейнер.removeChildAt(глубина)
```

После выполнения метода `removeChild( )` или `removeChildAt( )` переменной `parent` удаленного объекта присваивается значение `null`, поскольку удаленный ребенок больше не имеет контейнера. Если удаленный ребенок находился в списке отображения перед вызовом метода `removeChild( )` или `removeChildAt( )`, он удаляется из этого списка. Если удаленный ребенок отображался на экране перед вызовом метода `removeChild( )` или `removeChildAt( )`, он также удаляется с экрана. Если удаленный ребенок является объектом `DisplayObjectContainer` со своими детьми, его дети будут также удалены с экрана.

Удаление элементов из памяти

Важно отметить, что методы `removeChild( )` и `removeChildAt( )`, рассмотренные в предыдущем разделе, не обязательно приводят к уничтожению удаленного объекта в памяти; они только удаляют объект из иерархии отображения родительского объекта `DisplayObjectContainer`. Если на удаленный объект ссылается переменная или элемент массива, то этот объект продолжает существовать и может быть в дальнейшем заново добавлен в другой контейнер. Например, рассмотрим следующий код, который создает объект `Shape`, присваивает его переменной `rect` и затем добавляет этот объект в иерархию отображения объекта `parent`:

```
var rect:Shape = new Shape( );
rect.graphics.lineStyle(1);
rect.graphics.beginFill(0x0000FF, 1);
rect.graphics.drawRect(0, 0, 75, 50);
parent.addChild(rect);
```

Если теперь мы воспользуемся методом `removeChild( )`, чтобы удалить объект `Shape` из объекта `parent`, переменная `rect` будет по-прежнему ссылаться на объект `Shape`:

```
parent.removeChild(rect);
trace(rect); // Выводит: [object Shape]
```

До тех пор пока переменная `rect` существует, мы можем использовать ее для повторного добавления объекта `Shape` в иерархию отображения объекта `parent`, как показано в следующем коде:

```
parent.addChild(rect);
```

Чтобы полностью удалить отображаемый объект из программы, мы должны удалить не только его с экрана с помощью метода `removeChild( )`, но и все ссылки на него. Чтобы удалить все ссылки на объект, мы должны вручную удалить его из всех массивов, содержащих этот объект, и присвоить значение `null` (или любое другое значение) всем переменным, которые ссылаются на него. Как только все ссылки на объект будут удалены, он становится доступным для сборки мусора и в определенный момент будет удален из памяти сборщиком мусора языка ActionScript.

Однако, как уже говорилось в гл. 14, даже после того, как все ссылки на объект будут удалены, он продолжает быть активным до тех пор, пока сборщик мусора не удалит его из памяти. Например, если объект зарегистрировал приемники для события `Event.ENTER_FRAME`, то это событие будет по-прежнему приводить к выполнению кода. Подобным образом, если объект запустил таймеры, используя метод

`setInterval( )` или класс `Timer`, эти таймеры будут по-прежнему приводить к выполнению кода. Точно так же, если объект является экземпляром класса `MovieClip`, воспроизводимым в настоящий момент, его головка воспроизведения будет продолжать перемещаться, приводя к выполнению существующих сценариев кадров.

Пока объект ожидает начала процесса сборки мусора, приемники событий, таймеры и сценарии кадров могут вызывать незапланированное выполнение кода, приводя к расходованию памяти или к нежелательным побочным эффектам.

Чтобы избежать нежелательного выполнения кода при удалении отображаемого объекта из программы, убедитесь, что вы полностью деактивировали этот объект перед тем, как освободить все ссылки на него. Дополнительную информацию о деактивации объектов можно найти в гл. 14.

Всегда деактивируйте отображаемые объекты перед их уничтожением.

Удаление всех детей

Язык ActionScript не предоставляет прямого способа для удаления всех детей объекта. Следовательно, чтобы удалить всех отображаемых детей из конкретного объекта, мы должны использовать циклы `while` или `for`. Например, следующий код использует цикл `while` для удаления всех детей объекта *родитель* в направлении снизу вверх. Сначала удаляется ребенок на глубине 0, затем глубина всех детей уменьшается на 1, после этого удаляется новый ребенок на глубине 0, и этот процесс повторяется до тех пор, пока у объекта не останется детей.

```
// Удаляем всех детей объекта родитель
while (родитель.numChildren > 0) {
  родитель.removeChildAt(0);
}
```

Следующий код также удаляет всех детей объекта *родитель*, но в направлении сверху вниз. Однако использования такого кода следует избегать, поскольку он медленнее предыдущего, применяемого для удаления детей в направлении снизу вверх.

```
while (родитель.numChildren > 0) {
  родитель.removeChildAt(родитель.numChildren-1);
}
```

Следующий код удаляет всех детей в направлении снизу вверх с помощью цикла `for` вместо цикла `while`:

```
for (;numChildren > 0;) {
  родитель.removeChildAt(0);
}
```

Если вам потребуется удалить детей в направлении сверху вниз (возможно, для того, чтобы обработать их в таком порядке перед удалением), будьте осторожны, чтобы не использовать цикл, в котором значение счетчика не уменьшается, а увеличивается. Например, никогда не используйте код наподобие следующего:

```
// ВНИМАНИЕ: ПРОБЛЕМНЫЙ КОД! НЕ ИСПОЛЬЗОВАТЬ!
for (var i:int = 0; i < родитель.numChildren; i++) {
  родитель.removeChildAt(i);
}
```

Что же неправильно с предыдущим циклом `for`? Представьте, что у объекта *родитель* есть три ребенка — A, B и C, — которые находятся на глубинах 0, 1 и 2 соответственно:

```
Дети     Глубины
 A          0
 B          1
 C          2
```

Когда цикл выполняется в первый раз, значение счетчика `i` равно 0, поэтому удаляется объект A. После удаления объекта A глубины объектов B и C автоматически уменьшаются на 1, таким образом, глубина объекта B теперь равна 0, а глубина объекта C — 1:

```
Дети     Глубины
 B          0
 C          1
```

Когда цикл выполняется во второй раз, значение счетчика `i` равно 1, поэтому удаляется объект C. После удаления объекта C значение переменной *родитель*`.numChildren` становится равным 1 и цикл завершается, поскольку значение переменной `i` перестает быть меньше значения переменной *родитель*`.numChildren`. Но объект B никогда не будет удален!

Изменение родителей элементов

В языке ActionScript версии 3.0 удаление ребенка из одного экземпляра объекта `DisplayObjectContainer` и его перенос в другой экземпляр является совершенно допустимой и распространенной операцией. На самом деле действие по добавлению объекта в контейнер приводит к автоматическому удалению этого объекта из любого контейнера, в котором он находился до настоящего момента.

Чтобы продемонстрировать эту методику, в листинге 20.2 представлено простое приложение `WordHighlighter`, в котором объект `Shape` (присвоенный переменной `bgRect`) перемещается между двумя экземплярами класса `Sprite` (присвоенными переменным `word1` и `word2`). Экземпляры класса `Sprite` содержат экземпляры класса `TextField` (присвоенные переменным `text1` и `text2`), которые отображают слова Продукты и Услуги. Объект `Shape` — это прямоугольник с закругленными углами, который служит для выделения слова, находящегося в настоящий момент под указателем мыши, как показано на рис. 20.12. Когда указатель мыши находится над одним из экземпляров класса `TextField`, объект `Shape` перемещается в тот экземпляр класса `Shape`, который содержит данный объект `TextField`.

Рис. 20.12. Перемещение объекта между контейнерами

Мы еще не рассматривали методики обработки событий мыши, которые применяются в листинге 20.2. Информацию об обработке событий ввода можно найти в гл. 22.

Листинг 20.2. Перемещение объекта между контейнерами

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;

  public class WordHighlighter extends Sprite {
    // Первое слово
    private var word1:Sprite;
    private var text1:TextField;

    // Второе слово
    private var word2:Sprite;
    private var text2:TextField;

    // Выделяющая фигура
    private var bgRect:Shape;

    public function WordHighlighter ( ) {
      // Создаем первый объект TextField и Sprite
      word1 = new Sprite( );
      text1 = new TextField( );
      text1.text = "Products";
      text1.selectable = false;
      text1.autoSize = TextFieldAutoSize.LEFT;
      word1.addChild(text1)
      text1.addEventListener(MouseEvent.MOUSE_OVER, mouseOverListener);

      // Создаем второй объект TextField и Sprite
      word2 = new Sprite( );
      text2 = new TextField( );
      text2.text = "Services";
      text2.selectable = false;
      text2.autoSize = TextFieldAutoSize.LEFT;
      word2.x = 75;
      word2.addChild(text2)
      text2.addEventListener(MouseEvent.MOUSE_OVER, mouseOverListener);

      // Добавляем экземпляры класса Sprite в иерархию отображения
      // объекта WordHighlighter
      addChild(word1);
      addChild(word2);

      // Создаем объект Shape (прямоугольник с закругленными углами)
      bgRect = new Shape( );
      bgRect.graphics.lineStyle(1);
      bgRect.graphics.beginFill(0xCCCCCC, 1);
```

```
      bgRect.graphics.drawRoundRect(0, 0, 60, 15, 8);
    }

    // Вызывается при перемещении указателя мыши над текстовым полем.
    private function mouseOverListener (e:MouseEvent):void {
      // Если родительский объект Sprite экземпляра объекта TextField
      // не содержит выделяющую фигуру, перемещаем фигуру в этот объект.
      // DisplayObjectContainer.contains( ) возвращает true, если указанный
      // объект является потомком данного контейнера.
      if (!e.target.parent.contains(bgRect)) {
        e.target.parent.addChildAt(bgRect, 0);
      }
    }
  }
}
```

В таком виде код из листинга 20.2 всегда оставляет выделенным одно из текстовых полей. Для удаления выделения при выходе указателя мыши за пределы обоих текстовых полей мы должны сначала зарегистрировать приемник в обоих текстовых полях для получения события `MouseEvent.MOUSE_OUT`:

```
text1.addEventListener(MouseEvent.MOUSE_OUT, mouseOutListener);
text2.addEventListener(MouseEvent.MOUSE_OUT, mouseOutListener);
```

Затем нам придется реализовать код, который удаляет прямоугольник в ответ на событие `MouseEvent.MOUSE_OUT`:

```
private function mouseOutListener (e:MouseEvent):void {
  // Если выделение присутствует...
  if (e.target.parent.contains(bgRect)) {
    // ...удаляем его
    e.target.parent.removeChild(bgRect);
  }
}
```

Обход объектов в иерархии отображения

Обход объектов в иерархии отображения означает систематическое обращение к некоторым или ко всем дочерним объектам контейнера, в основном для обработки этих объектов.

Для обращения к *непосредственным* детям контейнера (но не к правнукам или к любым другим детям-потомкам) используется инструкция цикла. Цикл обрабатывает каждую позицию глубины в контейнере. Внутри тела цикла мы обращаемся к каждому ребенку по его позиции глубины, используя метод экземпляра `getChildAt( )` класса `DisplayObjectContainer`. Следующий код демонстрирует общий подход; он отображает строковые значения всех объектов, содержащихся в объекте *контейнер*:

```
for (var i:int=0; i < контейнер.numChildren; i++) {
  trace(контейнер.getChildAt(i).toString( ));
}
```

В листинге 20.3 приведено более конкретное и слегка причудливое приложение для обхода отображаемых дочерних объектов. Оно создает 20 экземпляров класса

Shape, содержащих прямоугольники, и затем использует описанный подход для обхода объектов, чтобы выполнять вращение этих экземпляров, когда на них щелкают кнопкой мыши. Код, осуществляющий обход, выделен полужирным шрифтом. В следующих разделах мы рассмотрим приемы векторного рисования и обработки событий мыши, применяемые в этом примере.

Листинг 20.3. Вращение прямоугольников

```
package {
  import flash.display.*;
  import flash.events.*;

  public class RotatingRectangles extends Sprite {
    public function RotatingRectangles ( ) {
      // Создаем 20 прямоугольников
      var rects:Array = new Array( );
      for (var i:int = 0; i < 20; i++) {
        rects[i] = new Shape( );
        rects[i].graphics.lineStyle(1);
        rects[i].graphics.beginFill(Math.floor(Math.random( )*0xFFFFFF), 1);
        rects[i].graphics.drawRect(0, 0, 100, 50);
        rects[i].x = Math.floor(Math.random( )*500);
        rects[i].y = Math.floor(Math.random( )*400);
        addChild(rects[i]);
      }

      // Регистрируем приемник для щелчков кнопкой мыши
      stage.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownListener);
    }

    // Вращает прямоугольники, когда пользователь щелкает кнопкой мыши
    private function mouseDownListener (e:Event):void {
      // Случайным образом вращает каждого отображаемого ребенка
      // данного объекта.
      for (var i:int=0; i < numChildren; i++) {
        getChildAt(i).rotation = Math.floor(Math.random( )*360);
      }
    }
  }
}
```

Чтобы обращаться не только к непосредственным детям контейнера, но и ко всем его потомкам, мы интегрируем предыдущий цикл for в рекурсивную функцию. В листинге 20.4 представлен общий подход.

Листинг 20.4. Рекурсивный обход дерева списка отображения

```
public function processChildren (container:DisplayObjectContainer):void {
  for (var i:int = 0; i < container.numChildren; i++) {
    // Обрабатываем ребенка здесь. Например, следующая строка выводит
    // строковое значение данного ребенка на консоль.
    var thisChild:DisplayObject = container.getChildAt(i);
    trace(thisChild.toString( ));
```

```
    // Если этот ребенок сам по себе является контейнером, приступаем
    // к обработке его детей.
    if (thisChild is DisplayObjectContainer) {
      processChildren(DisplayObjectContainer(thisChild));
    }
  }
}
```

Следующая функция `rotateChildren( )` использует обобщенный код из листинга 20.4. Она случайным образом поворачивает всех потомков указанного контейнера (а не только детей). Однако обратите внимание на небольшое изменение в подходе, представленном в листинге 20.4: функция `rotateChildren( )` вращает только тех детей, которые не являются контейнерами.

```
public function rotateChildren (container:DisplayObjectContainer):void {
  for (var i:int = 0; i < container.numChildren; i++) {
    var thisChild:DisplayObject = container.getChildAt(i);
    if (thisChild is DisplayObjectContainer) {
      rotateChildren(DisplayObjectContainer(thisChild));
    } else {
      thisChild.rotation = Math.floor(Math.random( )*360);
    }
  }
}
```

Управление одновременно несколькими объектами в контейнерах

Ранее в разд. «Обзор API отображения» было сказано, что дочерние объекты автоматически перемещаются, вращаются и трансформируются при перемещении, вращении и трансформировании их предков. Мы можем использовать эту возможность для выполнения коллективных визуальных модификаций над группами объектов. Чтобы познакомиться с этой методикой, создадим два экземпляра класса `Shape`, представляющих прямоугольники, в экземпляре класса `Sprite`:

```
// Создаем два прямоугольника
var rect1:Shape = new Shape( );
rect1.graphics.lineStyle(1);
rect1.graphics.beginFill(0x0000FF, 1);
rect1.graphics.drawRect(0, 0, 75, 50);

var rect2:Shape = new Shape( );
rect2.graphics.lineStyle(1);
rect2.graphics.beginFill(0xFF0000, 1);
rect2.graphics.drawRect(0, 0, 75, 50);
rect2.x = 50;
rect2.y = 75;

// Создаем контейнер
var group:Sprite = new Sprite( );
```

```
// Добавляем прямоугольники в контейнер
group.addChild(rect1);
group.addChild(rect2);

// Добавляем контейнер в основное приложение
некоеОсновноеПриложение.addChild(group);
```

На рис. 20.13 показан результат выполнения предыдущего кода.

Теперь с помощью следующего кода переместим контейнер, изменим его масштаб и выполним вращение:

```
group.x = 40;
group.scaleY = .15;
group.rotation = 15;
```

Изменения затрагивают дочерние экземпляры класса `Shape`, как показано на рис. 20.14.

Трансформации контейнера влияют и на дочерние элементы, добавленные в контейнер *после* выполнения трансформаций. Например, если сейчас мы добавим третий экземпляр класса `Shape`, представляющий прямоугольник, в объект `group`, этот экземпляр класса `Shape` будет перемещен, масштабирован и повернут в соответствии с существующими трансформациями объекта `group`:

```
// Создаем третий прямоугольник
var rect3:Shape = new Shape( );
rect3.graphics.lineStyle(1);
rect3.graphics.beginFill(0x00FF00, 1);
rect3.graphics.drawRect(0, 0, 75, 50);
rect3.x = 25;
rect3.y = 35;
group.addChild(rect3);
```

На рис. 20.15 показан результат.

В любой момент мы можем удалить или изменить трансформацию контейнера — модификации затронут все дочерние объекты. Например, следующий код вернет контейнер в его исходное состояние:

```
group.scaleY = 1;
group.x = 0;
group.rotation = 0;
```

На рис. 20.16 показан результат. Обратите внимание, что теперь размеры и позиция третьего прямоугольника соответствуют указанным значениям.

Цветовые и координатные трансформации, осуществляемые через переменную экземпляра `transform` класса `DisplayObject`, также наследуются всеми потомками этого экземпляра. Например, следующий код применяет черную цветовую трансформацию к объекту `group`, в результате чего все три прямоугольника окрашиваются в сплошной черный цвет.

```
import flash.geom.ColorTransform;
var blackTransform:ColorTransform = new ColorTransform( );
blackTransform.color = 0x000000;
group.transform.colorTransform = blackTransform;
```

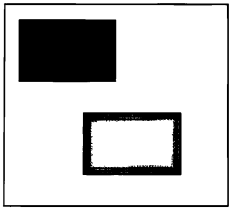

Рис. 20.13. Два прямоугольника в контейнере

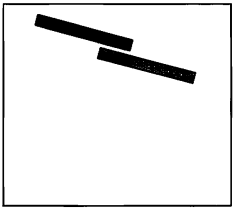

Рис. 20.14. Перемещение, масштабирование и вращение

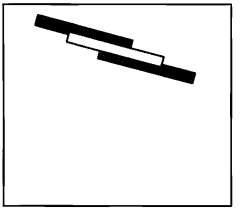

Рис. 20.15. Третий прямоугольник

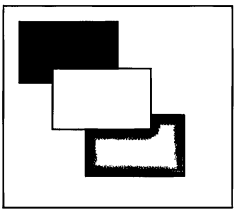

Рис. 20.16. Трансформации удалены

Подробную информацию о типах цветовых и координатных трансформаций, доступных в языке ActionScript, можно найти в описании пакета flash.geom.Transform в справочнике по языку ActionScript корпорации Adobe.

Трансформации, выполняемые над вложенными контейнерами, являются составными. Например, следующий код помещает прямоугольник в объект `Sprite`, который вложен в другой объект `Sprite`. Каждый из экземпляров класса `Sprite` будет повернут на 45°. В результате прямоугольник окажется повернутым на 90° (45 + 45).

```
// Создаем прямоугольник
var rect1:Shape = new Shape( );
rect1.graphics.lineStyle(1);
rect1.graphics.beginFill(0x0000FF, 1);
rect1.graphics.drawRect(0, 0, 75, 50);

var outerGroup:Sprite = new Sprite( );
var innerGroup:Sprite = new Sprite( );

innerGroup.addChild(rect1);
outerGroup.addChild(innerGroup);
innerGroup.rotation = 45;
outerGroup.rotation = 45;
```

Обращение к экземпляру основного класса SWF-файла из потомков

В языке ActionScript 3.0 отображаемые потомки экземпляра основного класса SWF-файла могут получать ссылку на этот экземпляр через переменную экземпляра `root` класса `DisplayObject`. Например, рассмотрим код из листинга 20.5, представляющий основной класс SWF-файла `App`. При выполнении этого кода среда Flash автоматически создает экземпляр класса `App` и выполняет его конструктор. Внутри конструктора два потомка экземпляра класса `App` (объект `Sprite` и объект `Shape`) обращаются к экземпляру класса `App` через переменную `root`.

Листинг 20.5. Обращение к экземпляру основного класса SWF-файла из потомков

```
package {
  import flash.display.*;
  import flash.geom.*;
```

```
  public class App extends Sprite {
    public function App ( ) {
      // Создаем потомков...
      var rect:Shape = new Shape( );
      rect.graphics.lineStyle(1);
      rect.graphics.beginFill(0x0000FF, 1);
      rect.graphics.drawRect(0, 0, 75, 50);
      var sprite:Sprite = new Sprite( );
      sprite.addChild(rect);
      addChild(sprite);

      // Используем выражение DisplayObject.root для обращения
      // к данному экземпляру класса App
      trace(rect.root); // Выводит: [object App]
      trace(sprite.root); // Выводит: [object App]
    }
  }
}
```

Когда объект находится в списке отображения, но *не* является потомком экземпляра основного класса SWF-файла, его переменная `stage` возвращает ссылку на экземпляр класса `Stage`. Например, следующий код модифицирует класс `App` из листинга 20.5 таким образом, чтобы объект `Sprite` и его дочерний объект `Shape` добавлялись непосредственно в экземпляр `Stage`. Поскольку объекты `Sprite` и `Shape` не являются потомками экземпляра основного класса SWF-файла, их переменные `root` ссылаются на экземпляр класса `Stage`.

```
package {
  import flash.display.*;
  import flash.geom.*;

  public class App extends Sprite {
    public function App ( ) {
      var rect:Shape = new Shape( );
      rect.graphics.lineStyle(1);
      rect.graphics.beginFill(0x0000FF, 1);
      rect.graphics.drawRect(0, 0, 75, 50);
      var sprite:Sprite = new Sprite( );
      sprite.addChild(rect);
      // Добавляем дочерний объект к экземпляру класса Stage,
      // а не к данному экземпляру класса App
      stage.addChild(sprite);

      trace(rect.root); // Отображает: [object Stage]
      trace(sprite.root); // Отображает: [object Stage]
    }
  }
}
```

Для объектов, которые находятся в списке отображения, но не являются потомками экземпляра основного класса SWF-файла, переменная экземпляра root класса DisplayObject тождественна его переменной экземпляра stage.

В первом SWF-файле, открытом средой выполнения Flash, переменной `root` всех отображаемых объектов, которые не находятся в списке отображения, присвоено значение `null`.

В SWF-файлах, которые загружаются другими SWF-файлами, значение переменной `root` устанавливается следующим образом:

- если отображаемые объекты являются отображаемыми потомками экземпляра основного класса, то переменная `root` ссылается на этот экземпляр, даже если экземпляр основного класса не находится в списке отображения;
- для отображаемых объектов, которые не являются отображаемыми потомками экземпляра основного класса и не находятся в списке отображения, переменной `root` присваивается значение `null`.

Возрождение переменной _root

В языке ActionScript 2.0 и в более старых версиях языка глобальная переменная `_root` ссылалась на самый верхний клип текущего уровня `_level`. До появления языка ActionScript 3.0 обычный здравый смысл подсказывал, что использования переменной `_root` следует избегать, поскольку ее значение было непостоянным (объект, на который ссылалась данная переменная, изменялся при загрузке SWF-файла в клип).

В языке ActionScript 3.0 глобальную переменную `_root` заменяет переменная экземпляра `root` класса `DisplayObject`. Переменная `root` класса `DisplayObject` не подвержена непостоянству, которое было свойственно ее предку, и считается чистым, безопасным членом API отображения.

Программисты на языке ActionScript, долгое время работавшие с предыдущими версиями языка и избегавшие применения устаревшей переменной _root, не должны испытывать чувство страха или вины при использовании переменной экземпляра root класса DisplayObject в языке ActionScript 3.0.

Для чего нужна переменная _level?

В языках ActionScript 1.0 и 2.0 функция `loadMovieNum( )` использовалась для размещения внешних SWF-файлов на независимых *уровнях* приложения Flash Player. Для обращения к каждому уровню использовался следующий формат: `_level`*n*, где *n* обозначает порядковый номер уровня в стеке уровней. В интерфейсе API среды выполнения Flash языка ActionScript 3.0 концепция слоев полностью отсутствует.

В языке ActionScript версии 3.0 ближайшим аналогом слоев являются дочерние объекты экземпляра класса `Stage`. Тем не менее, если в языках ActionScript 1.0 и 2.0 внешние SWF-файлы могли быть загружены непосредственно на уровень `_level`, в языке ActionScript 3.0 внешние SWF-файлы не могут быть загружены непосредственно в список дочерних объектов экземпляра класса `Stage`. Вместо этого, чтобы добавить внешний SWF-файл в список дочерних объектов экземпляра класса `Stage`, мы должны сначала загрузить его с помощью объекта `Loader`, а затем

поместить его в экземпляр класса `Stage` через вызов метода `stage.addChild( )`, как показано в следующем коде:

```
var loader:Loader = new Loader( );
loader.load(new URLRequest("newContent.swf"));
stage.addChild(loader);
```

Более того, в языке ActionScript 3.0 невозможно удалить все содержимое из приложения Flash Player, выполнив выгрузку уровня `_level0`. Код наподобие следующего больше не является допустимым:

```
// Очистить все содержимое в приложении Flash Player.
// Исключено в языке ActionScript 3.0.
unloadMovieNum(0);
```

Ближайшей заменой выражения `unloadMovieNum(0)` в языке ActionScript 3.0 является следующее выражение:

```
stage.removeChildAt(0);
```

Выражение `stage.removeChildAt(0)` удаляет первый дочерний объект экземпляра класса `Stage` из списка отображения, но не обязательно удаляет его из программы. Если в программе существуют другие ссылки на этот дочерний объект, он продолжит свое существование, при этом его можно повторно добавить в любой другой контейнер. Как было показано в подразд. «Удаление элементов из памяти» этого раздела, чтобы полностью удалить отображаемый объект из программы, необходимо не только удалить объект из контейнера, но и удалить все ссылки на него. Более того, вызов метода `stage.removeChildAt(0)` затрагивает только первый дочерний объект экземпляра класса `Stage`; другие дочерние объекты не удаляются из списка отображения (в отличие от вызова функции `unloadMovieNum(0)` в языках ActionScript 1.0 и 2.0, которая удаляет содержимое со всех уровней `_level`). Чтобы удалить все дочерние объекты экземпляра класса `Stage`, используется следующий код внутри объекта, находящегося на глубине 0 экземпляра класса `Stage`:

```
while (stage.numChildren > 0) {
  stage.removeChildAt(stage.numChildren-1);
  // Когда удаляется последний дочерний объект, переменной stage
  // присваивается значение null, поэтому завершаем цикл
  if (stage == null) {
    break;
  }
}
```

Подобным образом следующий устаревший код, который очищает приложение Flash Player от всего содержимого и затем помещает файл `newContent.swf` на уровень `_level0`, больше не является допустимым:

```
loadMovieNum("newContent.swf", 0);
```

В ActionScript 3.0 эквивалент подобного выражения отсутствует. Но вполне вероятно, что в будущих версиях языка снова появится возможность очищать среду выполнения Flash от всего содержимого, заменяя его новым внешним SWF-файлом.

События контейнеров

Мы уже знаем, как использовать методы `addChild( )` и `addChildAt( )` для добавления нового отображаемого дочернего объекта в объект `DisplayObjectContainer`. Вспомним обобщенный код:

```
// Метод addChild( )
некийКонтейнер.addChild(новыйРебенок)

// Метод addChildAt( )
некийКонтейнер.addChild(новыйРебенок, глубина)
```

Мы также знаем, что существующие дочерние отображаемые объекты могут быть удалены из объекта `DisplayObjectContainer` с помощью методов `removeChild( )` и `removeChildAt( )`. Опять же вспомним следующий обобщенный код:

```
// Метод removeChild( )
некийКонтейнер.removeChild(ребенокДляУдаления)
// Метод removeChildAt( )
некийКонтейнер.removeChildAt(глубинаРебенкаДляУдаления)
```

Наконец, мы знаем, что существующий дочерний отображаемый объект может быть удален из контейнера путем перемещения этого объекта в другой контейнер либо с помощью метода `addChild( )`, либо с помощью метода `addChildAt( )`. Вот этот код:

```
// Добавление объекта ребенок в контейнер некийКонтейнер
некийКонтейнер.addChild(ребенок)
// Удаление объекта ребенок из контейнера некийКонтейнер путем
// его перемещения в контейнер некийДругойКонтейнер
некийДругойКонтейнер.addChild(ребенок)
```

Каждая из этих операций добавления и удаления дочерних объектов сопровождается внутренним событием среды выполнения Flash — `Event.ADDED` или `Event.REMOVED`. В следующих трех разделах рассматривается, как эти два события используются в программировании экранного вывода.

Для изучения следующих разделов вам потребуется хорошее понимание системы иерархической диспетчеризации событий языка ActionScript, рассматриваемой в гл. 21. Если вы еще не до конца знакомы с иерархической диспетчеризацией событий, прочтите гл. 21 перед тем, как приступить к изучению следующих разделов.

События Event.ADDED и Event.REMOVED

После того как новый дочерний отображаемый объект добавляется в объект `DisplayObjectContainer`, среда Flash выполняет диспетчеризацию события `Event.ADDED`, получателем которого является новый дочерний объект. Подобным образом, когда существующий дочерний отображаемый объект удаляется из объекта `DisplayObjectContainer`, среда Flash выполняет диспетчеризацию события `Event.REMOVED`, получателем которого является удаленный дочерний объект.

Как будет рассмотрено в гл. 21, когда при диспетчеризации получателем события является объект в иерархии отображения, уведомление об этом событии получают данный объект и все его предки. Следовательно, когда возникает событие `Event.ADDED`, уведомление о добавленном дочернем объекте получают сам объект, его новый родительский контейнер и все предки данного контейнера. Подобным образом, когда возникает событие `Event.REMOVED`, уведомление об удаленном дочернем объекте получают сам объект, его старый родительский контейнер и все предки данного контейнера. Таким образом, события `Event.ADDED` и `Event.REMOVED` могут использоваться двумя различными способами:

- экземпляр класса `DisplayObjectContainer` может использовать эти события, чтобы определить, когда у него появляется или теряется отображаемый потомок;
- экземпляр класса `DisplayObject` может использовать данные события, чтобы определить, когда он будет добавлен в родительский контейнер или удален из него.

Рассмотрим обобщенный код, который демонстрирует описанные сценарии, начиная с контейнера, определяющего факт появления нового потомка.

Сначала мы создадим два объекта `Sprite`: один из них будет играть роль контейнера, а другой — дочернего объекта:

```
var container:Sprite = new Sprite( );
var child:Sprite = new Sprite( );
```

Теперь мы создадим метод-приемник `addedListener( )`, который будет регистрироваться в объекте `container` для получения событий `Event.ADDED`:

```
private function addedListener (e:Event):void {
  trace("Added was triggered");
}
```

После этого мы регистрируем метод `addedListener( )` в объекте `container`:

```
container.addEventListener(Event.ADDED, addedListener);
```

Наконец, мы добавляем объект `child` в объект `container`:

```
container.addChild(child);
```

При выполнении предыдущего кода среда Flash выполняет диспетчеризацию события `Event.ADDED`, получателем которого является объект `child`. В результате, поскольку объект `container` является отображаемым предком объекта `child`, на фазе всплытия события будет вызвана функция `addedListener( )`, зарегистрированная в объекте `container` (дополнительную информацию по фазе всплытия можно найти в гл. 21).

Когда событие Event.ADDED вызывает приемник события на фазе захвата или всплытия, мы знаем, что у объекта, в котором зарегистрирован приемник, появился новый отображаемый потомок.

Теперь добавим новый дочерний объект в объект `child`, превратив `container` в гордого прародителя:

```
var grandchild:Sprite = new Sprite( );
child.addChild(grandchild);
```

При выполнении предыдущего кода среда Flash снова осуществляет диспетчеризацию события `Event.ADDED`, получателем которого на этот раз является объект `grandchild`, и на фазе всплытия вновь вызывается метод `addedListener( )`. Поскольку приемник вызван на фазе всплытия, мы знаем, что у объекта `container` появился новый потомок, но не уверены, является ли этот потомок непосредственным ребенком объекта `container`. Чтобы определить это, мы проверяем, совпадает ли значение переменной `parent` объекта `child` с объектом `container`, как показано в следующем коде:

```
private function addedListener (e:Event):void {
  // Помните, что переменная Event.currentTarget ссылается на объект,
  // в котором зарегистрирован выполняемый в настоящий момент приемник, -
  // в данном случае, на объект container. Помните также, что переменная
  // Event.target ссылается на получатель события, в данном случае -
  // на объект grandchild.
  if (DisplayObject(e.target.parent) == e.currentTarget) {
    trace("A direct child was added");
  } else {
    trace("A descendant was added");
  }
}
```

Продолжая работать с нашим примером, снова сделаем объект `container` ребенком, добавив его (и соответственно два его потомка) в экземпляр класса `Stage`:

```
stage.addChild(container);
```

При выполнении предыдущего кода среда Flash снова осуществляет диспетчеризацию события `Event.ADDED`, получателем которого является объект `container`. И снова вызывается метод `addedListener( )` — на этот раз на фазе получения, а не на фазе всплытия.

Когда событие Event.ADDED вызывает приемник события на фазе получения, мы знаем, что объект, в котором зарегистрирован приемник, был добавлен в родительский контейнер.

Чтобы различать ситуации, когда объект `container` приобретает нового потомка и когда сам объект `container` добавляется в родительский контейнер, мы проверяем текущую фазу события, как показано в следующем коде:

```
private function addedListener (e:Event):void {
  // Если этот приемник был вызван на фазе захвата или всплытия...
  if (e.eventPhase != EventPhase.AT_TARGET) {
    // ...контейнер получил нового потомка
    trace("new descendant: " + e.target);
  } else {
    // ...в противном случае контейнер был добавлен к новому родителю
    trace("new parent: " + DisplayObject(e.target).parent);
  }
}
```

Теперь перейдем к событию Event.REMOVED. Оно работает аналогично событию Event.ADDED, но его диспетчеризация выполняется при удалении объектов, а не при их добавлении.

Следующий код регистрирует приемник события Event.REMOVED с именем removedListener( ) в объекте container для события Event.REMOVED:

```
container.addEventListener(Event.REMOVED, removedListener);
```

Теперь удалим потомка из объекта container:

```
child.removeChild(grandchild)
```

При выполнении предыдущего кода среда Flash осуществляет диспетчеризацию события Event.REMOVED, получателем которого является объект grandchild, и метод removedListener( ) вызывается на фазе всплытия.

Далее следующий код удаляет сам объект container из экземпляра класса Stage:

```
stage.removeChild(container)
```

При выполнении приведенного кода среда Flash осуществляет диспетчеризацию события Event.REMOVED, получателем которого является объект container, и метод removedListener( ) вызывается на фазе получения.

Как и в случае с методом addedListener( ), в методе removedListener( ) мы можем различать ситуации, когда объект container теряет потомка и когда сам объект container удаляется из своего родительского контейнера, проверяя текущую фазу события, как показано в следующем коде:

```
private function removedListener (e:Event):void {
  // Если этот приемник был вызван на фазе захвата или на фазе всплытия...
  if (e.eventPhase != EventPhase.AT_TARGET) {
    // ...из контейнера удаляется потомок
    trace("a descendant was removed from container: " + e.target);
  } else {
    // ...в противном случае контейнер удаляется из своего родителя
    trace("container is about to be removed from its parent: "
          + DisplayObject(e.target).parent);
  }
}
```

В листинге 20.6 представлены предыдущие примеры кода для событий Event.ADDED и Event.REMOVED в контексте тестового класса ContainmentEventDemo. Мы рассмотрим реальные примеры использования событий контейнеров в двух следующих разделах.

Листинг 20.6. Демонстрация событий контейнера

```
package {
  import flash.display.*;
  import flash.events.*;

  public class ContainmentEventDemo extends Sprite {
    public function ContainmentEventDemo ( ) {
      // Создаем объекты Sprite
```

```
      var container:Sprite = new Sprite( );
      var child:Sprite = new Sprite( );
      var grandchild:Sprite = new Sprite( );

      // Приступаем к приему событий Event.ADDED и Event.REMOVED
      container.addEventListener(Event.ADDED, addedListener);
      container.addEventListener(Event.REMOVED, removedListener);

      // Добавляем объект child в объект container
      container.addChild(child);  // Вызывает метод addedListener( )
                                  // на фазе всплытия

      // Добавляем объект grandchild в объект child
      child.addChild(grandchild);  // Вызывает метод addedListener( )
                                   // на фазе всплытия

      // Добавляем объект container в экземпляр класса Stage
      stage.addChild(container);  // Вызывает метод addedListener( )
                                  // на фазе получения

      // Удаляем объект grandchild из объекта child
      child.removeChild(grandchild)  // Вызывает метод removedListener( )
                                     // на фазе всплытия

      // Удаляем объект container из экземпляра класса Stage
      stage.removeChild(container)  // Вызывает метод removedListener( )
                                    // на фазе получения
    }

    // Обрабатывает события Event.ADDED
    private function addedListener (e:Event):void {
      if (e.eventPhase != EventPhase.AT_TARGET) {
        trace("container has a new descendant: " + e.target);
      } else {
        trace("container was added to a new parent: "
              + DisplayObject(e.target).parent);
      }
    }

    // Обрабатывает события Event.REMOVED
    private function removedListener (e:Event):void {
      if (e.eventPhase != EventPhase.AT_TARGET) {
        trace("a descendant was removed from container: " + e.target);
      } else {
        trace("container was removed from its parent: "
              + DisplayObject(e.target).parent);
      }
    }
  }
}

// В результате выполнения приложения ContainmentEventDemo
// выводятся следующие строки:
```

```
container has a new descendant: [object Sprite]
container has a new descendant: [object Sprite]
container was added to a new parent: [object Stage]
a descendant was removed from container: [object Sprite]
container was removed from its parent: [object Stage]
```

Реальный пример использования событий контейнера

Теперь, когда известно, как работают события `Event.ADDED` и `Event.REMOVED` в теории, рассмотрим, как можно использовать их в реальном приложении. Предположим, что мы создаем класс `IconPanel`, который управляет визуальным расположением графических значков. Класс `IconPanel` используется в качестве одной из частей большего оконного компонента в интерфейсе, основанном на оконном представлении. Всякий раз при добавлении нового или удалении существующего значка из объекта `IconPanel` этот объект исполняет алгоритм по размещению значков. Чтобы определить моменты добавления и удаления дочерних значков, объект `IconPanel` регистрирует приемники для событий `Event.ADDED` и `Event.REMOVED`.

В листинге 20.7 представлен код класса `IconPanel`, который намеренно был упрощен, чтобы продемонстрировать использование событий `Event.ADDED` и `Event.REMOVED`. Обратите внимание, что приемники событий `Event.ADDED` и `Event.REMOVED` выполняют код по размещению значков только в том случае, когда объект `IconPanel` приобретает нового или теряет существующего непосредственного ребенка. Код по размещению значков не выполняется в следующих ситуациях:

- когда объект `IconPanel` приобретает или теряет потомка, который не является непосредственным ребенком;
- если сам объект `IconPanel` добавляется в родительский контейнер.

Листинг 20.7. Размещение значков в классе IconPanel

```
package {
  import flash.display.*;
  import flash.events.*;

  public class IconPanel extends Sprite {
    public function IconPanel ( ) {
      addEventListener(Event.ADDED, addedListener);
      addEventListener(Event.REMOVED, removedListener);
    }

    public function updateLayout ( ):void {
      // Выполняет алгоритм размещения значков (код не приводится)
    }

    // Обрабатывает события Event.ADDED
    private function addedListener (e:Event):void {
      if (DisplayObject(e.target.parent) == e.currentTarget) {
```

```
        updateLayout( );
      }
    }

    // Обрабатывает события Event.REMOVED
    private function removedListener (e:Event):void {
      if (DisplayObject(e.target.parent) == e.currentTarget) {
        updateLayout( );
      }
    }
  }
}
```

События ADDED_TO_STAGE и REMOVED_FROM_STAGE

Как мы уже знаем из двух приведенных выше разделов, события `Event.ADDED` и `Event.REMOVED` возникают, когда экземпляр класса `DisplayObject` добавляется в экземпляр класса `DisplayObjectContainer` или удаляется из него. Однако по событиям `Event.ADDED` и `Event.REMOVED` нельзя определить, находится данный объект в настоящий момент в списке отображения или нет. Чтобы определить, когда экземпляр класса `DisplayObject` добавляется в список отображения или удаляется из него, используются события `Event.ADDED_TO_STAGE` и `Event.REMOVED_FROM_STAGE`, которые были добавлены в API отображения, начиная с приложения Flash Player 9.0.28.0.

Когда отображаемый объект (или один из его предков) добавляется в список отображения, Flash выполняет диспетчеризацию события `Event.ADDED_TO_STAGE`, получателем которого является данный объект. С другой стороны, когда отображаемый объект (или один из его предков) удаляется из списка отображения, среда Flash выполняет диспетчеризацию события `Event.REMOVED_FROM_STAGE`, получателем которого является данный объект.

В отличие от событий Event.ADDED и Event.REMOVED, у событий Event.ADDED_TO_STAGE и Event.REMOVED_FROM_STAGE отсутствует фаза всплытия. Для получения события Event.ADDED_TO_STAGE или Event.REMOVE_FROM_STAGE через предка объекта зарегистрируйте приемник в этом предке для фазы захвата события.

Обобщенный код, который необходим для регистрации приемника события `Event.ADDED_TO_STAGE` в экземпляре класса `DisplayObject`, выглядит следующим образом:

```
отображаемыйОбъект.addEventListener(Event.ADDED_TO_STAGE,
                                    приемникСобытияAddedToStage);
```

Код, необходимый для приемника события `Event.ADDED_TO_STAGE`, выглядит таким образом:

```
private function приемникСобытияAddedToStage (e:Event):void {
}
```

Обобщенный код, который необходим для регистрации приемника события `Event.REMOVED_FROM_STAGE` в экземпляре класса `DisplayObject`, выглядит следующим образом:

```
отображаемыйОбъект.addEventListener(Event.REMOVED_FROM_STAGE,
                                    приемникСобытияRemovedFromStage);
```

Теперь рассмотрим обобщенный код, необходимый для приемника события `Event.REMOVED_FROM_STAGE`:

```
private function приемникСобытияRemovedFromStage (e:Event):void {
}
```

Отображаемые объекты обычно используют событие `Event.ADDED_TO_STAGE`, чтобы убедиться в доступности объекта `Stage` перед обращением к его методам, переменным или событиям. Предположим, что мы создаем класс `CustomMousePointer`, который представляет пользовательский указатель мыши. Наш класс `CustomMousePointer` расширяет класс `Sprite`, поэтому его экземпляры могут добавляться в список отображения. Мы хотим, чтобы наш класс был зарегистрирован в экземпляре класса `Stage` для получения событий `MouseEvent.MOUSE_EVENT`, благодаря чему мы сможем синхронизировать позицию пользовательского указателя мыши с позицией системного указателя мыши.

Однако после создания новый объект `CustomMousePointer` не находится в списке отображения, поэтому не имеет доступа к экземпляру класса `Stage` и не может зарегистрироваться для получения события `MouseEvent.MOUSE_MOVE`. Вместо этого объект должен ожидать уведомления о том, что он был добавлен в список отображения (через событие `Event.ADDED_TO_STAGE`). Как только объект `CustomMousePointer` будет добавлен в список отображения, его переменная `stage` будет ссылаться на экземпляр класса `Stage` и он сможет успешно зарегистрироваться для получения события `MouseEvent.MOUSE_MOVE`. Следующий код демонстрирует соответствующий фрагмент кода, относящийся к событию `Event.ADDED_TO_STAGE`, из класса `CustomMousePointer`. Полный листинг кода класса `CustomMousePointer` можно найти в подразд. «Определение позиции указателя мыши» разд. «События мыши» гл. 22.

```
package {
  public class CustomMousePointer extends Sprite {
    public function CustomMousePointer ( ) {
      // Просим сообщить, когда этот объект будет добавлен
      // в список отображения
      addEventListener(Event.ADDED_TO_STAGE, addedToStageListener);
    }

    // Вызывается, когда этот объект добавляется в список отображения
    private function addedToStageListener (e:Event):void {
      // Теперь можно безопасно регистрироваться в экземпляре класса Stage
      // на получение событий MouseEvent.MOUSE_MOVE
      stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
    }
  }
}
```

Собственные события Event.ADDED_TO_STAGE и Event.REMOVE_FROM_STAGE. Первая версия приложения Flash Player 9 не содержала ни события `Event.ADDED_TO_STAGE`, ни события `Event.REMOVED_FROM_STAGE`. Тем не

менее, воспользовавшись обычным интерфейсом API отображения и проявив немного изобретательности, мы можем вручную определить, что данный объект был добавлен в список отображения или удален из него. Для этого мы должны отслеживать состояние предков данного объекта с помощью событий `Event.ADDED` и `Event.REMOVED`.

В листинге 20.8 демонстрируется данный подход. В этом примере пользовательский класс `StageDetector` следит за отображаемым объектом, чтобы определить, когда он добавляется в список отображения или удаляется из него. Класс `StageDetector` рассылает пользовательское событие:

- `StageDetector.ADDED_TO_STAGE` — когда объект добавляется в список отображения;
- `StageDetector.REMOVED_FROM_STAGE` — когда объект удаляется из списка отображения.

Вы вполне можете применять пользовательские события `ADDED_TO_STAGE` и `REMOVED_FROM_STAGE`, даже не имея ни малейшего представления о том, как устроен и функционирует класс `StageDetector`. Однако он позволяет провести интересный обзор методик программирования, относящихся к списку отображения, которые были рассмотрены в этой главе, поэтому посмотрим, как он работает.

В классе `StageDetector` объект, для которого отслеживаются пользовательские события `ADDED_TO_STAGE` и `REMOVED_FROM_STAGE`, присваивается переменной `watchedRoot`. Рассмотрим общий подход, применяемый в классе `StageDetector` для определения присутствия объекта `watchedObject` в списке отображения.

- Нужно отслеживать события `Event.ADDED` и `Event.REMOVED` для объекта `watchedRoot`.
- Когда объект `watchedRoot` добавляется в объект `DisplayObjectContainer`, проверять, находится ли `watchedObject` в настоящий момент в списке отображения (он находится в списке отображения, если значение его переменной `stage` не равно `null`). Если в настоящий момент объект `watchedObject` находится в списке отображения, то следует выполнить диспетчеризацию события `StageDetector.ADDED_TO_STAGE`. Если нет, то приступить к отслеживанию событий `Event.ADDED` и `Event.REMOVED` для нового объекта `watchedRoot`.
- Когда объект `watchedObject` есть в списке отображения, если `watchedRoot` или любой из его потомков удаляется из объекта `DisplayObjectContainer`, проверять, является ли удаленный объект предком `watchedObject`. Если да, то выполнить диспетчеризацию события `StageDetector.REMOVED_FROM_STAGE` и приступить к отслеживанию событий `Event.ADDED` и `Event.REMOVED` для нового корня иерархии отображения объекта `watchedObject`.

Теперь рассмотрим код класса `StageDetector`.

Листинг 20.8. Пользовательские события ADDED_TO_STAGE и REMOVED_FROM_STAGE

```
package {
  import flash.display.*;
  import flash.events.*;
```

```
// Наблюдает за указанным отображаемым объектом, чтобы определить,
// когда этот объект добавляется или удаляется из экземпляра класса Stage,
// и рассылает соответствующие пользовательские события
// StageDetector.ADDED_TO_STAGE и StageDetector.REMOVED_FROM_STAGE.

// ИСПОЛЬЗОВАНИЕ:
// var stageDetector:StageDetector = new StageDetector(someDisplayObject);
// stageDetector.addEventListener(StageDetector.ADDED_TO_STAGE,
//                                addedToStageListenerFunction);
// stageDetector.addEventListener(StageDetector.REMOVED_FROM_STAGE,
//                                removedFromStageListenerFunction);
public class StageDetector extends EventDispatcher {
  // Событийные константы
  public static const ADDED_TO_STAGE:String = "ADDED_TO_STAGE";
  public static const REMOVED_FROM_STAGE:String = "REMOVED_FROM_STAGE";

  // Объект, для которого будут генерироваться события ADDED_TO_STAGE
  // и REMOVED_FROM_STAGE
  private var watchedObject:DisplayObject = null;

  // Корень иерархии отображения, содержащей объект watchedObject
  private var watchedRoot:DisplayObject = null;

  // Флаг, который говорит, находится ли объект watchedObject
  // в настоящий момент в списке отображения
  private var onStage:Boolean = false;

  // Конструктор
  public function StageDetector (objectToWatch:DisplayObject) {
    // Приступаем к мониторингу указанного объекта
    setWatchedObject(objectToWatch);
  }

  // Приступает к мониторингу указанного объекта, чтобы определить,
  // добавлен объект в список отображения или удален из него
  public function setWatchedObject (objectToWatch:DisplayObject):void {
    // Сохраняем отслеживаемый объект
    watchedObject = objectToWatch;

    // Помечает, находится ли объект watchedObject в настоящий момент
    // в списке отображения
    if (watchedObject.stage != null) {
      onStage = true;
    }

    // Находит корень иерархии отображения, содержащей объект
    // watchedObject, и регистрируем в найденном объекте приемники
    // для событий ADDED/REMOVED. Проверяя, добавлен или удален
    // корень объекта watchedObject, мы сможем определить,
    // находится объект watchedObject в списке отображения или нет.
    setWatchedRoot(findWatchedObjectRoot( ));
  }
```

```
// Возвращает ссылку на отслеживаемый объект
public function getWatchedObject ( ):DisplayObject {
  return watchedObject;
}

// Освобождает ресурсы данного объекта StageDetector. Вызываем этот
// метод перед уничтожением объекта StageDetector.
public function dispose ( ):void {
  clearWatchedRoot( );
  watchedObject = null;
}

// Обрабатывает события Event.ADDED, получателем которых является
// корневой объект иерархии отображения объекта watchedObject
private function addedListener (e:Event):void {
  // Если текущий объект watchedRoot был добавлен...
  if (e.eventPhase == EventPhase.AT_TARGET) {
    // ...проверяем, находится ли объект watchedObject в настоящий
    // момент в списке отображения
    if (watchedObject.stage != null) {
      // Помечаем, что объект watchedObject теперь находится
      // в списке отображения
      onStage = true;
      // Сообщаем приемникам, что объект watchedObject теперь
      // находится в списке отображения
      dispatchEvent(new Event(StageDetector.ADDED_TO_STAGE));
    }
    // Объект watchedRoot был добавлен в другой контейнер, поэтому
    // сейчас корнем иерархии отображения объекта, содержащей объект
    // watchedObject, является новый объект. Находим этот новый корень
    // и регистрируемся в нем для получения событий ADDED и REMOVED.
    setWatchedRoot(findWatchedObjectRoot( ));
  }
}

// Обрабатывает события Event.REMOVED для корневого объекта иерархии
// отображения объекта watchedObject
private function removedListener (e:Event):void {
  // Если объект watchedObject находится в списке отображения...
  if (onStage) {
    // ...проверяем, был ли удален объект watchedObject
    // или один из его предков
    var wasRemoved:Boolean = false;
    var ancestor:DisplayObject = watchedObject;
    var target:DisplayObject = DisplayObject(e.target);
    while (ancestor != null) {
      if (target == ancestor) {
        wasRemoved = true;
        break;
      }
      ancestor = ancestor.parent;
    }
```

```
    // Если объект watchedObject или один из его предков был удален...
    if (wasRemoved) {
      // ...регистрируемся для получения событий ADDED и REMOVED
      // от удаленного объекта (который является новым корнем
      // иерархии отображения объекта watchedObject).
      setWatchedRoot(target);

      // Помечает, что объект watchedObject больше не находится
      // в списке отображения
      onStage = false;

      // Сообщаем приемникам, что объект watchedObject был удален
      // из объекта Stage
      dispatchEvent(new Event(StageDetector.REMOVED_FROM_STAGE));
    }
  }
}

// Возвращает корневой объект иерархии отображения, в настоящий момент
// содержащий объект watchedObject
private function findWatchedObjectRoot ( ):DisplayObject {
  var watchedObjectRoot:DisplayObject = watchedObject;
  while (watchedObjectRoot.parent != null) {
    watchedObjectRoot = watchedObjectRoot.parent;
  }
  return watchedObjectRoot;
}

// Начинает отслеживание событий ADDED и REMOVED, получателем
// которых является корневой объект иерархии отображения
// объекта watchedObject
private function setWatchedRoot (newWatchedRoot:DisplayObject):void {
  clearWatchedRoot( );
  watchedRoot = newWatchedRoot;
  registerListeners(watchedRoot);
}

// Удаляет приемники событий из объекта watchedRoot и ссылку
// на объект watchedRoot из данного объекта StageDetector
private function clearWatchedRoot ( ):void {
  if (watchedRoot != null) {
    unregisterListeners(watchedRoot);
    watchedRoot = null;
  }
}
// Регистрирует приемники событий ADDED и REMOVED в объекте watchedRoot
private function registerListeners (target:DisplayObject):void {
  target.addEventListener(Event.ADDED, addedListener);
  target.addEventListener(Event.REMOVED, removedListener);
}
```

```
    // Отменяет регистрацию приемников событий ADDED и REMOVED
    // в объекте watchedRoot
    private function unregisterListeners (target:DisplayObject):void {
      target.removeEventListener(Event.ADDED, addedListener);
      target.removeEventListener(Event.REMOVED, removedListener);
    }
  }
}
```

О том, как пользовательские события `StageDetector.ADDED_TO_STAGE` и `Stage.REMOVED_FROM_STAGE` применяются в классе `CustomMousePointer`, будет рассказано в гл. 22.

Мы завершили изучение интерфейса API, связанного с контейнерами. Напоследок рассмотрим одну небольшую, но очень важную тему в программировании экранного вывода: пользовательские графические классы.

Пользовательские графические классы

В этой главе мы нарисовали множество прямоугольников, окружностей и треугольников. Так много, что некоторые из рассмотренных примеров содержали «код с погрешностями»: код повторялся и, как результат, способствовал ошибкам.

Дополнительную информацию о «коде с погрешностями» (общие признаки потенциальных проблем в коде) можно найти по адресу http://xp.c2.com/CodeSmell.html.

Чтобы обеспечить возможность повторного применения и модульность при работе с примитивными фигурами, мы можем перенести повторяющиеся процедуры рисования в пользовательские классы, которые расширяют класс `Shape`. Начнем с пользовательского класса `Rectangle`, применив чрезвычайно простой подход, который предоставляет весьма ограниченный набор вариантов контура и заливки и не позволяет изменять прямоугольник после того, как он будет нарисован (мы расширим возможности класса `Rectangle` в гл. 25). Этот код продемонстрирован в листинге 20.9.

Листинг 20.9. Rectangle — простой подкласс класса Shape

```
package {
  import flash.display.Shape;

  public class Rectangle extends Shape {
    public function Rectangle (w:Number,
                               h:Number,
                               lineThickness:Number,
                               lineColor:uint,
                               fillColor:uint) {
      graphics.lineStyle(lineThickness, lineColor);
```

```
      graphics.beginFill(fillColor, 1);
      graphics.drawRect(0, 0, w, h);
    }
  }
}
```

Поскольку класс `Rectangle` расширяет класс `Shape`, он наследует переменную `graphics` класса `Shape` и может использовать ее для рисования прямоугольника.

Для создания нового экземпляра класса `Rectangle` применяется следующий знакомый нам код:

```
var rect:Rectangle = new Rectangle(100, 50, 3, 0xFF0000, 0x0000FF);
```

Поскольку класс `Shape` является потомком `DisplayObject`, класс `Rectangle` наследует возможность быть добавленным в список отображения (как и любой другой потомок класса `DisplayObject`), наподобие следующего кода:

```
некийКонтейнер.addChild(rect);
```

Являясь потомком класса `DisplayObject`, объект `Rectangle` может быть позиционирован, повернут и подвергнут другим действиям, как и любой другой отображаемый объект. Например, следующий код устанавливает горизонтальной позиции объекта `Rectangle` значение 15, а вертикальной — значение 30:

```
rect.x = 15;
rect.y = 30;
```

Однако развлечения на этом не заканчиваются. Любой класс API отображения может быть расширен. Например, приложение может расширить класс `TextField` для отображения специализированной текстовой формы. Это демонстрирует код из листинга 20.10, который представляет подкласс класса `TextField`, создающий текстовый заголовок с возможностью перехода по ссылке.

Листинг 20.10. ClickableHeading — подкласс класса TextField

```
package {
  import flash.display.*;

  public class ClickableHeading extends TextField {
    public function ClickableHeading (headText:String, URL:String) {
      html = true;
      autoSize = TextFieldAutoSize.LEFT;
      htmlText = "<a href='" + URL + "'>" + headText + "</a>";
      border = true;
      background = true;
    }
  }
}
```

Вот как можно использовать класс `ClickableHeading` в приложении:

```
var head:ClickableHeading = new ClickableHeading(
                                          "Essential ActionScript 3.0",
                                          "http://www.moock.org/eas3");
addChild(head);
```

На рис. 20.17 показано результирующее содержимое, отображаемое на экране. При выполнении этого примера в среде Flash текст будет связан с сопутствующим сайтом для этой книги.

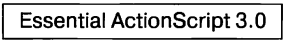

Рис. 20.17. Экземпляр класса ClickableHeading

В следующих главах мы встретим множество примеров отображаемых подклассов. Разрабатывая графические элементы, необходимые для ваших приложений, подумайте над возможностью расширения существующего класса отображения вместо того, чтобы создавать классы с нуля.

Переходим к цепочке диспетчеризации событий

После прочтения этой главы вы должны чувствовать себя достаточно комфортно, создавая отображаемое содержимое и выводя его на экран. Многие примеры, представленные далее в книге, опираются на принципы, рассмотренные в этой главе, поэтому у вас появится масса возможностей, чтобы вспомнить пройденный материал и расширить полученные знания. Из следующей главы мы узнаем, как событийная архитектура языка ActionScript 3.0 применяется к объектам в списке отображения.

События и иерархии отображения

В гл. 12 мы в общих чертах ознакомились с внутренней событийной архитектурой языка ActionScript. В этой главе мы подробно рассмотрим, как эта событийная архитектура адаптируется к объектам в иерархиях отображений.

Система диспетчеризации событий через иерархию объектов языка ActionScript, о которой пойдет речь в этой главе, основана на спецификации Document Object Model (DOM) Level 3 Events Specification консорциума W3C, доступной по адресу http://www.w3.org/TR/DOM-Level-3-Events.

Иерархическая диспетчеризация событий

Как мы уже видели в гл. 12, когда среда Flash выполняет диспетчеризацию события, получателем которого является объект, не входящий в состав иерархии отображения, этот получатель будет единственным, кто узнает о возникновении события. Например, когда завершается воспроизведение звукового файла в объекте `Sound`, среда Flash выполняет диспетчеризацию события `Event.COMPLETE`, получателем которого выступает соответствующий объект `SoundChannel`. Этот объект не входит в состав иерархии отображения, поэтому он будет единственным объектом, который узнает о возникновении этого события.

В отличие от этого, когда среда Flash выполняет диспетчеризацию события, получателем которого является объект, входящий в состав иерархии отображения, этот получатель и все его предки в иерархии отображения узнают о возникновении события. Например, если объект `Sprite` содержит объект `TextField` и пользователь щелкает кнопкой мыши на втором объекте, то и `TextField` (получатель события), и `Sprite` (предок получателя события) узнают о том, что произошел щелчок кнопкой мыши.

Система иерархической диспетчеризации событий языка ActionScript позволяет каждому контейнеру отображаемых объектов регистрировать приемники для обработки событий, получателями которых являются отображаемые объекты-потомки этого контейнера. Например, объект `Sprite`, представляющий окно, может зарегистрировать приемник, обрабатывающий события о щелчке кнопкой мыши, получателем которых является вложенный элемент управления «кнопка OK». Или объект `Sprite`, представляющий форму для авторизации, может зарегистрировать приемник, обрабатывающий события фокуса, получателями которых являются вложенные поля ввода.

Такая централизованная архитектура позволяет сократить объемы повторяющегося кода, особенно реагирующего на события пользовательского ввода. Далее, в разд. «Использование цепочки диспетчеризации события для централизации кода», мы рассмотрим пример, который демонстрирует преимущества централизованной обработки событий. Но для начала познакомимся с основами иерархической диспетчеризации событий и регистрации.

В этой главе термины «предок» и «потомок» в основном используются для обозначения объектов в иерархии отображения, а не суперклассов и подклассов в иерархии наследования. Чтобы избежать путаницы, в этой главе иногда используются неофициальные термины «отображаемый предок» и «отображаемый потомок» для обозначения объектов-предков и объектов-потомков в иерархии отображения.

Фазы диспетчеризации событий

Как уже известно, когда среда Flash выполняет диспетчеризацию события, получателем которого является объект в иерархии отображения, о возникновении события узнает не только данный получатель, но и все его отображаемые предки. Процесс, в результате которого о возникновении события узнают получатель и все его предки, разбивается на три отдельные фазы. На первой фазе процесса диспетчеризации события, называемой *фазой захвата*, уведомление о возникновении события получают все предки объекта-получателя. Как только все предки объекта-получателя получат уведомление о возникновении события, начинается вторая фаза процесса диспетчеризации события, называемая *фазой получения*. На этой фазе среда выполнения Flash уведомляет объект-получатель о возникновении события.

Для некоторых типов событий процесс диспетчеризации завершается сразу после окончания фазы получения. Для остальных типов событий процесс диспетчеризации переходит в третью фазу, называемую *фазой всплытия*. На этой фазе предки объекта-получателя узнают о том, что получатель был успешно уведомлен о возникновении события. События, имеющие фазу всплытия, называются *всплывающими*; события, не имеющие фазу всплытия, называются *невсплывающими*.

У четырех типов событий — Event.ACTIVATE, Event.DEACTIVATE, Event.ENTER_FRAME и Event.RENDER — есть только фаза получения. Процесс диспетчеризации всех остальных событий, получателем которых является объект в иерархии отображения, включает фазу захвата и фазу получения. Некоторые типы событий также имеют фазу всплытия.

Порядок, в котором объекты узнают о возникновении события в процессе диспетчеризации, зависит от фазы события. На фазе захвата уведомление предков начинается от корневого объекта иерархии отображения объекта-получателя и, проходя вниз по всем потомкам, завершается на непосредственном родителе объекта-получателя. На фазе захвата уведомление самого получателя не происходит. На фазе всплытия уведомление предков происходит в порядке, обратном порядку на фазе захвата, — начинается от непосредственного родителя объекта-получателя и, проходя вверх, заканчивается на корневом объекте иерархии. Процесс, в результате

которого уведомление о событии передается вниз через предков объекта-получателя (фаза захвата) к объекту-получателю (фаза получения) и обратно через его предков (фаза всплытия), называется *цепочкой диспетчеризации события*. Когда уведомление о событии проходит по цепочке диспетчеризации события, говорят, что событие *передается* от объекта к объекту.

Рассмотрим простой пример цепочки диспетчеризации события. Предположим, что экземпляр класса `Stage` содержит объект `Sprite`, который, в свою очередь, содержит объект `TextField`, как показано на рис. 21.1. Видно, что корнем иерархии отображения объекта `TextField` является экземпляр класса `Stage`, а непосредственным родителем объекта `TextField` — объект `Sprite`.

Теперь предположим, что пользователь вводит некий текст в объект `TextField`, в результате чего среда Flash вынуждена выполнить диспетчеризацию события `TextEvent.TEXT_INPUT`, получателем которого является объект `TextField`. Поскольку объект `TextField` является частью иерархии отображения, событие передается по цепочке диспетчеризации события. В первой фазе процесса (фазе захвата) уведомление о возникновении события сначала получает экземпляр класса `Stage`, а затем — экземпляр класса `Sprite`. Во второй фазе (фазе получения) уведомление о возникновении события получает сам объект `TextField`. Наконец, в третьей фазе процесса (фазе всплытия) уведомление о том, что получатель был уведомлен о возникновении события, сначала получает экземпляр класса `Sprite`, а затем — экземпляр класса `Stage`. Всего в процессе диспетчеризации события `TextEvent.TEXT_INPUT` происходит пять уведомлений о возникновении события, как показано на рис. 21.2.

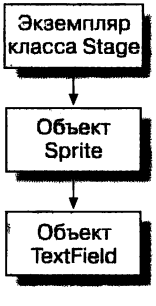

Рис. 21.1. Пример иерархии отображения

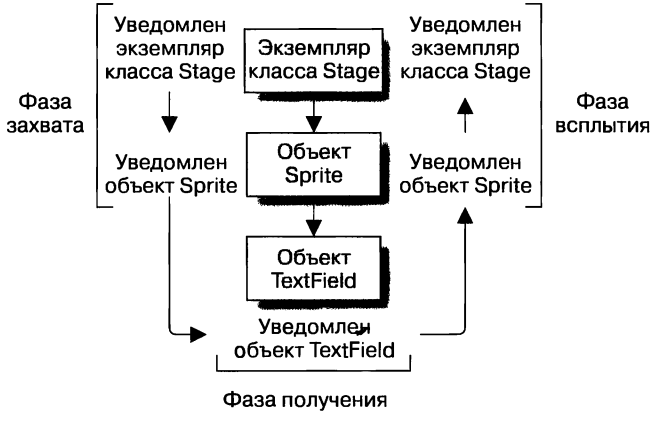

Рис. 21.2. Цепочка диспетчеризации для события TextEvent.TEXT_INPUT

Приемники событий и цепочка диспетчеризации событий

Как мы только что увидели, в процессе диспетчеризации события, получателем которого является некоторый отображаемый объект, отображаемые предки этого

объекта получают уведомление о возникновении события на фазе захвата и теоретически на фазе всплытия (если событие является всплывающим). Соответственно, при регистрации приемника в предке получателя события мы должны указать, когда вызывать этот приемник — в фазе захвата или в фазе всплытия.

Чтобы зарегистрировать приемник в предке получателя события для фазы захвата диспетчеризации события, мы устанавливаем третьему параметру `useCapture` метода `addEventListener( )` значение `true`, как показано в следующем коде:

```
предок.addEventListener(событие, приемник, true)
```

Эта строка кода заставляет метод *приемник( )* выполняться всякий раз, когда среда Flash осуществляет диспетчеризацию события *событие*, получателем которого является один из потомков объекта *предок*, *до того* как потомки получат уведомление о возникновении этого события.

Чтобы зарегистрировать приемник в предке получателя события для фазы всплытия диспетчеризации события, мы устанавливаем третьему параметру `useCapture` метода `addEventListener( )` значение `false`, как показано в следующем коде:

```
предок.addEventListener(событие, приемник, false)
```

В качестве альтернативы, поскольку в качестве значения по умолчанию параметра `useCapture` используется `false`, мы можем просто опустить аргумент `useCapture`, как показано в следующем коде:

```
предок.addEventListener(событие, приемник)
```

Приведенная строка кода заставляет метод *приемник( )* выполняться всякий раз, когда среда Flash осуществляет диспетчеризацию события *событие*, получателем которого является один из потомков объекта *предок*, *после того* как потомки получат уведомление о возникновении этого события.

Для краткости в оставшейся части этой главы мы будем использовать неофициальный термин «приемник предка», который обозначает «приемник события, зарегистрированный в отображаемом предке получателя события». Подобным образом, мы будем использовать термин «приемник получателя», который обозначает «приемник события, зарегистрированный непосредственно в получателе события».

При регистрации приемника предка для *невсплывающего* события мы всегда выполняем регистрацию для фазы захвата (то есть в качестве значения параметра `useCapture` передаем `true`). В противном случае этот приемник вызван не будет. При регистрации приемника предка для *всплывающего* события мы выбираем уведомления либо в фазе захвата (значение параметра `useCapture` — `true`), либо в фазе всплытия (значение параметра `useCapture` — `false`), или же оба типа уведомлений, в зависимости от потребностей приложения.

Фаза захвата дает возможность приемникам предков обработать событие до того, как приемники получателя события ответят на него. Обычно приемники, зарегистрированные для фазы захвата, используются для прекращения передачи события его получателю в зависимости от некоторого условия. Например, элемент пользовательского интерфейса, представляющий панель с состояниями

«доступна» и «недоступна», может использовать приемник, зарегистрированный для фазы захвата, чтобы предотвратить передачу событий мыши потомкам этой панели, когда панель находится в недоступном состоянии (как останавливать события, будет описано далее, в разд. «Остановка процесса диспетчеризации события»).

В отличие от этого, фаза всплытия дает возможность приемникам предков обработать событие после того, как приемники получателя события уже отреагируют на него. Обычно фаза всплытия применяется для реакции на изменения в состоянии целевого объекта перед тем, как будет продолжено выполнение программы и обновлено содержимое на экране. Например, элемент пользовательского интерфейса, представляющий панель, которая содержит перетаскиваемые значки, может использовать приемник, зарегистрированный для фазы всплытия, чтобы автоматически выравнивать значки после перетаскивания одного из них.

В отличие от приемников предков, приемники, зарегистрированные в получателе события, могут вызываться только в одной фазе — фазе получения. Чтобы зарегистрировать приемник в получателе события для фазы получения процесса диспетчеризации события, мы регистрируем этот приемник с помощью вызова метода `addEventListener( )`, третьему параметру `useCapture` которого устанавливается значение `false`, — точно так же, как мы регистрируем приемник предка для получения уведомлений на фазе всплытия. Этот подход демонстрирует следующий обобщенный код:

```
получательСобытия.addEventListener(событие, приемник, false)
```

Или просто:

```
получательСобытия.addEventListener(событие, приемник)
```

Приведенная строка кода заставляет метод *приемник( )* выполняться всякий раз, когда среда Flash выполняет диспетчеризацию события *событие*, получателем которого является объект *получательСобытия*, после того как предки объекта *получательСобытия* на фазе захвата получат уведомление о возникновении этого события.

При регистрации приемника события непосредственно в получателе события для уведомлений на фазе получения параметр useCapture должен быть всегда установлен в значение false или вообще опущен. В противном случае приемник никогда не будет вызван.

В следующих разделах представлено множество примеров применения параметра `useCapture` и рассматривается несколько вопросов, связанных с регистрацией для получения событий на конкретных фазах.

Регистрация приемника предка для фазы захвата

Как мы уже знаем, чтобы зарегистрировать приемник предка для уведомлений о возникновении события в фазе захвата, мы устанавливаем параметру `useCapture` метода `addEventListener( )` значение `true`, как показано в следующем коде:

```
предок.addEventListener(событие, приемник, true)
```

Теперь используем этот код в работающем примере. Для тестовой иерархии отображения возьмем сценарий, который был изображен на рис. 21.1, — экземпляр класса `Stage` включает в себя объект `Sprite`, который, в свою очередь, содержит объект `TextField`. В листинге 21.1 представлен код для создания этой иерархии.

Листинг 21.1. Тестовая иерархия отображения

```
// Создаем экземпляр класса Sprite
var theSprite:Sprite = new Sprite( );

// Создаем экземпляр класса TextField
var theTextField:TextField = new TextField( );
theTextField.text = "enter input here";
theTextField.autoSize = TextFieldAutoSize.LEFT;
theTextField.type = TextFieldType.INPUT;

// Добавляем объект TextField в объект Sprite
theSprite.addChild(theTextField);

// Добавляем объект Sprite в экземпляр класса Stage. Обратите внимание,
// что объект некийОтображаемыйОбъект должен находиться в списке
// отображения, чтобы иметь доступ к экземпляру класса Stage.
некийОтображаемыйОбъект.stage.addChild(theSprite);
```

Предположим, что мы хотим зарегистрировать функцию `textInputListener( )` в объекте `theSprite` для событий `TextEvent.TEXT_INPUT`. Вот код функции `textInputListener( )`:

```
private function textInputListener (e:TextEvent):void {
  trace("The user entered some text");
}
```

Мы хотим, чтобы функция `textInputListener( )` вызывалась в фазе захвата (то есть до того, как объект `TextField` получит уведомление о возникновении события), поэтому для ее регистрации используем следующий код:

```
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener, true)
```

Предыдущая строка кода заставляет функцию `textInputListener( )` выполняться всякий раз, когда среда Flash осуществляет диспетчеризацию события `TextEvent.TEXT_INPUT`, получателем которого является объект `theTextField`, до того как объект `theTextField` получит уведомление о возникновении этого события.

Регистрация приемника предка для фазы всплытия

Напомним, что для регистрации приемника предка для уведомлений о возникновения события в фазе всплытия мы устанавливаем параметру `useCapture` метода `addEventListener( )` значение `false`, как показано в следующем коде:

```
предок.addEventListener(событие, приемник, false)
```

Продолжая работу с нашим объектом `TextField` из листинга 21.1, предположим, что мы хотим зарегистрировать функцию `textInputListener( )` в объекте

theSprite для событий TextEvent.TEXT_INPUT. При этом мы хотим, чтобы функция textInputListener( ) вызывалась в фазе всплытия (то есть после получения уведомления о возникновении события объектом TextField). Мы используем следующий код:

```
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener, false)
```

Можно сделать то же самое, полностью опустив значение параметра useCapture:

```
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener)
```

Эта строка кода заставляет функцию textInputListener( ) выполняться всегда, когда среда Flash осуществляет диспетчеризацию события TextEvent.TEXT_INPUT, получателем которого является объект theTextField, но после того, как объект theTextField получит уведомление о возникновении этого события.

Обратите внимание, что, если бы событие TextEvent.TEXT_INPUT было невсплывающим, функция textInputListener( ) никогда бы не была вызвана. Стоит еще раз повторить то, о чем мы узнали ранее: если приемник предка регистрируется для невсплывающего события либо с опущенным параметром useCapture, либо с параметром useCapture, которому установлено значение true, этот приемник никогда не будет вызван. Чтобы приемник предка вызывался при диспетчеризации невсплывающего события, он должен быть зарегистрирован для фазы захвата с параметром useCapture, которому установлено значение true.

Чтобы определить, каким является событие, можно воспользоваться любым из следующих способов.

- Обратиться к описанию события в справочнике по языку ActionScript корпорации Adobe.
- Обработать событие с помощью приемника события либо в фазе захвата, либо на фазе получения, и проверить значение переменной bubbles объекта Event, переданного в этот приемник. Если значение переменной bubbles равно true, событие является всплывающим; в противном случае — невсплывающим.

Следующий код демонстрирует последнюю методику:

```
// Регистрируем функцию clickListener( ) в экземпляре класса Stage
// для событий MouseEvent.CLICK. Обратите внимание, что объект
// некийОтображаемыйОбъект должен находиться в списке отображения,
// чтобы иметь доступ к экземпляру класса Stage.
некийОтображаемыйОбъект.stage.addEventListener(MouseEvent.CLICK, clickListener);

// ...далее в коде определяем функцию clickListener( )
private function clickListener (e:MouseEvent):void {
  // Когда возникает событие, проверяем, является ли оно всплывающим
  if (e.bubbles) {
    trace("The MouseEvent.CLICK event is a bubbling event.");
  } else {
    trace("The MouseEvent.CLICK event is a non-bubbling event.");
  }
}
```

Для удобства поиска во всех разделах справочника по языку ActionScript корпорации Adobe, где описываются внутренние события, указано значение переменной экземпляра `bubbles` класса `Event`. Как правило, большинство внутренних событий, получателями которых являются отображаемые объекты, — всплывающие.

Регистрация приемника предка для фазы захвата и фазы всплытия

Чтобы указать, что приемник предка должен вызываться и в фазе захвата, и в фазе всплытия (то есть до и после того, как получатель будет уведомлен о возникновении события), мы должны зарегистрировать этот приемник дважды — один раз параметр `useCapture` должно быть установлено значение `true`, а другой раз — значение `false`. Например, возвращаясь к нашему сценарию с объектом `TextField`, предположим, что мы хотим зарегистрировать наш приемник `textInputListener( )` в объекте `theSprite` для событий `TextEvent.TEXT_INPUT` и чтобы функция `textInputListener( )` вызывалась и в фазе захвата, и в фазе всплытия. Мы используем следующий код:

```
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener, true)
theSprite.addEventListener(TextEvent.TEXT_INPUT, textInputListener, false)
```

Если необходимо сделать так, чтобы приемник предка вызывался и в фазе захвата, и в фазе всплытия процесса диспетчеризации события, он должен быть зарегистрирован для этого события дважды.

Регистрация приемника в получателе события

Напомним, что для регистрации приемника получателя для фазы получения уведомления мы устанавливаем параметру `useCapture` метода `addEventListener( )` значение `false`, как показано в следующем коде:

```
получательСобытия.addEventListener(событие, приемник, false)
```

Таким образом, возвращаясь к нашему текущему сценарию с объектом `TextField`, чтобы зарегистрировать функцию `textInputListener( )` в объекте `theTextField` для событий `TextEvent.TEXT_INPUT`, мы используем следующий код:

```
theTextField.addEventListener(TextEvent.TEXT_INPUT,
                              textInputListener,
                              false)
```

Или просто:

```
theTextField.addEventListener(TextEvent.TEXT_INPUT, textInputListener)
```

Приведенная строка кода заставляет функцию `textInputListener( )` выполняться всякий раз, когда среда Flash осуществляет диспетчеризацию события `TextEvent.TEXT_INPUT`, получателем которого является объект `theTextField`. Функция `textInputListener( )` выполняется после того, как экземпляр класса `Stage` и объект `theSprite` получат уведомления о возникновении события в фазе захвата, но до того, как экземпляр класса `Stage` и объект `theSprite` получат уведомления в фазе всплытия.

Двойное назначение параметра useCapture

Как было показано в двух предыдущих разделах, параметру `useCapture` метода `addEventListener( )` устанавливается значение `false` в двух различных ситуациях:

- когда регистрируемый приемник предка должен вызываться в фазе всплытия;
- когда регистрируемый приемник получателя должен вызываться в фазе получения.

Следовательно, когда при регистрации приемника для события параметр `useCapture` установлено значение `false`, этот приемник будет вызываться в процессе диспетчеризации события, если справедливо любое из следующих условий:

- получателем события является объект, в котором зарегистрирован приемник (в данном случае приемник вызывается в фазе получения);
- получателем события является потомок объекта, в котором зарегистрирован приемник (в данном случае приемник вызывается на фазе всплытия, после того как потомок обработает это событие).

Например, следующий код регистрирует функцию `clickListener( )` в экземпляре класса `Stage` для событий `MouseEvent.CLICK`:

```
некийОтображаемыйОбъект.stage.addEventListener(MouseEvent.CLICK,
                                                clickListener,
                                                false);
```

Поскольку параметру `useCapture` установлено значение `false`, функция `clickListener( )` будет вызываться в следующих ситуациях:

- когда пользователь щелкает кнопкой мыши в области отображения (в этом случае среда Flash выполняет диспетчеризацию события, получателем которого является экземпляр класса `Stage`);
- когда пользователь щелкает кнопкой мыши на любом отображаемом объекте, видимом на экране (в этом случае среда Flash выполняет диспетчеризацию события для объекта-получателя, на котором щелкнули кнопкой мыши, — данный объект всегда является потомком экземпляра класса `Stage`).

Обратите внимание, что, хотя функция `clickListener( )` зарегистрирована в одном объекте (в экземпляре класса `Stage`), на этапе выполнения она может вызываться в процессе диспетчеризации событий, получателями которых является как данный объект, так и потомки данного объекта! По этой причине в некоторых случаях функция приемника должна включать код, игнорирующий процессы диспетчеризации событий, которые ее не интересуют. Мы рассмотрим код, необходимый для игнорирования процессов диспетчеризации событий, далее, в разд. «Отличие событий, получателем которых является некий объект, от событий, получателями которых являются его потомки».

Удаление приемников событий

При отмене регистрации приемника события в объекте, находящемся в иерархии отображения, мы должны указать, для получения уведомлений в какой фазе был

зарегистрирован этот приемник изначально — в фазе захвата или в фазах получения или всплытия. Для этого мы используем третий параметр `useCapture` метода `removeEventListener( )`, который полностью аналогичен параметру `useCapture` метода `addEventListener( )`.

Если приемник, для которого отменяется регистрация, изначально был зарегистрирован для фазы захвата (то есть параметру `useCapture` метода `addEventListener( )` было установлено значение `true`), мы должны отменить его регистрацию, присвоив параметру `useCapture` метода `removeEventListener( )` значение `true`. Если приемник был изначально зарегистрирован для уведомлений на фазах получения или всплытия (то есть параметру `useCapture` метода `addEventListener( )` было установлено значение `false`), мы должны отменить его регистрацию, присвоив параметру `useCapture` метода `removeEventListener( )` значение `false`.

В случае отмены регистрации приемника значение параметра useCapture метода removeEventListener( ) должно всегда соответствовать значению, установленному для параметра useCapture при исходном вызове метода addEventListener( ).

Например, в следующем коде мы регистрируем функцию `clickListener( )` в объекте *некийОтображаемыйОбъект* для фазы захвата, указывая в качестве параметра `useCapture` метода `addEventListener( )` значение `true`:

```
некийОтображаемыйОбъект.addEventListener(MouseEvent.CLICK,
                                          clickListener,
                                          true);
```

Соответственно при отмене регистрации функции `clickListener( )` в объекте *некийОтображаемыйОбъект* мы должны указать значение `true` в качестве параметра `useCapture` метода `removeEventListener( )`:

```
некийОтображаемыйОбъект.removeEventListener(MouseEvent.CLICK,
                                            clickListener,
                                            true);
```

При отмене регистрации приемника события, который был дважды зарегистрирован в одном и том же объекте (для получения уведомлений как в фазе захвата, так и в фазах получения или всплытия), мы должны подобным образом дважды вызвать метод `removeEventListener( )`. Например, следующий код дважды регистрирует приемник события `MouseEvent.CLICK` в экземпляре класса `Stage`, чтобы этот приемник вызывался и в фазе захвата, и в фазах получения или всплытия:

```
некийОтображаемыйОбъект.stage.addEventListener(MouseEvent.CLICK,
                                                clickListener,
                                                true);
некийОтображаемыйОбъект.stage.addEventListener(MouseEvent.CLICK,
                                                clickListener,
                                                false);
```

Следующий код удаляет два предыдущих приемника события `MouseEvent.CLICK`. Поскольку метод `clickListener( )` был зарегистрирован отдельно для фазы захвата и отдельно для фазы получения или всплытия, его регистрация должна также отменяться отдельно для каждой из этих фаз.

```
некийОтображаемыйОбъект.stage.removeEventListener(MouseEvent.CLICK,
                                                   clickListener,
                                                   true);
некийОтображаемыйОбъект.stage.removeEventListener(MouseEvent.CLICK,
                                                   clickListener,
                                                   false);
```

Поскольку каждая регистрация приемника с помощью метода addEventListener( ) считается отдельной операцией, отмена каждой регистрации должна осуществляться соответствующим вызовом метода removeEventListener( ).

Теперь, когда мы получили общее представление о цепочке диспетчеризации события, рассмотрим пример, который демонстрирует, как цепочка диспетчеризации события может помочь централизовать код в реальном приложении.

Использование цепочки диспетчеризации событий для централизации кода

Ожидая появления свободной комнаты в полностью забронированном отеле, проще попросить управляющего гостиницей сообщить вам, когда освободится комната, чем спрашивать каждого постояльца о том, когда он планирует уезжать. Подобным образом при обработке диспетчеризаций событий зачастую бывает гораздо эффективнее зарегистрировать приемники событий в контейнере отображаемых объектов, чем регистрировать приемники в каждом из потомков этого контейнера.

Предположим, что мы создаем простой элемент управления «флажок», состоящий из двух следующих классов:

- `CheckBox` — подкласс класса `Sprite`, который выступает в роли контейнера для всего элемента управления;
- `CheckBoxIcon` — подкласс класса `Sprite`, который представляет графический значок флажка.

На этапе выполнения каждый экземпляр класса `CheckBox` создает два дочерних объекта: экземпляр класса `CheckBoxIcon` для значка флажка и экземпляр класса `TextField` для текстовой надписи флажка. Для справочных целей назовем экземпляр основного класса `CheckBox` именем `container`, а два его дочерних объекта — `icon` и `label`. На рис. 21.3 представлена схема нашего элемента управления «флажок».

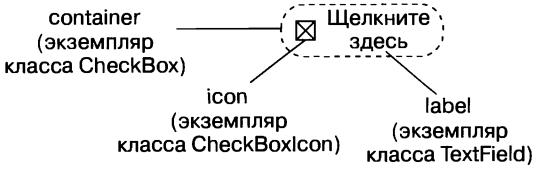

Рис. 21.3. Объекты элемента управления «флажок»

Мы хотим максимально упростить использование нашего флажка, поэтому проектируем его таким образом, чтобы состояние флажка — установлен или снят — изменялось в тот момент, когда пользователь щелкает кнопкой мыши на значке флажка или на его надписи. Соответственно, в нашей реализации мы должны отслеживать события щелчка кнопкой мыши, получателями которых являются объекты `icon` и `label`.

Мы могли бы зарегистрировать отдельные приемники для события щелчка кнопкой мыши в каждом из указанных объектов. Однако регистрация двух приемников событий приведет к увеличению длительности разработки и из-за повторения практически идентичного кода для регистрации приемников увеличит вероятность появления ошибок в нашем флажке. Вместо этого, чтобы избежать появления повторяющегося кода, мы можем обрабатывать все события щелчка кнопкой мыши в одном приемнике, зарегистрированном в объекте `container`. Поскольку объект `container` является отображаемым предком и для объекта `icon`, и для объекта `label`, он получает уведомления всякий раз, когда среда Flash выполняет диспетчеризацию события щелчка кнопкой мыши, получателем которого является любой из этих объектов. Когда выполняется код приемника объекта `container`, обрабатывающий события щелчка кнопкой мыши, мы знаем, что пользователь щелкнул либо на значке, либо на надписи, и в ответ на это действие мы можем изменить состояние флажка — установить или снять.

В листинге 21.2 представлен код для нашего примера с флажком. Части кода, связанные с обработкой событий, выделены полужирным шрифтом.

Листинг 21.2. Иерархическая обработка событий класса CheckBox

```
// Файл CheckBox.as
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  // Очень простой элемент управления «флажок»
  public class CheckBox extends Sprite {
    private var label:TextField;     // Текстовая надпись флажка
    private var icon:CheckBoxIcon;   // Графический значок флажка
    private var checked:Boolean;     // Флаг, который указывает, установлен
                                     // ли флажок в настоящий момент
    // Конструктор
    public function CheckBox (msg:String) {
      // При создании объекта флажок не установлен
      checked = false;

      // Создаем графический значок
      icon = new CheckBoxIcon( );

      // Создаем текстовую надпись
      label = new TextField( );
      label.text = msg;
```

```
      label.autoSize = TextFieldAutoSize.LEFT;
      label.selectable = false;

      // Размещаем текстовую надпись рядом с графическим значком
      label.x = icon.x + icon.width + 5;

      // Добавляем объекты label и icon к данному объекту в качестве
      // его отображаемых детей
      addChild(icon);
      addChild(label);

      // Регистрируем приемник для получения событий щелчка кнопкой мыши,
      // получателем которых является данный объект или любой
      // из его дочерних объектов (то есть label или icon)
      addEventListener(MouseEvent.CLICK, clickListener);
    }

    // Обрабатывает события щелчка кнопкой мыши. Этот метод выполняется
    // всякий раз, когда пользователь щелкает на объекте label или icon.
    private function clickListener (e:MouseEvent):void {
      if (checked) {
        icon.uncheck( );
        checked = false;
      } else {
        icon.check( );
        checked = true;
      }
    }
  }
}

// Файл CheckBoxIcon.as
package {
  import flash.display.*;

  // Графический значок для элемента управления «флажок»
  public class CheckBoxIcon extends Sprite {

    // Конструктор
    public function CheckBoxIcon ( ) {
      uncheck( );
    }

    // Рисует значок флажка в состоянии «установлен»
    public function check ( ):void {
      graphics.clear( );
      graphics.lineStyle(1);
      graphics.beginFill(0xFFFFFF);
      graphics.drawRect(0, 0, 15, 15);
      graphics.endFill( );
      graphics.lineTo(15, 15);
```

```
      graphics.moveTo(0, 15);
      graphics.lineTo(15, 0);
    }

    // Рисует значок флажка в состоянии «снят»
    public function uncheck ( ):void {
      graphics.clear( );
      graphics.lineStyle(1);
      graphics.beginFill(0xFFFFFF);
      graphics.drawRect(0, 0, 15, 15);
    }
  }
}

// Файл CheckBoxDemo.as (основной класс, который использует класс CheckBox)
package {
  import flash.display.Sprite;

  // Демонстрирует использование класса CheckBox
  public class CheckboxDemo extends Sprite {
    public function CheckboxDemo( ) {
      var c:CheckBox = new CheckBox("Click here");
      addChild(c);
    }
  }
}
```

Мы познакомились с основами системы иерархической диспетчеризации событий языка ActionScript, но нам осталось рассмотреть еще несколько тем. Не будем сдаваться.

Определение текущей фазы события

Как уже известно из подразд. «Регистрация приемника предка для фазы захвата и фазы всплытия» разд. «Приемники событий и цепочка диспетчеризации событий», дважды вызвав метод `addEventListener( )`, можно зарегистрировать одну и ту же функцию-приемник события как для фазы захвата, так и для фазы всплытия процесса диспетчеризации события. В подразд. «Двойное назначение параметра useCapture» разд. «Приемники событий и цепочка диспетчеризации событий» мы также узнали, что приемник события, при регистрации которого параметру `useCapture` было установлено значение `false`, может вызываться как в фазе получения, так и в фазе всплытия диспетчеризации события. Следовательно, когда в ответ на возникшее событие выполняется функция-приемник события, текущая фаза события будет известна не всегда. В соответствии с этим язык ActionScript предоставляет переменную экземпляра `eventPhase` класса `Event`, которая может быть использована внутри функции-приемника события для определения текущей фазы события.

Переменная `eventPhase` содержит информацию о том, в какой фазе находится текущий процесс диспетчеризации события — захвата, получения или всплытия.

Когда процесс диспетчеризации события находится в фазе захвата, переменной `eventPhase` присваивается значение `EventPhase.CAPTURING_PHASE`, которое означает, что объект-получатель еще не получил уведомление о возникновении события.

Когда процесс диспетчеризации события находится в фазе получения, переменной `eventPhase` присваивается значение `EventPhase.AT_TARGET`, которое означает, что в настоящий момент объект-получатель обрабатывает событие.

Когда процесс диспетчеризации события находится в фазе всплытия, переменной `eventPhase` присваивается значение `EventPhase.BUBBLING_PHASE`, которое означает, что объект-получатель завершил обработку события.

Обычно константам `EventPhase.CAPTURING_PHASE`, `EventPhase.AT_TARGET` и `EventPhase.BUBBLING_PHASE` присваиваются значения `1`, `2` и `3` соответственно, но эти значения со временем могут измениться, поэтому использовать их непосредственно в коде не следует. Вместо этого, чтобы определить текущую фазу события внутри функции-приемника события, всегда сравнивайте значение переменной `eventPhase` с константами класса `EventPhase`. Например, всегда используйте код наподобие следующего:

```
private function someListener (e:Event):void {
  if (e.eventPhase == EventPhase.AT_TARGET) {
    // Этот приемник был вызван в фазе получения...
  }
}
```

И никогда не используйте код наподобие следующего:

```
private function someListener (e:Event):void {
  // Плохой код! Никогда не используйте в коде
  // непосредственные значения
  // констант класса EventPhase!
  if (e.eventPhase == 2) {
    // Этот приемник был вызван в фазе получения...
  }
}
```

Следующий код демонстрирует общий подход к использованию переменной `eventPhase`. Сначала в экземпляр класса `Stage` добавляется объект `TextField`. Затем в экземпляре класса `Stage` регистрируется приемник `clickListener( )` для получения уведомлений о возникновении события `MouseEvent.CLICK` в фазе захвата.

Наконец, в экземпляре класса `Stage` регистрируется приемник `clickListener( )` для получения уведомлений о возникновении события `MouseEvent.CLICK` в фазе получения и в фазе всплытия.

При выполнении приемника `clickListener( )` на консоль выводится текущая фаза. Обратите внимание, что она определяется путем сравнения значения переменной `eventPhase` с тремя константами класса `EventPhase`.

```
var t:TextField = new TextField( );
t.text = "click here";
```

```
t.autoSize = TextFieldAutoSize.LEFT;
stage.addChild(t);

// Регистрируем приемник для фазы захвата
stage.addEventListener(MouseEvent.CLICK, clickListener, true);

// Регистрируем приемник для фазы получения или всплытия
stage.addEventListener(MouseEvent.CLICK, clickListener, false);

// ...где-то в классе
private function clickListener (e:MouseEvent):void {
  var phase:String;
  switch (e.eventPhase) {
    case EventPhase.CAPTURING_PHASE:
    phase = "Capture";
    break;

    case EventPhase.AT_TARGET:
    phase = "Target";
    break;

    case EventPhase.BUBBLING_PHASE:
    phase = "Bubbling";
    break;
  }
  trace("Current event phase: " + phase);
}
```

Если при выполнении предыдущего кода пользователь щелкнет кнопкой мыши на объекте `TextField`, то среда Flash осуществит диспетчеризацию события `MouseEvent.CLICK`, получателем которого является объект `TextField`, и в результате будет выведена следующая информация:

```
Current event phase: Capture
Current event phase: Bubbling
```

Не забывайте, что приемник `clickListener( )` был зарегистрирован в экземпляре класса `Stage` и для фазы захвата, и для фазы всплытия, поэтому в процессе диспетчеризации событий, получателями которых являются потомки экземпляра класса `Stage`, он вызывается дважды.

С другой стороны, если пользователь щелкнет кнопкой мыши в области отображения, среда Flash выполнит диспетчеризацию события `MouseEvent.CLICK`, получателем которого является объект `Stage`, и в результате будет выведена следующая информация:

```
Current event phase: Target
```

Как будет рассмотрено в следующем разделе, переменная `eventPhase` обычно используется для отличия событий, получателем которых является некий объект, от событий, получателями которых являются потомки этого объекта. Реже переменная `eventPhase` применяется для отличия фазы захвата от фазы всплытия внутри приемников предка, зарегистрированных для обеих фаз.

Отличие событий, получателем которых является некий объект, от событий, получателями которых являются его потомки

Когда переменной `eventPhase` объекта `Event`, передаваемого в функцию-приемник, присвоено значение `EventPhase.AT_TARGET`, мы знаем, что получателем события является объект, в котором зарегистрирован данный приемник. С другой стороны, когда переменной `eventPhase` присвоено значение `EventPhase.CAPTURING_PHASE` или `EventPhase.BUBBLING_PHASE`, мы знаем, что получателем события является *потомок* объекта, в котором зарегистрирован данный приемник.

Таким образом, приемник может использовать следующий код, чтобы игнорировать события, получателями которых являются потомки объекта, зарегистрировавшего этот приемник:

```
private function некийПриемник (e:НекоеСобытие):void {
  if (e.eventPhase == EventPhase.AT_TARGET) {
    // Этот код выполняется только в том случае, если получателем события
    // является объект, зарегистрировавший данный приемник.
  }
}
```

Мы можем использовать предыдущую методику, чтобы написать код, который реагирует на события ввода, получаемые неким конкретным объектом, но не его потомками. Например, представьте приложение, в котором экземпляр класса `Stage` содержит множество кнопок, текстовых полей и других объектов для ввода данных. Чтобы реагировать только на щелчки кнопкой мыши, которые происходят над незаполненными областями экземпляра класса `Stage`, мы используем следующий код:

```
// Регистрируем приемник в экземпляре класса Stage для событий
// MouseEvent.CLICK. В результате метод clickListener( ) будет вызываться
// всегда, когда щелкают на *любом* объекте в списке отображения.
stage.addEventListener(MouseEvent.CLICK, clickListener);

// ...где-то в классе определяем приемник события MouseEvent.CLICK
private function clickListener (e:MouseEvent):void {
  // Если этот приемник был вызван в фазе получения...
  if (e.eventPhase == EventPhase.AT_TARGET) {
    // ...значит щелкнули на экземпляре класса Stage. Продолжаем выполнение
    // кода для ситуации «Щелкнули на экземпляре класса Stage».
  }
}
```

Чтобы создать приемник, игнорирующий события, получателем которых является зарегистрировавший его объект, мы используем следующий код:

```
private function некийПриемник (e:некоеСобытие):void {
  if (e.eventPhase != EventPhase.AT_TARGET) {
    // Этот код выполняется только в том случае, если получателем события
    // является потомок объекта, в котором был зарегистрирован данный
    // приемник.
  }
}
```

Например, следующий приемник реагирует на щелчки кнопкой мыши, которые происходят над любым объектом в списке отображения, но не над пустой областью экземпляра класса `Stage`:

```
// Регистрируем приемник в экземпляре класса Stage для событий
// MouseEvent.CLICK
stage.addEventListener(MouseEvent.CLICK, clickListener);

// ...где-то в классе определяем приемник события MouseEvent.CLICK
private function clickListener (e:SomeInputEvent):void {
  // Если этот приемник не был вызван на фазе получения...
  if (e.eventPhase != EventPhase.AT_TARGET) {
    // ...получателем события является потомок экземпляра класса Stage.
    // Следовательно, в момент щелчка указатель мыши находился над одним
    // из объектов в списке отображения, но не над пустой областью
    // экземпляра класса Stage.
  }
}
```

Перейдем к следующей теме, относящейся к процессу иерархической диспетчеризации событий, — рассмотрим остановку процесса диспетчеризации события.

Остановка процесса диспетчеризации события

В любой точке цепочки диспетчеризации события каждый приемник события — включая приемники, зарегистрированные в объекте-получателе, и приемники, зарегистрированные в предках объекта-получателя, — может остановить весь процесс диспетчеризации. Остановка процесса диспетчеризации события называется *поглощением события*.

Чтобы остановить процесс диспетчеризации события, мы вызываем метод экземпляра `stopImmediatePropagation( )` или `stopPropagation( )` класса `Event` над объектом `Event`, передаваемым в функцию-приемник. Метод `stopImmediatePropagation( )` останавливает процесс диспетчеризации события немедленно, не позволяя вызвать оставшиеся приемники; метод `stopPropagation( )` останавливает процесс диспетчеризации события после того, как среда Flash вызовет оставшиеся приемники, зарегистрированные в объекте, который на настоящий момент получил уведомление о возникновении события.

Предположим, что у нас есть экземпляр класса `Sprite` — `container`, который содержит объект `TextField` — `child`:

```
var container:Sprite = new Sprite( );
var child:TextField = new TextField( );
child.text = "click here";
child.autoSize = TextFieldAutoSize.LEFT;
container.addChild(child);
```

Предположим также, что у нас есть три функции-приемника: `containerClickListenerOne( )`, `containerClickListenerTwo( )` и `childClickListener( )`. Регистрируем функции `containerClickListenerOne( )` и `containerClickListenerTwo( )` в объекте `container` для получения уведомлений о возникновении события `MouseEvent.CLICK` в фазе захвата:

```
container.addEventListener(MouseEvent.CLICK,
                           containerClickListenerOne,
                           true);
container.addEventListener(MouseEvent.CLICK,
                           containerClickListenerTwo,
                           true);
```

После этого регистрируем функцию `childClickListener( )` в объекте `child` для уведомлений о возникновении события `MouseEvent.CLICK` в фазе получения:

```
child.addEventListener(MouseEvent.CLICK, childClickListener, false);
```

В обычной ситуации, когда пользователь щелкает кнопкой мыши на объекте `child`, вызываются все три приемника события: два в фазе захвата и один в фазе получения. Если тем не менее функция `containerClickListenerOne( )` поглотит событие путем вызова метода `stopImmediatePropagation( )`, ни `containerClickListenerTwo( )`, ни `childClickListener( )` вызваны не будут.

```
public function containerClickListenerOne (e:MouseEvent):void {
  // Исключаем получение события функциями containerClickListenerTwo( )
  // и childClickListener( )
  e.stopImmediatePropagation( );
}
```

С другой стороны, если функция `containerClickListenerOne( )` поглотит событие путем вызова метода `stopPropagation( )` вместо метода `stopImmediatePropagation( )`, то перед остановкой процесса диспетчеризации события будут вызваны оставшиеся приемники события `MouseEvent.CLICK` объекта `container`. Следовательно, функция `containerClickListenerTwo( )` получит событие, а функция `childClickListener( )` — нет.

```
public function containerClickListenerOne (e:MouseEvent):void {
  // Исключаем получение события только функцией childClickListener( )
  e.stopPropagation( );
}
```

Обратите внимание, что предыдущий пример основывается на условии, что функция `containerClickListenerOne( )` была зарегистрирована перед функцией `containerClickListenerTwo( )`. Дополнительную информацию о порядке, в котором вызываются приемники событий, можно найти в гл. 12.

Поглощение событий обычно происходит для того, чтобы остановить или переопределить стандартную реакцию программы на событие. Предположим, что подкласс

ToolPanel класса Sprite содержит группу элементов управления интерфейса, каждый из которых позволяет вводить данные. Класс ToolPanel имеет два рабочих состояния: активен и неактивен. Когда объект ToolPanel неактивен, пользователь не должен иметь возможность взаимодействовать с любым из его вложенных элементов управления интерфейса.

Для реализации состояния «неактивен» в каждом объекте ToolPanel регистрируется метод clickListener( ) для уведомления о возникновении события MouseEvent.CLICK в фазе захвата. Когда объект ToolPanel неактивен, метод clickListener( ) останавливает все события щелчка кнопкой мыши до того, как они достигнут дочерних объектов Tool. В листинге 21.3 представлен класс ToolPanel, который был сильно упрощен, чтобы акцентировать внимание на коде, поглощающем событие (выделен полужирным шрифтом). В этом листинге дочерние элементы управления интерфейса класса ToolPanel являются экземплярами базового класса Tool, который отсутствует в листинге. Однако в реальном приложении эти элементы управления могут быть кнопками, меню или любыми другими видами интерактивных инструментов.

Листинг 21.3. Поглощение события

```
package {
  import flash.display.Sprite;
  import flash.events.*;

  public class ToolPanel extends Sprite {
    private var enabled:Boolean;

    public function ToolPanel ( ) {
      enabled = false;

      var tool1:Tool = new Tool( );
      var tool2:Tool = new Tool( );
      var tool3:Tool = new Tool( );

      tool2.x = tool1.width + 10;
      tool3.x = tool2.x + tool2.width + 10;

      addChild(tool1);
      addChild(tool2);
      addChild(tool3);

      // Регистрируем приемник в этом объекте для уведомлений
      // о возникновении события MouseEvent.CLICK в фазе захвата
      addEventListener(MouseEvent.CLICK, clickListener, true);
    }

    private function clickListener (e:MouseEvent):void {
      // Если данный объект ToolPanel неактивен...
      if (!enabled) {
        // ...останавливаем процесс диспетчеризации данного события -
        // щелчка кнопкой мыши, чтобы оно не достигло потомков данного
        // объекта ToolPanel
```

```
            e.stopPropagation( );
            trace("Panel disabled. Click event dispatch halted.");
        }
    }
  }
}
```

В повседневной разработке приложений метод `stopPropagation( )` используется гораздо более часто, чем `stopImmediatePropagation( )`. Тем не менее второй метод применяется в следующих ситуациях:

- когда объект-получатель желает, чтобы его собственные приемники не вызывались в результате возникновения события;
- когда программа не желает, чтобы приемники реагировали на данное событие.

Рассмотрим пример каждой из описанных ситуаций, начав с объекта-получателя, который желает, чтобы его собственные приемники реагировали на событие. Представьте игру «космический шутер», которая включает следующие классы:

- `GameManager` — класс, управляющий процессом игры;
- `PlayerShip` — класс, который представляет космический корабль игрока.

Класс `GameManager` определяет собственное событие `GameManager.SHIP_HIT`, диспетчеризация которого осуществляется всякий раз, когда вражеская ракета попадает в корабль игрока. Получателем каждого события `GameManager.SHIP_HIT` является объект `PlayerShip`. Этот объект регистрирует два приемника, которые реагируют на события `GameManager.SHIP_HIT`. Один приемник проигрывает анимацию, имитирующую взрыв, а другой приемник воспроизводит звук взрыва.

Когда корабль игрока погибает, создается новый корабль, и на протяжении пяти секунд он остается неуязвимым. До тех пор пока корабль остается неуязвимым, приемники события `GameManager.SHIP_HIT` объекта `PlayerShip` не должны воспроизводить анимацию или звук «повреждения корабля».

Чтобы исключить выполнение приемников события `GameManager.SHIP_HIT` до тех пор, пока корабль остается неуязвимым, класс `PlayerShip` регистрирует третий приемник `hitListener( )`, предназначенный для поглощения событий `GameManager.SHIP_HIT` на основании текущего статуса корабля (неуязвим или уязвим). Регистрация метода `hitListener( )` с приоритетом `int.MAX_VALUE` происходит в конструкторе класса `PlayerShip`, как показано в следующем коде:

```
public class PlayerShip {
  public function PlayerShip ( ) {
    addEventListener(GameManager.HIT, hitListener, false, int.MAX_VALUE);
  }
}
```

В гл. 12 рассказывалось, что по умолчанию приемники событий объекта вызываются в том порядке, в котором они были зарегистрированы. Мы также знаем, что этот используемый по умолчанию «порядок вызова» может быть перекрыт с помощью параметра `priority` метода `addEventListener( )`.

Поскольку при регистрации метода `hitListener( )` в конструкторе класса `PlayerShip` используется максимально возможный приоритет, из всех прием-

ников события GameManager.SHIP_HIT класса PlayerShip этот приемник всегда будет вызываться первым. Если объект PlayerShip является уязвимым и возникает событие GameManager.SHIP_HIT, метод hitListener( ) не выполняет никаких действий. Но если объект PlayerShip не является уязвимым и возникает событие GameManager.SHIP_HIT, метод hitListener( ) сначала поглощает это событие, а затем выполняет диспетчеризацию нового события PlayerShip.HIT_DEFLECTED. После этого приемники, зарегистрированные для события PlayerShip.HIT_DEFLECTED, воспроизводят специальную анимацию и звук, которые сообщают о том, что корабль не был поврежден.

Далее представлен код метода hitListener( ); обратите внимание на использование метода stopImmediatePropagation( ):

```
private function hitListener (e:Event):void {
  if (invincible) {
    // Исключаем получение другими приемниками объекта PlayerShip
    // уведомления о возникновении события
    e.stopImmediatePropagation( );
    // Рассылаем новое событие
    dispatchEvent(new Event(PlayerShip.HIT_DEFLECTED, true));
  }
}
```

В предыдущем сценарии с классом PlayerShip стоит отметить, что, хотя объект PlayerShip может предотвратить вызов своих собственных приемников события GameManager.SHIP_HIT, он не может предотвратить вызов приемников события GameManager.SHIP_HIT, зарегистрированных в его отображаемых предках. В частности, любые приемники, зарегистрированные для фазы захвата в отображаемых предках объекта PlayerShip, будут всегда получать уведомление о возникновении события GameManager.SHIP_HIT, даже если объект PlayerShip поглотит это событие. Однако после того, как объект PlayerShip поглотит событие GameManager.SHIP_HIT, его предки не получат уведомление в фазе всплытия.

Теперь перейдем ко второму сценарию с использованием метода stopImmediate-Propagation( ), в котором программа желает, чтобы никакие приемники не отвечали на данное событие. Предположим, мы создаем набор компонент пользовательского интерфейса, которые автоматически переходят в неактивное состояние, когда среда выполнения теряет фокус операционной системы, и возвращаются в активное состояние, когда среда получает фокус операционной системы. Чтобы определить момент получения или потери фокуса операционной системы, наши компоненты регистрируют приемники для внутренних событий Event.ACTIVATE и Event.DEACTIVATE (дополнительные сведения об этих событиях можно найти в гл. 22).

Предположим, мы разрабатываем тестовое приложение для выполнения нагрузочных испытаний наших компонентов. Наше приложение программным путем вызывает интерактивное поведение компонентов. В тесте мы должны гарантировать, что внутреннее событие Event.DEACTIVATE не переводит наши тестируемые компоненты в неактивное состояние. В противном случае наше тестовое приложение не сможет работать с ними программным путем. Таким образом, в конструкторе основного класса нашего текстового приложения мы регистрируем приемник для события Event.DEACTIVATE в экземпляре класса Stage. Этот приемник использует

метод stopImmediatePropagation( ) для поглощения всех внутренних событий Event.DEACTIVATE, как показано в следующем коде:

```
private function deactivateListener (e: Event):void {
  e.stopImmediatePropagation( );
}
```

Поскольку разрабатываемое нами тестовое приложение поглощает все события Event.DEACTIVATE, компоненты никогда не получат уведомления о возникновении события Event.DEACTIVATE и, следовательно, никогда не перейдут в неактивное состояние в ответ на потерю фокуса средой Flash. В процессе тестирования оператор тестового приложения сможет в любой момент устанавливать или убирать фокус со среды выполнения, не оказывая никакого влияния на способность тестового приложения управлять компонентами программным путем.

Приоритет и цепочка диспетчеризации событий

Когда приемник события регистрируется в объекте, находящемся в иерархии отображения, параметр priority влияет на порядок вызова только тех приемников, которые зарегистрированы в этом объекте. Параметр priority не изменяет и не может изменить порядок, в котором происходит уведомление объектов в цепочке диспетчеризации событий.

Не существует способа сделать так, чтобы приемник одного объекта в цепочке диспетчеризации события вызывался до или после приемника другого объекта в той же цепочке.

Например, представьте объект Sprite, содержащий объект TextField. Объект Sprite регистрирует приемник spriteClickListener( ) для события MouseEvent.CLICK — параметру useCapture установлено значение false, а параметру priority присвоено значение 2:

```
объектSprite.addEventListener(MouseEvent.CLICK, spriteClickListener, false, 2)
```

Точно так же объект TextField регистрирует приемник textClickListener( ) для события MouseEvent.CLICK — значение параметра useCapture установлено в false, а параметру priority присвоено значение 1:

```
объектTextField.addEventListener(MouseEvent.CLICK, textClickListener, false, 1)
```

Когда пользователь щелкает кнопкой мыши на объекте TextField, среда Flash выполняет диспетчеризацию события MouseEvent.CLICK, получателем которого является объект TextField. В результате метод textClickListener( ) вызывается в фазе получения перед методом spriteClickListener( ), который вызывается в фазе всплытия. Два приемника события вызываются в порядке, устанавливаемом цепочкой диспетчеризации события, даже несмотря на то, что метод spriteClickListener( ) был зарегистрирован с более высоким приоритетом, чем textClickListener( ).

Дополнительную информацию о приоритетах событий можно найти в гл. 12.

Изменение иерархии отображения и цепочка диспетчеризации событий

Непосредственно перед тем, как приступить к диспетчеризации заданного события, среда выполнения Flash заранее определяет для него всю цепочку диспетчеризации, учитывая текущее состояние иерархии отображения его получателя. Другими словами, список объектов, которые будут уведомлены о возникновении события, и порядок уведомления этих объектов определяются заранее — до того как начнется процесс диспетчеризации события. После начала процесса диспетчеризации уведомление объектов о возникновении события осуществляется в соответствии с этим предопределенным порядком, даже если в процессе диспетчеризации структура иерархии отображения объекта-получателя будет изменена приемниками событий.

Предположим, у нас есть экземпляр класса `TextField`, содержащийся в экземпляре класса `Sprite`, который, в свою очередь, содержится в экземпляре класса `Stage`. Пусть также мы регистрируем приемник `stageClickListener( )` в экземпляре класса `Stage` для уведомления о возникновении события `MouseEvent.CLICK` в фазе захвата, как показано в следующем коде:

```
stage.addEventListener(MouseEvent.CLICK, stageClickListener, true);
```

Наконец, предположим, что зарегистрированная функция `stageClickListener( )` содержит код, который удаляет объект `TextField` из его родительского объекта `Sprite`, как показано в следующем коде:

```
private function stageClickListener (e:MouseEvent):void {
  // Если щелкнули кнопкой мыши на объекте TextField...
  if (e.target == textField) {
    // ...то удаляем его
    removeChild(textField);
    textField = null;
  }
}
```

Когда пользователь щелкает кнопкой мыши в текстовом поле, среда Flash выполняет диспетчеризацию события `MouseEvent.CLICK`, получателем которого является объект `TextField`. Перед началом процесса диспетчеризации среда выполнения заранее определяет всю цепочку диспетчеризации события, как показано ниже:

```
ФАЗА ЗАХВАТА:    (1) Объект Stage
                 (2) Объект Sprite
ФАЗА ПОЛУЧЕНИЯ: (3) Объект TextField
ФАЗА ВСПЛЫТИЯ:  (4) Объект Sprite
                 (5) Объект Stage
```

Когда начинается процесс диспетчеризации, среда Flash сначала уведомляет о возникновении события объект `Stage` (1). В результате этого уведомления вызывается приемник `stageClickListener( )` объекта `Stage`, который удаляет объект `TextField` из списка отображения. Далее, несмотря на то, что объект `Sprite` больше не является потомком объекта `TextField`, среда выполнения уведомляет о возникновении события объект `Sprite` (2). Затем, несмотря на то

что объекта TextField больше нет в списке отображения, среда выполнения уведомляет о возникновении события объект TextField (3). Наконец, в фазе всплытия среда Flash снова уведомляет о возникновении события объекты Sprite (4) и Stage (5). Даже несмотря на то, что иерархия отображения получателя события была изменена в процессе диспетчеризации, событие все равно проходит по всей предопределенной цепочке диспетчеризации события.

Как только для данного события будет установлена цепочка диспетчеризации, она останется неизменной до окончания этого процесса диспетчеризации.

В листинге 21.4 представлен код для приведенного сценария в контексте тестового класса DisappearingTextField. В этом листинге экземпляр пользовательского класса DisappearingTextField (подкласс класса Sprite) играет роль объекта Sprite из описанного сценария.

Листинг 21.4. Неизменяемая цепочка диспетчеризации события

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  public class DisappearingTextField extends Sprite {
    private var textField:TextField;
    public function DisappearingTextField ( ) {
      // Создаем объект TextField
      textField = new TextField( );
      textField.text = "Click here";
      textField.autoSize = TextFieldAutoSize.LEFT;

      // Добавляем объект TextField в объект DisappearingTextField
      addChild(textField);

      // Регистрируем приемник в экземпляре класса Stage для событий
      // MouseEvent.CLICK
      stage.addEventListener(MouseEvent.CLICK, stageClickListener, true);

      // Чтобы доказать, что объект TextField получает уведомление
      // о возникновении события MouseEvent.CLICK даже после того, как он
      // будет удален из данного экземпляра класса DisappearingTextField,
      // регистрируем приемник в объекте TextField для событий
      // MouseEvent.CLICK
      textField.addEventListener(MouseEvent.CLICK, textFieldClickListener);
    }

    // Эта функция выполняется, когда пользователь щелкает кнопкой мыши
    // на любом объекте в списке отображения
    private function stageClickListener (e:MouseEvent):void {
      // Если щелкнули кнопкой мыши на объекте TextField...
      if (e.target == textField) {
        // ...то удаляем его
```

```
        removeChild(textField);
        textField = null;
      }
    }

    // Эта функция выполняется, когда пользователь щелкает кнопкой мыши
    // на объекте TextField
    private function textFieldClickListener (e:MouseEvent):void {
      // На данный момент метод stageClickListener( )
      // уже удалил объект TextField,
      // однако приемник все равно вызывается.
      trace("textFieldClickListener triggered");
    }
  }
}
```

Изменение списка приемников события. Как мы уже знаем, когда возникает некоторое событие, среда выполнения уведомляет надлежащие объекты в соответствии с предопределенным порядком, устанавливаемым цепочкой диспетчеризации события. В свою очередь, когда каждый объект получает уведомление о возникновении события, вызываются приемники события этого объекта. Список конкретных приемников, вызываемых при возникновении данного события, определяется непосредственно перед началом процесса уведомления этого объекта. Как только начнется процесс уведомления, изменить этот список приемников будет невозможно.

Например, рассмотрим процесс диспетчеризации события `MouseEvent.CLICK`, получателем которого является объект `Sprite`, содержащийся в экземпляре класса `Stage`. Цепочка диспетчеризации события включает в себя три уведомления о возникновении события, как показано ниже:

```
ФАЗА ЗАХВАТА:   (1) Объект Stage уведомлен
ФАЗА ПОЛУЧЕНИЯ: (2) Объект Sprite уведомлен
ФАЗА ВСПЛЫТИЯ:  (3) Объект Stage уведомлен
```

Предположим, что на этапе первого уведомления (1) код в приемнике объекта `Stage` регистрирует новый приемник для события `MouseEvent.CLICK` в объекте `Sprite`. Поскольку событие еще не было передано в объект `Sprite`, новый приемник будет вызван на этапе второго уведомления (2).

Теперь предположим, что на этапе первого уведомления (1) код в приемнике объекта `Stage` регистрирует новый приемник для события `MouseEvent.CLICK` в экземпляре класса `Stage`, который будет вызываться в фазе всплытия. Поскольку этап первого уведомления (1) уже начался, список приемников экземпляра класса `Stage` уже заморожен, поэтому новый приемник не будет вызван на этапе первого уведомления (1). Тем не менее новый приемник появится в цепочке диспетчеризации события и будет вызван на этапе третьего уведомления (3).

Наконец, предположим, что на этапе второго уведомления (2) код в приемнике объекта `Sprite` отменяет регистрацию существующего приемника для события `MouseEvent.CLICK` в объекте `Sprite`. Поскольку этап второго уведомле-

ния (2) уже начался, его список приемников был заморожен, поэтому удаленный приемник будет все равно вызван на этапе второго уведомления. Конечно, когда возникнет другое событие `MouseEvent.CLICK`, удаленный приемник не будет вызываться.

В любой момент процесса диспетчеризации данного события список вызываемых приемников на текущем этапе уведомления изменить невозможно, однако можно изменить список приемников, вызываемых на дальнейших этапах уведомления цепочки диспетчеризации события.

Пользовательские события и цепочка диспетчеризации события

Иерархическая событийная система языка ActionScript применяется ко всем событиям, получателями которых являются отображаемые объекты, — даже к тем событиям, которые генерируются вручную программистом. Когда получателем пользовательского события является объект в иерархии отображения, предки этого объекта получат уведомление о возникновении данного события.

Обобщенный код, показанный в листинге 21.5, демонстрирует, как пользовательские события, аналогично внутренним событиям, передаются по цепочке диспетчеризации события. В этом листинге тестовый класс `CustomEventDemo` говорит среде Flash выполнить диспетчеризацию пользовательского события, получателем которого является объект `Sprite` в списке отображения.

Листинг 21.5. Пользовательское событие, передаваемое по цепочке диспетчеризации

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  public class CustomEventDemo extends Sprite {
    public static const SOME_EVENT:String = "SOME_EVENT";

    public function CustomEventDemo ( ) {
      var sprite:Sprite = new Sprite( );
      addChild(sprite);

      // Регистрируем приемник someEventListener( ) в экземпляре класса Stage
      // для уведомлений о возникновении события CustomEventDemo.SOME_EVENT.
      stage.addEventListener(CustomEventDemo.SOME_EVENT, someEventListener);

      // Осуществляем диспетчеризацию события CustomEventDemo.SOME_EVENT,
      // получателем которого является объект в списке отображения.
      // Второй параметр конструктора класса Event устанавливаем в значение
      // true, чтобы у события появилась фаза всплытия.
      sprite.dispatchEvent(new Event(CustomEventDemo.SOME_EVENT, true));
    }
```

```
    private function someEventListener (e:Event):void {
      trace("SOME_EVENT occurred.");
    }
  }
}
```

В результате вызова метода `dispatchEvent( )` из листинга 21.5 среда выполнения Flash осуществит диспетчеризацию события `CustomEventDemo.SOME_EVENT`, получателем которого является объект `sprite`, через цепочку диспетчеризации события.

Цепочка выглядит следующим образом:

```
Экземпляр класса Stage
  |
  |-> Объект CustomEventDemo
        |
        |-> Объект Sprite
```

В фазе захвата событие `CustomEventDemo.SOME_EVENT` проходит от экземпляра класса `Stage` до объекта `CustomEventDemo`. В фазе получения событие переходит к объекту `Sprite`. Наконец, в фазе всплытия событие снова переходит к объекту `CustomEventDemo` и затем вновь к экземпляру класса `Stage`. Когда экземпляр класса `Stage` в фазе всплытия получает уведомление о возникновении события, вызывается функция `someEventListener( )`. Несмотря на то что событие `CustomEventDemo.SOME_EVENT` является пользовательским, оно все равно передается по цепочке диспетчеризации события.

Как и в случае с внутренними событиями, иерархическая событийная архитектура языка ActionScript позволяет централизовать код, реагирующий на пользовательские события. Предположим, что мы создаем интернет-систему подачи заказов с элементом управления «корзина», который содержит значки выбранных продуктов. Элемент управления «корзина» представляется экземпляром пользовательского класса `ShoppingBasket`. Подобным образом значок каждого продукта представляется экземпляром пользовательского класса `Product`. Экземпляры этого класса являются отображаемыми детьми экземпляра класса `ShoppingBasket`. Экземпляр класса `ShoppingBasket` имеет строку заголовка, которая выводит название выбранного на настоящий момент продукта.

Когда пользователь выбирает значок продукта в элементе управления «корзина», приложение осуществляет диспетчеризацию пользовательского события `Product.PRODUCT_SELECTED`, получателем которого является соответствующий экземпляр класса `Product`. Поскольку экземпляр класса `ShoppingBasket` является предком всех экземпляров класса `Product`, он получает уведомления всякий раз, когда возникает событие `Product.PRODUCT_SELECTED`.

Таким образом, чтобы синхронизировать текст строки заголовка экземпляра `ShoppingBasket` с выбранным в настоящий момент продуктом, мы просто регистрируем единственный приемник `productSelectedListener( )` для события `Product.PRODUCT_SELECTED` в экземпляре класса `ShoppingBasket`. Когда вызывается метод `productSelectedListener( )`, мы знаем, что был выбран продукт, поэтому заменяем текст строки заголовка корзины названием нового выбранного продукта.

В листингах 21.6 и 21.7 показаны классы `ShoppingBasket` и `Product`. Пояснения к коду представлены в виде комментариев. Части кода, относящиеся к событиям, выделены полужирным шрифтом.

Листинг 21.6. Класс ShoppingBasket

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;

  // Интернет-корзина, которая может визуально содержать объекты Product.
  public class ShoppingBasket extends Sprite {
    // Отображаемая на экране строка заголовка панели «корзина»
    private var title:TextField;
    // Массив товаров в этой корзине
    private var products:Array;
    // Выбранный в настоящий момент продукт
    private var selectedProduct:Product;

    // Конструктор
    public function ShoppingBasket ( ) {
      // Создаем новый пустой массив для хранения объектов Product
      products = new Array( );
      // Создаем отображаемую на экране строку заголовка
      title = new TextField( );
      title.text = "No product selected";
      title.autoSize = TextFieldAutoSize.LEFT;
      title.border = true;
      title.background = true;
      title.selectable = false;
      addChild(title);

      // Регистрируем приемник для событий Product.PRODUCT_SELECTED,
      // получателями которых являются дочерние объекты Product.
      addEventListener(Product.PRODUCT_SELECTED, productSelectedListener);
    }

    // Добавляет новый объект Product в корзину
    public function addProduct (product:Product):void {
      // Создаем новый продукт и добавляем его в массив products
      products.push(product);
      // Добавляем новый продукт в иерархию отображения данного объекта
      addChild(products[products.length-1]);

      // Теперь, когда у нас появился новый продукт, изменяем
      // положение всех продуктов
      updateLayout( );
    }

    // Очень простой алгоритм размещения продуктов,
    // упорядочивающий все продукты в одну строку
```

```
    // в том порядке, в котором они были добавлены
    // в корзину, слева направо
    public function updateLayout ( ):void {
      var totalX:Number = 0;
      for (var i:int = 0; i < products.length; i++) {
        products[i].x = totalX;
        totalX += products[i].width + 20;   // 20 – интервал между колонками
        products[i].y = title.height + 20;  // 20 – интервал между строками
      }
    }

    // Реагирует на возникновение событий Product.PRODUCT_SELECTED,
    // получателями которых являются дочерние объекты Product.
    // Когда продукт выбран, этот метод обновляет
    // строку заголовка корзины, используя
    // название выбранного продукта.
    private function productSelectedListener (e:Event):void {
      // Запоминаем выбранный продукт на тот случай,
      // если нам в дальнейшем потребуется
      // обратиться к нему
      selectedProduct = Product(e.target);

      // Обновляем строку заголовка корзины
      title.text = selectedProduct.getName( );
    }
  }
}
```

Листинг 21.7. Класс Product

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  // Значок продукта, который может быть помещен в объект ShoppingBasket
  // с помощью метода ShoppingBasket.addProduct( ).
  // В этой упрощенной версии класса значок – просто текстовое поле
  // без соответствующего графического значка.
  public class Product extends Sprite {
    // Событийная константа для пользовательского события PRODUCT_SELECTED
    public static const PRODUCT_SELECTED:String = "PRODUCT_SELECTED";
    // Отображаемая на экране надпись, представляющая название продукта
    private var label:TextField;
    // Название продукта
    private var productName:String;

    // Конструктор
    public function Product (productName:String) {
      // Сохраняем название продукта
      this.productName = productName;
      // Создаем отображаемую на экране надпись
      label = new TextField( );
```

```
      label.text = productName;
      label.autoSize = TextFieldAutoSize.LEFT;
      label.border = true;
      label.background = true;
      label.selectable = false;
      addChild(label);
      // Регистрируем приемник для событий щелчка кнопкой мыши.
      // Регистрируя приемник для событий MouseEvent.CLICK в этом объекте,
      // мы будем получать уведомление о возникновении события всякий раз,
      // когда пользователь щелкает кнопкой мыши на его дочерних объектах
      // (например, на надписи).
      addEventListener(MouseEvent.CLICK, clickListener);
    }

    // Возвращает название продукта
    public function getName ( ):String {
      return productName;
    }

    // Обрабатывает события MouseEvent.CLICK. В этом примере, чтобы выбрать
    // продукт и осуществить диспетчеризацию события
    // Product.PRODUCT_SELECTED, достаточно просто щелкнуть кнопкой мыши
    // на названии продукта. В более сложной реализации могут учитываться
    // другие факторы. Например, можно добавить возможность выбора
    // продуктов с помощью клавиатуры, а на время трансакции с сервером
    // отключать возможность выбора продуктов.
    private function clickListener (e:MouseEvent):void {
      // Уведомляем всех зарегистрированных приемников о том, что был выбран
      // данный продукт. Благодаря системе иерархической диспетчеризации
      // событий языка ActionScript, в процессе диспетчеризации
      // пользовательского события, получателем которого является данный
      // объект, вызываются не только приемники события
      // Product.PRODUCT_SELECTED данного объекта, но и приемники события
      // Product.PRODUCT_SELECTED, зарегистрированные в экземпляре класса
      // ShoppingBasket.
      dispatchEvent(new Event(Product.PRODUCT_SELECTED, true));
    }
  }
}
```

В листинге 21.8 представлено очень простое приложение, демонстрирующее основы использования классов `ShoppingBasket` и `Product` из листингов 21.6 и 21.7.

Листинг 21.8. Демонстрация использования класса ShoppingDemo

```
package {
  import flash.display.Sprite;

  public class ShoppingBasketDemo extends Sprite {
    public function ShoppingBasketDemo ( ) {
      var basket:ShoppingBasket = new ShoppingBasket( );
      basket.addProduct(new Product("Nintendo Wii"));
      basket.addProduct(new Product("Xbox 360"));
```

```
            basket.addProduct(new Product("PlayStation 3"));
            addChild(basket);
        }
    }
}
```

Переходим к событиям ввода

Мы рассмотрели практически все вопросы, касающиеся основных процессов, происходящих в иерархической событийной системе языка ActionScript. В следующем разделе мы применим полученные теоретические знания на практике, знакомясь с внутренними событиями пользовательского ввода приложения Flash Player.

Интерактивность

В этой главе мы рассмотрим, как можно добавить интерактивность в приложение, реагируя на события ввода Flash Player. В частности, мы познакомимся с пятью различными категориями событий ввода:

- мыши;
- фокуса;
- клавиатуры;
- текстовыми событиями;
- уровня приложения Flash Player.

Для каждой из перечисленных категорий мы рассмотрим конкретные события, предлагаемые интерфейсом API приложения Flash Player, и код, необходимый для обработки этих событий.

Представленные в этой главе описания событий относятся к приложению Flash Player (к версии, реализованной в виде модуля расширения браузера, и к автономной версии), но в целом данная информация применима и к любой другой среде, поддерживающей ввод данных с помощью мыши и клавиатуры (например, приложение Adobe AIR). Если вы используете другие среды выполнения, обязательно просмотрите соответствующую документацию по событиям ввода. Официальную справочную информацию, относящуюся к событиям ввода приложения Flash Player, можно найти в разделе Constants описания класса `Event` и его подклассов в справочнике по языку ActionScript корпорации Adobe. Полезная информация также находится в разделе Events описания классов `TextField`, `DisplayObject`, `InteractiveObject` и `Stage`.

Перед тем как приступить к нашему знакомству с конкретными событиями ввода, бегло рассмотрим некоторые основные правила, относящиеся ко всем событиям ввода.

Только экземпляры класса InteractiveObject. Уведомления о возникновении событий ввода отправляются экземплярам только тех классов, которые наследуются от класса `InteractiveObject` (`Sprite`, `TextField`, `Stage`, `Loader`, `MovieClip`, `SimpleButton` и подклассов этих классов). Объекты других типов не получают уведомления о возникновении событий ввода, даже если эти объекты находятся в списке отображения. Например, экземпляры класса `Shape` могут помещаться в список отображения, однако класс `Shape` не наследуется от класса `InteractiveObject`, поэтому экземпляры класса `Shape` не получают уведомления о возникновении событий ввода. Если объект `Shape` визуально перекрывает объект `TextField` и пользователь щелкает кнопкой мыши на объекте `Shape`, получателем результирующего события щелчка будет являться не `Shape`,

а TextField. Чтобы организовать взаимодействие с объектом Shape или объектом Bitmap, поместите его в контейнер (Sprite или MovieClip) и зарегистрируйте приемник в этом контейнере для событий ввода.

Только объекты из списка отображения. Объекты, не находящиеся в списке отображения на момент диспетчеризации приложением Flash Player некоторого события ввода, не смогут получить уведомление о возникновении этого события.

Поведение по умолчанию. Некоторые события ввода вызывают стандартную реакцию приложения Flash Player, называемую *поведением по умолчанию*. Например, перемещение указателя мыши над экземпляром класса SimpleButton приводит к тому, что этот экземпляр отображает графическое изображение, связанное с состоянием «над» (когда указатель мыши находится над данным экземпляром). В некоторых случаях поведение по умолчанию приложения Flash Player может быть отменено путем вызова метода экземпляра preventDefault( ) класса Event. Дополнительную информацию можно найти в разд. «Отмена стандартного поведения событий» гл. 12.

Теперь переходим непосредственно к событиям!

События мыши

Приложение Flash Player выполняет диспетчеризацию событий мыши, когда пользователь производит манипуляции с системным указательным устройством. К указательным устройствам, которые могут генерировать события мыши, относятся: мышь, трекбол, сенсорный планшет ноутбука, джойстик ноутбука и перо. Тем не менее для удобства в этой книге используется всеобъемлющий термин «мышь», подразумевающий системное указательное устройство. События мыши могут возникать в результате следующих типов манипуляций:

- нажатие или отпускание левой кнопки мыши;
- перемещение указателя;
- использование колесика прокрутки мыши (например, вращение).

Обратите внимание, что «щелчок правой кнопкой мыши» (то есть нажатие вспомогательной кнопки мыши) не включен в приведенный список. Приложение Flash Player генерирует события мыши только для основной (левой) кнопки мыши. Тем не менее стандартное контекстное меню приложения Flash Player, которое открывается щелчком вспомогательной кнопкой мыши, является настраиваемым. Дополнительную информацию можно найти в описании класса ContextMenu справочника по языку ActionScript корпорации Adobe.

Внутренние события мыши приложения Flash Player

В табл. 22.1 перечислены типы внутренних событий приложения Flash Player. Для каждого типа события столбец «Тип события» содержит константу класса

MouseEvent, представляющую официальное строковое название типа события. В столбце «Описание» указывается конкретное действие пользователя, приводящее к возникновению этого события. Столбец «Получатель» определяет объект, который играет роль получателя события при его диспетчеризации. Столбец «Поведение по умолчанию» описывает стандартную реакцию приложения Flash Player на данное событие. Столбец «Всплывает» содержит информацию о том, имеет ли данное событие фазу всплытия. Столбец «Тип данных объекта, передаваемого в функцию-приемник» определяет тип данных объекта, передаваемого в функцию-приемник, обрабатывающую данное событие. Наконец, столбец «Примечание» содержит важную информацию, касающуюся использования данного события.

Потратьте немного времени, чтобы познакомиться с событиями мыши приложения Flash Player, внимательно изучив табл. 22.1. Однако сначала рассмотрим несколько основных моментов, которые необходимо иметь в виду при чтении информации, представленной в таблице.

Системный фокус — следующие события генерируются даже тогда, когда приложение Flash Player не имеет системного фокуса: MouseEvent.MOUSE_MOVE, MouseEvent.MOUSE_OVER, MouseEvent.MOUSE_OUT, MouseEvent.ROLL_OVER и MouseEvent.ROLL_OUT. Все остальные события мыши генерируются только в том случае, когда приложение имеет системный фокус.

Местоположение имеет значение — диспетчеризация событий не выполняется, когда пользователь манипулирует мышью за пределами области отображения приложения Flash Player, с одним исключением: если пользователь нажимает основную кнопку мыши в области отображения приложения Flash Player и затем отпускает ее за пределами этой области, выполняется диспетчеризация события MouseEvent.MOUSE_UP.

Для того чтобы получить уведомление, когда указатель мыши покидает область отображения приложения Flash Player, зарегистрируйте приемник для события Event.MOUSE_MOVE, как рассматривается далее в разд. «События ввода уровня приложения Flash Player».

Векторные изображения игнорируются в экземпляре основного класса — взаимодействие мыши с векторным содержимым, нарисованным через переменную экземпляра graphics основного класса SWF-файла, не вызывает генерацию событий мыши. Тем не менее генерация событий мыши происходит в результате взаимодействия мыши с векторным содержимым, нарисованным через переменную экземпляра graphics любого другого экземпляра класса InteractiveObject или его подклассов.

Поведение по умолчанию отменить нельзя — поведение по умолчанию приложения Flash Player нельзя отменить ни для какого типа событий мыши.

Мгновенное обновление экрана — чтобы обновить содержимое на экране после обработки любого события мыши, используйте метод экземпляра updateAfterEvent( ) класса MouseEvent. Полную информацию можно найти в разд. «Постсобытийные обновления экрана» гл. 23.

Таблица 22.1. События мыши приложения Flash Player

Тип события	Описание	Получатель	Поведение по умолчанию	Всплывает	Тип данных объекта, передаваемого в функцию-приемник	Примечание
MouseEvent.MOUSE_DOWN	Нажатие основной кнопки мыши, когда указатель мыши находится над областью отображения приложения Flash Player	Объект InteractiveObject, находящийся под указателем мыши в момент нажатия основной кнопки мыши	Если получателем является объект SimpleButton, отображается графика, на которую ссылается его переменная downState. Если получателем является объект TextField, для которого включена возможность выделения, начинается процесс выделения. Отменить нельзя	Да	MouseEvent	
MouseEvent.MOUSE_UP	Отпускание основной кнопки мыши, когда указатель мыши находится над областью отображения приложения Flash Player	Объект InteractiveObject, находящийся под указателем мыши в момент отпускания основной кнопки мыши	Если получателем является объект SimpleButton, отображается графика, на которую ссылается его переменная overState. Если получателем является объект TextField, для которого включена возможность выделения, заканчивается процесс выделения. Отменить нельзя	Да	MouseEvent	
MouseEvent.CLICK	Нажатие и последующее отпускание основной кнопки мыши над одним и тем же объектом InteractiveObject. Или активизация пользователем экземпляра класса SimpleButton, Sprite или MovieClip клавишей Пробел или Enter. Дополнительную информацию можно найти далее, в разд. «События фокуса»	Объект InteractiveObject, на котором щелкнули кнопкой мыши или активизировали	Отсутствует	Да	MouseEvent	

Продолжение

Таблица 22.1 (продолжение)

Тип события	Описание	Получатель	Поведение по умолчанию	Всплывает	Тип данных объекта, передаваемого в функцию-приемник	Примечание
MouseEvent.DOUBLE_CLICK	Два события MouseEvent.CLICK, происходящих одно за другим над одним и тем же объектом InteractiveObject	Объект InteractiveObject, на котором дважды щелкнули кнопкой мыши	Если получателем является объект TextField, для которого включена возможность выделения, выделяется слово, находящееся под указателем мыши	Да	MouseEvent	Возникает, только если программист заранее присвоил переменной doubleClickEnabled объекта-получателя значение true. В последовательности двойного щелчка первый генерирует событие MouseEvent.CLICK; второй — событие MouseEvent.DOUBLE_CLICK. Скорость двойного щелчка определяется операционной системой
MouseEvent.MOUSE_MOVE	Указатель мыши перемещается над областью отображения приложения Flash Player	Объект InteractiveObject, находящийся под указателем мыши в момент его перемещения	Если получателем является объект TextField с выделенным текстом, происходит обновление выделения. Отменить нельзя	Да	MouseEvent	
MouseEvent.MOUSE_OVER	Указатель мыши переместился на отображаемый объект	Объект InteractiveObject, на который переместился указатель мыши	Когда получателем является объект SimpleButton, при нажатой основной кнопке мыши отображается объект, на который ссылается переменная upState. Если основная кнопка мыши отпущена, отображается объект, на который ссылается переменная overState. Отменить нельзя	Да	MouseEvent	Не генерируется для экземпляра класса Stage. Чтобы обратиться к объекту InteractiveObject, над которым до настоящего момента находился указатель мыши, используйте переменную MouseEvent.relatedObject

MouseEvent. MOUSE_OUT	Указатель мыши переместился за границы отображаемого объекта	Объект InteractiveObject, за границы которого переместился указатель мыши	Если получателем является объект SimpleButton, отображается объект, на который ссылается переменная upState. Отменить нельзя	Да	MouseEvent	Не генерируется для экземпляра класса Stage. Чтобы обратиться к объекту InteractiveObject, над которым находится указатель мыши, используйте переменную MouseEvent. relatedObject
MouseEvent. ROLL_OVER	Указатель мыши переместился на некоторый объект InteractiveObject или на одного из его потомков	Объект InteractiveObject, зарегистрировавший приемник события	Отсутствует	Нет	MouseEvent	Не генерируется для экземпляра класса Stage. Чтобы обратиться к объекту InteractiveObject, над которым до настоящего момента находился указатель мыши, используйте переменную MouseEvent. relatedObject
MouseEvent. ROLL_OUT	Указатель мыши прежде переместился на некоторый объект InteractiveObject или одного из его потомков, но больше не находится над этим отображаемым объектом или одним из его потомков	Объект InteractiveObject, зарегистрировавший приемник события	Отсутствует	Нет	MouseEvent	Не генерируется для экземпляра класса Stage. Чтобы обратиться к объекту InteractiveObject, над которым сейчас находится указатель мыши, используйте переменную MouseEvent.relatedObject
MouseEvent. MOUSE_WHEEL	Когда приложение Flash Player имело системный фокус, было использовано устройство прокрутки мыши	Объект InteractiveObject, над которым находился указатель мыши в момент использования устройства прокрутки	Если получателем является объект TextField, происходит прокручивание его данных. Нельзя отменить с помощью метода экземпляра preventDefault( ) класса Event, но см. столбец «Примечание»	Да	MouseEvent	Чтобы предотвратить прокручивание текстовых полей, присвойте переменной mouseWheelEnabled получателя значение false

Регистрация приемников для событий мыши

Для регистрации приемника для события мыши используется следующая базовая процедура.

1. Применяя информацию о событии в столбце «Описание» табл. 22.1, найдите константу для желаемого типа события в столбце «Тип события».
2. Создайте функцию-приемник с единственным параметром, имеющим тип данных `MouseEvent`.
3. Просмотрите столбец «Получатель» табл. 22.1, чтобы определить объект-получателя события.
4. Наконец, зарегистрируйте функцию, определенную на шаге 2 либо в объекте-получателе события (для уведомлений в фазе получения), либо в одном из предков объекта-получателя (для уведомлений в фазе захвата или в фазе всплытия). Большинство событий мыши обрабатывается приемниками, зарегистрированными в получателе события (то есть в объекте, который концептуально получает данные).

Применим описанные шаги на примере. Предположим, мы хотим зарегистрировать функцию-приемник для получения уведомлений, когда пользователь щелкает кнопкой мыши на следующем объекте `TextField`:

```
var theTextField:TextField = new TextField( );
theTextField.text       = "Click here";
theTextField.autoSize   = TextFieldAutoSize.LEFT;
theTextField.border     = true;
theTextField.background = true;
theTextField.selectable = false;
```

Мы осуществляем следующие шаги.

1. Используя информацию в столбце «Описание» табл. 22.1, мы определяем, что константой события щелчка кнопкой мыши является `MouseEvent.CLICK`.
2. Далее мы создаем функцию `clickListener( )`, которая будет получать уведомления о возникновении событий `MouseEvent.CLICK`. Будьте внимательны: типом данных параметра функции `clickListener( )` является `MouseEvent`.

```
private function clickListener (e:MouseEvent):void {
  trace("Mouse was clicked");
}
```

3. Далее в соответствии с информацией в столбце «Получатель» табл. 22.1 мы определяем, что получателем события `MouseClick.CLICK` является объект `InteractiveObject`, на котором пользователь щелкнул кнопкой мыши. Мы хотим знать, когда щелкают на объекте `theTextField`, поэтому наш приемник события должен быть зарегистрирован либо в объекте `theTextField`, либо в одном из его отображаемых предков.
4. Для этого примера мы зарегистрируем приемник `clickListener( )` непосредственно в объекте-получателе `TextField` — `theTextField`, как показано в следующем коде:

```
theTextField.addEventListener(MouseEvent.CLICK, clickListener);
```

В результате выполнения описанных шагов наш метод `clickListener( )` будет вызываться всякий раз, когда пользователь щелкает кнопкой мыши на объекте `theTextField`. В листинге 22.1 представлен код, демонстрирующий предыдущие шаги в контексте простого класса `ClickableText`.

Листинг 22.1. Обработка события MouseEvent.CLICK

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;

  public class ClickableText extends Sprite {
    public function ClickableText ( ) {
      var theTextField:TextField = new TextField( );
      theTextField.text       = "Click here";
      theTextField.autoSize   = TextFieldAutoSize.LEFT;
      theTextField.border     = true;
      theTextField.background = true;
      theTextField.selectable = false;
      addChild(theTextField);

      theTextField.addEventListener(MouseEvent.CLICK, clickListener);
    }

    private function clickListener (e:MouseEvent):void {
      trace("Mouse was clicked");
    }
  }
}
```

Все прошло практически без проблем. Рассмотрим другой пример. Следующий код регистрирует приемник `mouseMoveListener( )`, выполняющийся всякий раз при перемещении мыши, когда указатель находится над объектом `Sprite`, на который ссылается переменная `triangle`.

```
// Создаем треугольник
var triangle:Sprite = new Sprite( );
triangle.graphics.lineStyle(1);
triangle.graphics.beginFill(0x00FF00, 1);
triangle.graphics.moveTo(25, 0);
triangle.graphics.lineTo(50, 25);
triangle.graphics.lineTo(0, 25);
triangle.graphics.lineTo(25, 0);
triangle.graphics.endFill( );
triangle.x = 200;
triangle.y = 100;

// Регистрируем приемник в объекте triangle для событий MouseEvent.MOUSE_MOVE
triangle.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);

// ...где-то в классе определяем приемник
private function mouseMoveListener (e:MouseEvent):void {
```

```
  trace("mouse move");
}
```

Как видно из предыдущего кода, приемники для событий мыши можно регистрировать в объекте, который не находится в списке отображения. Тем не менее такие приемники не будут вызываться до тех пор, пока объект не будет добавлен в список отображения.

Объект не сможет получать уведомления о возникновении событий ввода, пока он не будет добавлен в список отображения.

В листинге 22.2 предыдущий код, создающий треугольник, помещается в контекст основного класса `MouseMotionSensor` SWF-файла. Здесь объект `triangle` добавляется в список отображения, поэтому может получать уведомления о возникновении событий мыши.

Листинг 22.2. Обработка события MouseEvent.MOUSE_MOVE над треугольником

```
package {
  import flash.display.*;
  import flash.events.*;

  public class MouseMotionSensor extends Sprite {
    public function MouseMotionSensor ( ) {
      // Создаем треугольник
      var triangle:Sprite = new Sprite( );
      triangle.graphics.lineStyle(1);
      triangle.graphics.beginFill(0x00FF00, 1);
      triangle.graphics.moveTo(25, 0);
      triangle.graphics.lineTo(50, 25);
      triangle.graphics.lineTo(0, 25);
      triangle.graphics.lineTo(25, 0);
      triangle.graphics.endFill( );
      triangle.x = 200;
      triangle.y = 100;

      // Добавляем объект triangle в список отображения
      addChild(triangle);

      // Регистрируем приемник в объекте triangle для событий
      // MouseEvent.MOUSE_MOVE
      triangle.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
    }

    private function mouseMoveListener (e:MouseEvent):void {
      trace("mouse move");
    }
  }
}
```

Базовый код, описывающий приемник, и код, регистрирующий приемник, представленные в листинге 22.2, могут быть использованы для любого события из табл. 22.1.

Упражнение для читателя. Попробуйте добавить новый код в листинг 22.2, который регистрирует приемники для всех событий, перечисленных в табл. 22.1. В качестве шаблона используйте код, описывающий и регистрирующий приемник для события `MouseEvent.MOUSE_MOVE`. Следующий код, который требуется для регистрации приемника события `MouseEvent.MOUSE_DOWN` (первое событие, перечисленное в табл. 22.1), поможет вам начать.

```
// Добавьте этот код, регистрирующий приемник события, в конструктор класса
triangle.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownListener);


// Добавьте этот код, описывающий приемник, в тело класса
private function mouseDownListener (e:MouseEvent):void {
  trace("mouse down");
}
```

События мыши и перекрывающиеся отображаемые объекты

По умолчанию, когда событие мыши возникает над двумя или более перекрывающимися экземплярами класса `InteractiveObject`, приложение Flash Player отправляет событие только одному экземпляру, который визуально расположен поверх остальных объектов. Объекты, находящиеся позади самого верхнего объекта, о возникновении события не уведомляются.

Например, если объект `TextField` визуально перекрывает объект `Sprite` и пользователь щелкает кнопкой мыши на объекте `TextField`, приложение Flash Player выполняет диспетчеризацию события `MouseEvent.CLICK`, получателем которого является только объект `TextField`. Объект `Sprite` не получает уведомления о том, что произошел щелчок кнопкой мыши.

В листинге 22.3 представлено простое приложение, которое демонстрирует предыдущий сценарий «объект `TextField` над объектом `Sprite`». Основной класс приложения `ClickSensor` регистрирует приемник `clickListener( )` для события `MouseEvent.CLICK` в объекте `Sprite` — `circle`. Объект `Sprite` частично перекрывается объектом `TextField` — `textfield`. Когда вызывается метод `clickListener( )`, происходит перемещение объекта `circle` на 10 пикселов вправо.

Листинг 22.3. Приемник событий мыши, зарегистрированный в частично перекрытом объекте

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  public class ClickSensor extends Sprite {
    public function ClickSensor ( ) {
      // Создаем круг
      var circle:Sprite = new Sprite( );
      circle.graphics.beginFill(0x999999, 1);
```

```
        circle.graphics.lineStyle(1);
        circle.graphics.drawEllipse(0, 0, 100, 100);

        // Создаем объект TextField
        var textfield:TextField = new TextField( );
        textfield.text = "Click here";
        textfield.autoSize = TextFieldAutoSize.LEFT;
        textfield.x = 30;
        textfield.y = 30;
        textfield.border = true;
        textfield.background = true;

        // Добавляем объект circle в список отображения
        addChild(circle);
        // Добавляем объект textfield в список отображения,
        // поверх объекта circle
        addChild(textfield);

        // Регистрируем приемник для получения уведомлений,
        // когда пользователь щелкает кнопкой мыши на круге
        circle.addEventListener(MouseEvent.CLICK, clickListener);
    }

    // Обрабатывает события MouseEvent.CLICK, получателем которых является
    // объект circle.
    private function clickListener (e:MouseEvent):void {
        trace("User clicked: " + e.target);
        DisplayObject(e.target).x += 10;
    }
  }
}
```

На рис. 22.1 показан результат выполнения кода из листинга 22.3.

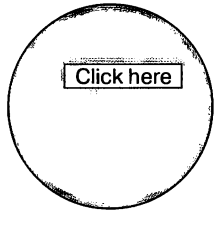

Рис. 22.1. Результат выполнения приложения ClickSensor

Если при выполнении приложения `ClickSensor` пользователь щелкнет кнопкой мыши на видимой части объекта `circle`, этот объект сместится вправо. Но если пользователь щелкнет кнопкой мыши на части объекта `circle`, которая перекрыта объектом `textfield`, объект `circle` не получит уведомления о возникновении события `MouseEvent.CLICK` и, следовательно, не будет смещен.

Однако существует возможность сделать так, чтобы объект `textfield` игнорировал любые события мыши, тем самым позволяя объекту `circle` определять щелчки

кнопкой мыши даже в тех местах, которые перекрыты объектом `textfield`. Чтобы объект `textfield` игнорировал все события мыши, мы присваиваем его переменной `mouseEnabled` значение `false`, как показано в следующем коде:

```
textfield.mouseEnabled = false;
```

Если бы эта строка кода была добавлена в метод-конструктор класса `ClickSensor`, все щелчки кнопкой мыши, происходящие над любой частью объекта `circle`, — видимой или невидимой, — приводили бы к перемещению объекта `circle` вправо.

Когда переменной `mouseEnabled` экземпляра класса `InteractiveObject` присвоено значение `false`, этот экземпляр не получает никаких уведомлений о возникновении событий мыши. Вместо этого события мыши отправляются следующему самому верхнему экземпляру класса `InteractiveObject` в списке отображения, для которого включена возможность обработки таких событий.

Определение позиции указателя мыши

Как было рассказано ранее, когда приложение Flash Player вызывает функцию-приемник события мыши, оно передает в эту функцию объект `MouseEvent`. Он хранит текущую позицию указателя мыши в следующих переменных экземпляра:

- `localX` и `localY`;
- `stageX` и `stageY`.

Переменные `localX` и `localY` представляют позицию указателя мыши в координатном пространстве получателя события (то есть относительно левого верхнего угла получателя события). В то же время переменные `stageX` и `stageY` представляют позицию указателя мыши в координатном пространстве экземпляра класса `Stage` (то есть относительно левого верхнего угла экземпляра класса `Stage`).

В листинге 22.4 демонстрируется использование переменных `localX`, `localY`, `stageX` и `stageY`. Здесь мы создаем объект `TextField`, добавляем его непосредственно в экземпляр класса `Stage` и затем размещаем его в позиции с координатами (100; 100). Когда пользователь щелкает кнопкой мыши на объекте `TextField`, мы выводим позицию указателя мыши относительно этого объекта (то есть получателя события) и относительно экземпляра класса `Stage`. Например, если пользователь щелкнет кнопкой мыши в точке, которая находится на 10 пикселов правее и 20 пикселов ниже левого верхнего угла объекта `TextField`, результат будет выглядеть следующим образом:

```
Position in TextField's coordinate space: (10, 20)
Position in Stage instance's coordinate space: (110, 120)
```

Вот этот код:

Листинг 22.4. Определение позиции указателя мыши

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;
```

```
  public class MousePositionChecker extends Sprite {
    public function MousePositionChecker ( ) {
      // Создаем объект TextField
      var textfield:TextField = new TextField( );
      textfield.text = "Click here";
      textfield.autoSize = TextFieldAutoSize.LEFT;
      textfield.x = 100;
      textfield.y = 100;

      // Добавляем объект textfield в список отображения в качестве
      // непосредственного ребенка экземпляра класса Stage
      stage.addChild(textfield);

      // Регистрируем приемник в объекте textfield
      // для событий щелчка кнопкой мыши
      textfield.addEventListener(MouseEvent.CLICK, clickListener);
    }

    // Когда пользователь щелкает кнопкой мыши на объекте textfield,
    // отображаем позицию указателя мыши
    private function clickListener (e:MouseEvent):void {
      // Позиция указателя мыши относительно объекта TextField
      trace("Position in TextField's coordinate space: ("
            + e.localX + ", " + e.localY + ")");
      // Позиция указателя мыши относительно экземпляра класса Stage
      trace("Position in Stage instance's coordinate space: ("
            + e.stageX + ", " + e.stageY + ")");
    }
  }
}
```

Обновляя позицию объекта в ответ на изменения позиции мыши, мы можем сделать так, чтобы этот объект следовал за мышью. В листинге 22.5 представлен код на языке ActionScript 3.0, реализующий пользовательский указатель мыши. В данном листинге применяются многие методики, с которыми мы познакомились на протяжении этой книги. В частности, код из листинга использует класс `StageDetector`, рассмотренный в подразд. «События ADDED_TO_STAGE и REMOVED_FROM_STAGE» разд. «События контейнеров» гл. 20. В листинге также применяются две методики, которые мы еще не рассматривали: преобразование координат и постсобытийные обновления экрана. Перекрестные ссылки на дополнительную информацию указаны в комментариях.

Листинг 22.5. Пользовательский указатель мыши

```
package {
  import flash.display.*;
  import flash.ui.*;
  import flash.events.*;
  import flash.geom.*;

  // Отображаемый класс, который заменяет указатель мыши новым изображением.
  // Когда объект CustomMousePointer добавляется в список отображения, он
```

```
// автоматически скрывает системный указатель и начинает следовать за его
// позицией. Когда объект CustomMousePointer удаляется из списка
// отображения, он автоматически восстанавливает системный указатель мыши.
public class CustomMousePointer extends Sprite {
  // Конструктор
  public function CustomMousePointer ( ) {
    // Создаем синий треугольник, который будем применять в качестве
    // пользовательского указателя мыши
    graphics.lineStyle(1);
    graphics.beginFill(0x0000FF, 1);
    graphics.lineTo(15, 5);
    graphics.lineTo(5, 15);
    graphics.lineTo(0, 0);
    graphics.endFill( );

    // Регистрируем приемник для получения уведомлений, когда этот объект
    // добавляется в список отображения или удаляется из него (требуется
    // пользовательский вспомогательный класс StageDetector)
    var stageDetector:StageDetector = new StageDetector(this);
    stageDetector.addEventListener(StageDetector.ADDED_TO_STAGE,
                                   addedToStageListener);
    stageDetector.addEventListener(StageDetector.REMOVED_FROM_STAGE,
                                   removedFromStageListener);
  }

  // Обрабатывает события StageDetector.ADDED_TO_STAGE
  private function addedToStageListener (e:Event):void {
    // Прячем системный указатель мыши
    Mouse.hide( );
              .
    // Регистрируем приемник для получения уведомлений, когда системный
    // указатель мыши перемещается над областью отображения приложения
    // Flash Player или выходит за ее пределы
    stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
    stage.addEventListener(Event.MOUSE_LEAVE, mouseLeaveListener);
  }

  // Обрабатываем события StageDetector.REMOVED_FROM_STAGE
  private function removedFromStageListener (e:Event):void {
    // Отображаем системный указатель мыши

    Mouse.show( );

    // Отменяем регистрацию приемников для событий мыши в экземпляре
    // класса Stage
    stage.removeEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
    stage.removeEventListener(Event.MOUSE_LEAVE, mouseLeaveListener);
  }

  // Обрабатываем события Event.MOUSE_LEAVE
  private function mouseLeaveListener (e:Event):void {
```

```
      // Когда мышь выходит за пределы области отображения приложения Flash
      // Player, прячем пользовательский указатель. В противном случае
      // пользовательский указатель мыши и системный указатель мыши будут
      // отображаться на экране одновременно.
      visible = false;
    }

    // Обрабатывает события MouseEvent.MOUSE_MOVE
    private function mouseMoveListener (e:MouseEvent):void {
      // При перемещении мыши обновляем позицию пользовательского указателя,
      // чтобы она соответствовала позиции системного указателя.
      // (Информацию по преобразованию точек между координатными
      // пространствами можно найти в описании метода
      // DisplayObject.globalToLocal( ) в справочнике по языку ActionScript
      // корпорации Adobe).
      var pointInParent:Point = parent.globalToLocal(new Point(e.stageX,
                                                               e.stageY));
      x = pointInParent.x;
      y = pointInParent.y;

      // Запрашиваем постсобытийное обновление экрана, чтобы анимация
      // указателя была максимально плавной. Дополнительную информацию
      // о методе MouseEvent.updateAfterEvent( ) можно найти в гл. 23.
      e.updateAfterEvent( );

      // Возникло событие MouseEvent.MOUSE_MOVE, поэтому системный указатель
      // должен находиться внутри области отображения приложения Flash
      // Player. Таким образом, убеждаемся, что пользовательский указатель
      // мыши отображается на экране (он может быть скрыт, поскольку
      // системный указатель мог покидать пределы области отображения
      // приложения Flash Player). Здесь этот код расположен неудачно –
      // он по праву принадлежит событию Event.MOUSE_ENTER, однако
      // в приложении Flash Player 9 такого события не существует.
      if (!visible) {
        visible = true;
      }
    }
  }
}
```

В листинге 22.6 представлен простой тестовый класс, который демонстрирует использование класса `CustomMousePointer` из листинга 22.5.

Листинг 22.6. Демонстрационный класс, представляющий использование класса CustomMousePointer

```
package {
  import flash.display.*;

  // Демонстрирует использование класса CustomMousePointer
  public class CustomMousePointerDemo extends Sprite {
    private var pointer:CustomMousePointer;
```

```
    // Конструктор
    public function CustomMousePointerDemo ( ) {
      // Создаем новый объект CustomMousePointer и добавляем его в список
      // отображения. В результате добавления объекта CustomMousePointer
      // в список отображения системный указатель мыши автоматически
      // заменяется объектом CustomMousePointer.
      pointer = new CustomMousePointer( );
      addChild(pointer);
    }
  }
}
```

Упражнение для читателя. Попытайтесь восстановить цепочку выполнения, которая начинается в конструкторе класса `CustomMousePointerDemo` из листинга 22.6 и завершается вызовом метода экземпляра `addedToStageListener( )` класса `CustomMousePointer`. В некоторый момент времени вам потребуется мысленно выполнить код класса `StageDetector` из гл. 20, что поможет получить более глубокие знания по API отображения и списку отображения. Ниже перечислены семь первых шагов в цепочке выполнения, которые помогут вам начать.

1. Создаем новый объект `CustomMousePointer`.
2. Выполняем метод-конструктор объекта `CustomMousePointer`, созданного на шаге 1.
3. Рисуем синий треугольник в объекте `CustomMousePointer`.
4. Создаем новый объект `StageDetector`, передавая в его конструктор объект `CustomMousePointer`.
5. Вызываем метод `setWatchedObject( )` объекта `StageDetector`, передавая объект `CustomMousePointer` в качестве его единственного аргумента.
6. Присваиваем объект `CustomMousePointer` переменной `watchedObject` объекта `StageDetector`.
7. Значение переменной `watchedObject.stage` равняется `null` (поскольку объект `CustomMousePointer` на настоящий момент не находится в списке отображения), поэтому переменной `onStage` объекта `StageDetector` присваиваем значение `false`.

Вы можете продолжить цепочку выполнения с этого момента. Желаем приятного времяпрепровождения!

Мысленное выполнение кода — это хороший способ понять, как работает программа.

«Глобальная» обработка событий мыши

В приложении Flash Player отсутствуют настоящие глобальные события мыши. Однако, зарегистрировав приемник событий мыши в экземпляре класса `Stage`, мы можем обрабатывать взаимодействия с мышью независимо от того, где они произошли в пределах области отображения приложения. Например, следующий

код регистрирует метод `mouseMoveListener( )` в экземпляре класса `Stage` для событий `MouseEvent.MOUSE_MOVE`:

```
package {
  import flash.display.*;
  import flash.events.*;

  public class GlobalMouseMotionSensor extends Sprite {
    public function GlobalMouseMotionSensor ( ) {
      stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
    }

    private function mouseMoveListener (e:MouseEvent):void {
      trace("The mouse moved.");
    }
  }
}
```

Всякий раз, когда при выполнении предыдущего кода указатель мыши перемещается над областью отображения Flash Player, приложение осуществляет диспетчеризацию события `MouseEvent.MOUSE_MOVE`, в результате чего вызывается метод `mouseMoveListener( )`.

Рассмотрим еще один пример «глобальной» обработки событий мыши. Чтобы выявить все моменты нажатий кнопки мыши в приложении, мы используем следующий код:

```
package {
  import flash.display.*;
  import flash.events.*;

  public class GlobalMouseDownSensor extends Sprite {
    public function GlobalMouseDownSensor ( ) {
      stage.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownListener);
    }

    private function mouseDownListener (e:MouseEvent):void {
      trace("The primary mouse button was pressed.");
    }
  }
}
```

Всякий раз, когда при выполнении предыдущего кода нажимается основная кнопка мыши и при этом указатель находится над областью отображения Flash Player, приложение осуществляет диспетчеризацию события `MouseEvent.MOUSE_DOWN`, в результате чего вызывается метод `mouseDownListener( )`.

Когда кнопка мыши нажимается над «пустым» участком области отображения приложения Flash Player (то есть над участком, где отсутствуют созданные программой отображаемые объекты), получателем события является экземпляр класса `Stage`. Когда нажатие кнопки мыши происходит над любым другим отображаемым объектом, получателем события является этот объект. Таким образом, проверяя получателя события в методе `mouseDownListener( )`, мы можем создать код, ко-

торый реагирует именно на нажатие кнопки мыши над пустыми участками области отображения приложения Flash Player. Описанную методику демонстрирует следующее приложение `CircleClicker`. Когда пользователь щелкает кнопкой мыши на пустом участке области отображения Flash Player, приложение `CircleClicker` рисует круг случайного размера и закрашивает его случайным цветом. Когда же пользователь щелкает кнопкой мыши на круге, приложение `CircleClicker` удаляет этот круг с экрана.

```
package {
  import flash.display.*;
  import flash.events.*;

  public class CircleClicker extends Sprite {
    public function CircleClicker ( ) {
      stage.addEventListener(MouseEvent.CLICK, clickListener);
    }

    private function clickListener (e:MouseEvent):void {
      // Если получателем события является экземпляр класса Stage,
      if (e.target == stage) {
        // ...рисуем круг
        drawCircle(e.stageX, e.stageY);
      } else {
        // ...в противном случае получателем события должен являться
        // объект Sprite, содержащий круг, поэтому удаляем данный объект
        removeChild(DisplayObject(e.target));
      }
    }

    public function drawCircle (x:int, y:int):void {
      var randomColor:int = Math.floor(Math.random( )*0xFFFFFF);
      var randomSize:int = 10 + Math.floor(Math.random( )*150);
      var circle:Sprite = new Sprite( )
      circle.graphics.beginFill(randomColor, 1);
      circle.graphics.lineStyle( );
      circle.graphics.drawEllipse(0, 0, randomSize, randomSize);
      circle.x = x-randomSize/2;
      circle.y = y-randomSize/2;
      addChild(circle);
    }
  }
}
```

Стоит отметить, что по соображениям безопасности загруженному SWF-файлу может быть запрещено обращаться к экземпляру класса `Stage` приложения Flash Player. В подобных ситуациях для «глобальной» обработки событий мыши применяйте методики, описанные в разд. «Обработка событий между границами зон безопасности» гл. 12.

Теперь перейдем к рассмотрению другого типа событий ввода — событий, возникающих в результате смены фокуса.

События фокуса

Когда объект получает *фокус ввода с клавиатуры*, он выступает в роли логического приемника всех введенных с клавиатуры данных и становится получателем всех событий клавиатурного ввода. Объект может получить фокус ввода с клавиатуры либо программным путем (через переменную экземпляра `focus` класса `Stage`), либо в результате взаимодействия с пользователем посредством мыши, клавиши Tab или клавиш управления курсором. Однако для получения фокуса ввода с клавиатуры объект должен быть экземпляром класса, который наследуется от класса `InteractiveObject`. Более того, в приложении Flash Player фокус ввода с клавиатуры единовременно может иметь только один объект.

Для краткости вместо термина «фокус ввода с клавиатуры» обычно используется сокращенный термин «фокус».

Установка фокуса на объекты программным путем

Чтобы установить фокус на объект программным путем, мы присваиваем этот объект переменной `focus` экземпляра класса `Stage`.

Например, следующий код создает объект `Sprite` и затем немедленно устанавливает на него фокус (предполагается, что объект `someDisplayContainer` находится в списке отображения):

```
var rect:Sprite = new Sprite( );
rect.graphics.lineStyle(1);
rect.graphics.beginFill(0x0000FF);
rect.graphics.drawRect(0, 0, 150, 75);
someDisplayContainer.addChild(rect);
someDisplayContainer.stage.focus = rect;
```

При выполнении предыдущего кода объект `rect` получает фокус и, следовательно, становится получателем всех возникающих событий клавиатурного ввода.

Установка фокуса на объекты с помощью клавиатуры

Чтобы установить фокус на объект с помощью клавиатуры, пользователь нажимает клавишу Tab или клавиши управления курсором. Тем не менее, чтобы объект мог получать фокус с помощью указанных клавиш, он должен быть частью порядка перехода приложения Flash Player. *Порядок перехода* — это набор всех объектов, находящихся в списке отображения, которые теоретически могут получать фокус ввода с помощью клавиатуры. Порядок перехода также определяет последовательность, в которой объекты получают фокус при нажатии пользователем клавиши Tab.

В приложении Flash Player существует два различных порядка перехода: автоматический и пользовательский. *Автоматический порядок перехода* — это порядок перехода, который используется приложением Flash Player по умолчанию, когда не

определен пользовательский порядок перехода. Автоматический порядок перехода включает следующие объекты:

- экземпляры класса `TextField`, находящиеся в списке отображения и переменной `type` которых присвоено значение `TextFieldType.INPUT`;
- экземпляры классов `Sprite` или `MovieClip`, находящиеся в списке отображения и у которых переменной `buttonMode` или `tabEnabled` присвоено значение `true`;
- экземпляры класса `SimpleBitton`, которые находятся в списке отображения.

Когда используется автоматический порядок перехода и пользователь нажимает клавишу **Tab**, фокус перемещается от одного объекта к другому в автоматическом порядке перехода. Последовательность автоматического порядка перехода приложения Flash Player определяется местоположением содержащихся в нем объектов — визуально перемещение фокуса происходит слева направо и затем сверху вниз (независимо от позиции этих объектов в иерархии отображения).

В отличие от автоматического, *пользовательский порядок* — это порядок перехода, определяемый программным путем и включающий произвольную последовательность объектов, переменной `tabIndex` которых присвоено неотрицательное целое число. В пользовательский порядок перехода могут быть включены следующие типы объектов:

- экземпляры класса `TextField`, переменной `type` которых присвоено значение `TextFieldType.INPUT`;
- экземпляры классов `Sprite`, `MovieClip` или `SimpleButton`.

Когда по крайней мере одному находящемуся в настоящий момент в списке отображения объекта переменной `tabIndex` присвоено значение 0 или большее, применяется пользовательский порядок перехода. В этом случае нажатие клавиши **Tab** приводит к установке фокуса на объекты в соответствии со значением их переменной `tabIndex`, начиная с минимального значения и заканчивая максимальным.

Например, если у объекта, имеющего фокус в настоящий момент, переменной `tabIndex` присвоено значение `2` и пользователь нажимает клавишу **Tab**, фокус устанавливается на объект, переменной `tabIndex` которого присвоено значение `3`. Если снова нажать клавишу **Tab**, фокус будет установлен на объект, переменной `tabIndex` которого присвоено значение `4`, и т. д. Если переменной `tabIndex` двух объектов присвоено одно и то же значение, то в порядке перехода первым окажется объект с меньшей глубиной. Объекты, переменной `tabIndex` которых значение не присвоено явно, исключаются из порядка перехода.

Независимо от того, какой порядок перехода используется в данный момент — автоматический или пользовательский, — когда фокус устанавливается на экземпляр класса `Sprite`, `MovieClip` или `SimpleButton` клавишей **Tab**, пользователь в дальнейшем может работать четырьмя клавишами для управления курсором, чтобы установить фокус на объект, расположенный в указанном направлении (сверху, снизу, слева или справа).

Чтобы исключить объект из автоматического или пользовательского порядка перехода, мы присваиваем переменной tabEnabled этого объекта значение false. Объекты, переменной _visible которых присвоено значение false, автоматически исключаются из порядка перехода. Чтобы исключить всех потомков некоторого контейнера отображаемых объектов из автоматического или пользовательского порядка перехода, мы присваиваем переменной tabChildren этого контейнера значение false.

По умолчанию приложение Flash Player отображает желтый прямоугольник вокруг экземпляров классов Sprite, MovieClip или SimpleButton, если фокус был установлен с помощью клавиатуры. Чтобы отключить отображение желтого прямоугольника для конкретного объекта, мы присваиваем его переменной focusRect значение false. Если нужно отключить отображение желтого прямоугольника для всех объектов, мы присваиваем переменной stageFocusRect экземпляра класса Stage значение false. Стоит отметить, что значение переменной stageFocusRect не отражается на значении переменной focusRect отдельных объектов. Тем не менее присваивание значения переменной focusRect конкретного объекта перекрывает значение переменной stageFocusRect.

Установка фокуса на объекты с помощью мыши

Подобно тому как пользователь может устанавливать фокус ввода с клавиатуры, используя клавишу Tab или клавиши управления курсором, он может устанавливать фокус, щелкая на объекте кнопкой мыши. Однако по умолчанию с помощью мыши устанавливать фокус можно только на экземпляры классов SimpleButton и TextField. Чтобы предоставить пользователю возможность устанавливать фокус на экземпляры классов Sprite или MovieClip с помощью мыши, используется один из следующих подходов:

- присвоить переменной buttonMode этого экземпляра значение true (убедившись, что его переменной tabEnabled явным образом не присвоено значение false);
- переменной tabEnabled этого экземпляра установить значение true;
- присвоить неотрицательное целое число переменной tabIndex этого экземпляра (убедившись, что его переменной tabEnabled явным образом не присвоено значение false).

Например, следующий код создает объект Sprite, который получает фокус ввода с клавиатуры, когда на нем щелкают кнопкой мыши:

```
var rect:Sprite = new Sprite( );
rect.graphics.lineStyle(1);
rect.graphics.beginFill(0x0000FF);
rect.graphics.drawRect(0, 0, 150, 75);
rect.tabEnabled = true;
```

Чтобы исключить установку фокуса на некоторый объект с помощью мыши, мы присваиваем переменной mouseEnabled этого объекта значение false. Чтобы исключить установку фокуса на потомков контейнера отображаемых

объектов, мы присваиваем переменной mouseChildren этого контейнера значение false.

На объект нельзя установить фокус с помощью мыши, если переменной mouseEnabled этого объекта присвоено значение false или если этот объект является отображаемым потомком контейнера, переменной mouseChildren которого присвоено значение false.

Более того, когда переменной tabEnabled экземпляра класса Sprite, MovieClip или SimpleButton явным образом присваивается значение false, на этот экземпляр невозможно установить фокус с помощью мыши. Подобным образом, когда экземпляр класса Sprite, MovieClip или SimpleButton является потомком контейнера, переменной tabChildren которого присвоено значение false, на этот экземпляр невозможно установить фокус с помощью мыши. Тем не менее из-за ошибки в приложении Flash Player, даже если переменной tabEnabled объекта TextField будет явным образом присвоено значение false, на этот объект можно установить фокус с помощью мыши. Подобным образом из-за той же ошибки, даже если переменной tabChildren контейнера, потомком которого является объект TextField, будет присвоено значение false, на этот объект все равно можно установить фокус с помощью мыши.

Установка фокуса на потомков через одного предка. Чтобы среда Flash считала контейнер отображаемых объектов и всех его потомков одной группой, которая может получать фокус одним щелчком кнопкой мыши, выполните следующие шаги.

1. Включите возможность установки фокуса с помощью мыши для данного контейнера, присвоив его переменной buttonMode или tabEnabled значение true либо переменной tabIndex любое неотрицательное целое число.
2. Отключите для дочерних объектов этого контейнера возможность взаимодействия с мышью, присвоив его переменной mouseChildren значение false.

События фокуса приложения Flash Player

Когда фокус ввода с клавиатуры приложения переходит на новый объект (с использованием одного из способов, описанных в трех предыдущих разделах), приложение Flash Player выполняет диспетчеризацию одного или нескольких событий фокуса, определяющих данный переход. В табл. 22.2 перечислены типы внутренних событий фокуса приложения Flash Player. Для каждого типа события столбец «Тип события» содержит константу класса FocusEvent, представляющую официальное строковое название типа события. В столбце «Описание» указывается конкретное действие пользователя, приводящее к возникновению этого события. Столбец «Получатель» описывает объект, который выступает в роли получателя события при его диспетчеризации. Столбец «Поведение по умолчанию» описывает стандартную реакцию приложения Flash Player на данное событие. Столбец «Всплывает» содержит информацию о том, имеет ли данное событие фазу всплытия. Наконец, столбец «Тип данных объекта, передаваемого в функцию-приемник» определяет тип данных объекта, передаваемого в функцию-приемник, обрабатывающую данное событие.

Таблица 22.2. События фокуса приложения Flash Player

Тип события	Описание	Получатель	Поведение по умолчанию	Всплывает	Тип данных объекта, передаваемого в функцию-приемник
FocusEvent.FOCUS_IN	Фокус был установлен на данный объект	Объект, получивший фокус. Чтобы обратиться к объекту, потерявшему фокус (если такой объект существует), используйте переменную экземпляра relatedObject класса FocusEvent	Отсутствует	Да	FocusEvent
FocusEvent. FOCUS_OUT	Фокус был потерян данным объектом	Объект, потерявший фокус. Чтобы обратиться к объекту, получившему фокус (если такой объект существует), используйте переменную экземпляра relatedObject класса FocusEvent	Отсутствует	Да	FocusEvent
FocusEvent. KEY_FOCUS_CHANGE	Пользователь попытался изменить фокус с помощью клавиатуры	Объект с установленным на настоящий момент фокусом. Чтобы обратиться к объекту, на который пытается установить фокус пользователь (если такой объект существует), используйте переменную экземпляра relatedObject класса FocusEvent	Приложение Flash Player изменяет фокус. Может быть отменено методом экземпляра preventDefault( ) класса Event	Да	FocusEvent
FocusEvent. MOUSE_FOCUS_CHANGE	Пользователь попытался изменить фокус с помощью мыши	Объект с установленным на настоящий момент фокусом. Чтобы обратиться к объекту, на который пытается установить фокус пользователь (если такой объект существует), используйте переменную экземпляра relatedObject класса FocusEvent	Приложение Flash Player изменяет фокус. Может быть отменено методом экземпляра preventDefault( ) класса Event	Да	FocusEvent

Как видно из табл. 22.2, события `FocusEvent.FOCUS_IN` и `FocusEvent.FOCUS_OUT` используются для определения момента, когда объект получает или теряет фокус. На тот момент, когда происходит диспетчеризация этих событий, изменение фокуса уже произошло. В отличие от этого, события `FocusEvent.KEY_FOCUS_CHANGE` и `FocusEvent.MOUSE_FOCUS_CHANGE` применяются для определения момента, когда объект собирается получить или потерять фокус, но еще не получил или не потерял его. Приложение обычно использует события `FocusEvent.KEY_FOCUS_CHANGE` и `FocusEvent.MOUSE_FOCUS_CHANGE`, чтобы исключить изменение фокуса пользователем с помощью клавиатуры или мыши, — возможно, чтобы заставить пользователя взаимодействовать с определенной частью интерфейса, например с модальным диалогом.

Как и в случае с событиями мыши, в приложении Flash Player отсутствуют настоящие глобальные события фокуса. Тем не менее, зарегистрировав приемник для событий фокуса в экземпляре класса `Stage`, мы можем обрабатывать все изменения фокуса, которые происходят в приложении Flash Player. Эта методика продемонстрирована в листинге 22.7 — приложение создает два объекта `TextField` и, когда на них устанавливается фокус, меняет цвет их фона на зеленый.

Листинг 22.7. Глобальная обработка событий фокуса

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  public class GlobalFocusSensor extends Sprite {
    public function GlobalFocusSensor ( ) {
      // Создаем текстовые поля
      var field1:TextField = new TextField( );
      field1.width      = 100;
      field1.height     = 30;
      field1.border     = true;
      field1.background = true;
      field1.type = TextFieldType.INPUT;

      var field2:TextField = new TextField( );
      field2.width      = 100;
      field2.height     = 30;
      field2.y          = 50;
      field2.border     = true;
      field2.background = true;
      field2.type = TextFieldType.INPUT;

      // Добавляем текстовые поля в список отображения
      addChild(field1);
      addChild(field2);

      // Регистрируем приемник для событий FocusEvent.FOCUS_IN
      stage.addEventListener(FocusEvent.FOCUS_IN, focusInListener);
    }
```

```
    // Обрабатываем все события FocusEvent.FOCUS_IN в данном приложении
    private function focusInListener (e:FocusEvent):void {
      // Изменяем цвет фона объекта TextField, на который был установлен
      // фокус, на зеленый
      TextField(e.target).backgroundColor = 0xFF00FF00;

      // Изменяем цвет фона объекта TextField, потерявшего фокус, на белый
      if (e.relatedObject is TextField) {
        TextField(e.relatedObject).backgroundColor = 0xFFFFFFFF;
      }
    }
  }
}
```

События фокуса могут также обрабатываться приемниками, зарегистрированными в объекте-получателе события или в любом из отображаемых предков объекта-получателя события. В листинге 22.8 показано приложение, которое создает объект `TextField`, не позволяющий сменить фокус (после того как он был установлен) до тех пор, пока не будет введено по крайней мере три символа. Подобный код часто применяется в приложениях, которые проверяют данные, введенные через заполняемые формы, например через экран регистрации или форму заказа товара.

Листинг 22.8. Обработка событий фокуса для конкретного объекта

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  // Демонстрирует, как обрабатывать события FocusEvent.FOCUS_IN для
  // отдельного объекта. Создает объект TextField, который после получения
  // фокуса не позволяет его сменить до тех пор, пока не будет введено
  // по крайней мере три символа.
  public class ObjectFocusSensor extends Sprite {
    private var namefield:TextField;
    private var passfield:TextField;

    public function ObjectFocusSensor ( ) {
      // Создаем текстовые поля
      namefield = new TextField( );
      namefield.width      = 100;
      namefield.height     = 30;
      namefield.border     = true;
      namefield.background = true;
      namefield.type = TextFieldType.INPUT;

      passfield = new TextField( );
      passfield.width      = 100;
      passfield.height     = 30;
      passfield.y          = 50;
      passfield.border     = true;
```

```
        passfield.background = true;
        passfield.type = TextFieldType.INPUT;

        // Добавляем текстовые поля в список отображения
        addChild(namefield);
        addChild(passfield);

        // Регистрируем приемник для событий изменения фокуса
        namefield.addEventListener(FocusEvent.MOUSE_FOCUS_CHANGE,
                                   focusChangeListener);
        namefield.addEventListener(FocusEvent.KEY_FOCUS_CHANGE,
                                   focusChangeListener);
    }

    // Обрабатывает все события изменения фокуса, получателем которых
    // является объект namefield
    private function focusChangeListener (e:FocusEvent):void {
      if (e.target == namefield && namefield.text.length < 3) {
        trace("Name entered is less than three characters long");
        e.preventDefault( );
      }
    }
  }
}
```

События ввода с клавиатуры

Приложение Flash Player выполняет диспетчеризацию событий ввода с клавиатуры, когда пользователь нажимает или отпускает клавишу. Вообще говоря, события ввода с клавиатуры обычно используются для инициирования ответного действия либо от всего приложения, либо от определенного элемента интерфейса. Например, нажатие клавиши S может вызывать глобальную команду сохранения пользовательских данных, а нажатие стрелки ↓ может выбирать элемент в конкретном компоненте меню.

События ввода с клавиатуры, которые вызывают команды для всего приложения, обычно обрабатываются глобально с помощью приемников, зарегистрированных в экземпляре класса `Stage` приложения Flash Player. В отличие от этого, события ввода с клавиатуры, инициирующие ответное действие для конкретного элемента интерфейса, обычно обрабатываются приемниками, зарегистрированными в объекте, который на настоящий момент имеет фокус ввода с клавиатуры.

События ввода с клавиатуры приложения Flash Player предназначены для использования при разработке приложений, управляемых с помощью клавиатуры, но не подходят для реакции на ввод текста в объекты TextField. Для реакции на ввод текста применяйте событие TextEvent.TEXT_INPUT, которое рассматривается далее, в разд. «События текстового ввода».

В табл. 22.3 перечислены типы внутренних событий клавиатуры приложения Flash Player. Для каждого типа события столбец «Тип события» содержит константу класса `KeyboardEvent`, представляющую официальное строковое название типа события. Столбец «Описание» определяет конкретное действие пользователя, приводящее к возникновению этого события. Столбец «Получатель» описывает объект, который выступает в роли получателя события при его диспетчеризации. В столбце «Поведение по умолчанию» приводится стандартная реакция приложения Flash Player на данное событие. В отличие от событий мыши и событий фокуса, у событий клавиатуры отсутствует поведение по умолчанию. Столбец «Всплывает» содержит информацию о том, имеет ли данное событие фазу всплытия. Наконец, столбец «Тип данных объекта, передаваемого в функцию-приемник» определяет тип данных объекта, передаваемого в функцию-приемник, обрабатывающую данное событие.

Стоит отметить, что диспетчеризация событий клавиатуры осуществляется только тогда, когда Flash Player имеет системный фокус. Чтобы получать уведомления о получении или потере системного фокуса приложением Flash Player, зарегистрируйте приемник для событий `Event.ACTIVATE` и `Event.DEACTIVATE` (которые рассматриваются далее, в разд. «События ввода уровня приложения Flash Player»).

Таблица 22.3. События клавиатуры приложения Flash Player

Тип события	Описание	Получатель	Поведение по умолчанию	Всплывает	Тип данных объекта, передаваемого в функцию-приемник
KeyboardEvent.KEY_DOWN	Клавиша нажата	Объект InteractiveObject, имеющий фокус ввода с клавиатуры, либо, если никакой объект не имеет фокуса, экземпляр класса Stage	Отсутствует	Да	KeyboardEvent
KeyboardEvent.KEY_UP	Клавиша отпущена	Объект InteractiveObject, имеющий фокус ввода с клавиатуры, либо, если никакой объект не имеет фокуса, экземпляр класса Stage	Отсутствует	Да	KeyboardEvent

Глобальная обработка событий клавиатуры

Как и в случае с событиями мыши и фокуса, в приложении Flash Player отсутствуют настоящие глобальные события клавиатуры. Тем не менее, зарегистрировав приемник для событий клавиатуры в экземпляре класса `Stage`, можно обрабатывать все взаимодействия с клавиатурой, которые происходят в приложении Flash Player, когда оно имеет системный фокус. Эта методика продемонстрирована в листинге 22.9 — упрощенный класс отображает отладочное сообщение при нажатии любой клавиши.

Листинг 22.9. Глобальная обработка событий клавиатуры

```
package {
  import flash.display.*;
  import flash.events.*;

  public class GlobalKeyboardSensor extends Sprite {
    public function GlobalKeyboardSensor ( ) {
      // Регистрируем приемник для получения уведомлений о нажатии клавиши
      stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
    }

    // Эта функция вызывается всякий раз при нажатии клавиши, когда
    // приложение Flash Player имеет системный фокус
    private function keyDownListener (e:KeyboardEvent):void {
      trace("A key was pressed.");
    }
  }
}
```

Обработка событий клавиатуры для конкретного объекта

Как видно из табл. 22.3, когда никакой объект не имеет фокуса ввода с клавиатуры, получателем всех событий клавиатуры является экземпляр класса `Stage`. В отличие от этого, когда экземпляр класса `InteractiveObject` имеет фокус ввода с клавиатуры, данный экземпляр является получателем всех событий клавиатуры, диспетчеризация которых выполняется приложением Flash Player.

Следовательно, чтобы реагировать на клавиатурный ввод, направленный в конкретный объект, мы регистрируем приемники в этом объекте. Это демонстрирует листинг 22.10, в котором показано приложение, создающее два объекта `Sprite` — `rect1` и `rect2`. Когда фокус установлен на объекте `rect1` и пользователь нажимает любую клавишу, приложение перемещает объект `rect1` вправо. Когда фокус установлен на объекте `rect2` и пользователь нажимает любую клавишу, приложение вращает объект `rect2`.

Листинг 22.10. Обработка событий клавиатуры для конкретного объекта

```
package {
  import flash.display.*;
  import flash.events.*;

  public class ObjectKeyboardSensor extends Sprite {
    public function ObjectKeyboardSensor ( ) {
      // Создаем прямоугольники
      var rect1:Sprite = new Sprite( );
      rect1.graphics.lineStyle(1);
      rect1.graphics.beginFill(0x0000FF);
      rect1.graphics.drawRect(0, 0, 75, 75);
      rect1.tabEnabled = true;

      var rect2:Sprite = new Sprite( );
      rect2.graphics.lineStyle(1);
```

```
      rect2.graphics.beginFill(0x0000FF);
      rect2.graphics.drawRect(0, 0, 75, 75);
      rect2.x = 200;
      rect2.tabEnabled = true;

      // Добавляем прямоугольники в список отображения
      addChild(rect1);
      addChild(rect2);

      // Регистрируем приемники в прямоугольниках для событий клавиатуры
      rect1.addEventListener(KeyboardEvent.KEY_DOWN, rect1KeyDownListener);
      rect2.addEventListener(KeyboardEvent.KEY_DOWN, rect2KeyDownListener);
    }

    // Выполняется, когда фокус установлен на объекте rect1 и пользователь
    // нажимает любую клавишу
    private function rect1KeyDownListener (e:KeyboardEvent):void {
      Sprite(e.target).x += 10;
    }

    // Выполняется, когда фокус установлен на объекте rect2 и пользователь
    // нажимает любую клавишу
    private function rect2KeyDownListener (e:KeyboardEvent):void {
      Sprite(e.target).rotation += 10;
    }
  }
}
```

Теперь, когда мы знаем, как определить момент нажатия или отпускания клавиши пользователем, выясним, как можно определить, *какая* клавиша была нажата или отпущена.

Определение последней нажатой или отпущенной клавиши

Приложение Flash Player присваивает обычный числовой идентификатор, называемый *кодом клавиши*, всем выбираемым клавишам. Чтобы определить код последней нажатой или отпущенной клавиши, мы получаем значение переменной экземпляра `keyCode` класса `KeyboardEvent` в функции-приемнике события `KeyboardEvent.KEY_UP` или `KeyboardEvent.KEY_DOWN`, как показано в листинге 22.11.

Листинг 22.11. Получение кода нажатой клавиши

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  // Отображаем код для любой нажатой клавиши
  public class KeyViewer extends Sprite {
```

```
    private var keyoutput:TextField;
    public function KeyViewer ( ) {
      keyoutput = new TextField( );
      keyoutput.text = "Press any key...";
      keyoutput.autoSize = TextFieldAutoSize.LEFT;
      keyoutput.border     = true;
      keyoutput.background = true;
      addChild(keyoutput);

      stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
    }


    private function keyDownListener (e:KeyboardEvent):void {
      // Отображаем код для нажатой клавиши
      keyoutput.text = "The key code for the key you pressed is: "
                       + e.key code;
    }
  }
}
```

Чтобы определить момент, когда будет нажата или отпущена определенная клавиша, мы сравниванием ее код со значением переменной экземпляра `keyCode` класса `KeyboardEvent` в функции-приемнике событий `KeyboardEvent.KEY_UP` или `KeyboardEvent.KEY_DOWN`. Например, следующая функция-приемник события `KeyboardEvent.KEY_DOWN` определяет момент, когда будет нажата клавиша **Esc**, путем сравнения ее кода (27) со значением переменной `keyCode`.

```
private function keyDownListener (e:KeyboardEvent):void {
  if (e.keyCode == 27) {
    trace("The Escape key was pressed");
  }
}
```

Коды для всех клавиш управления курсором и клавиш числовой клавиатуры представлены в виде констант класса `flash.ui.Keyboard`. Приведенный выше код обычно записывается следующим образом (обратите внимание на использование статической переменной `ESCAPE` класса `Keyboard` вместо значения литерала 27):

```
import flash.ui.Keyboard;
private function keyDownListener (e:KeyboardEvent):void {
  if (e.keyCode == Keyboard.ESCAPE) {
    trace("The Escape key was pressed");
  }
}
```

Коды клавиш зависят от выбранного языка и используемой операционной системы.

Список кодов клавиш на клавиатурах с раскладкой U.S. English можно найти по адресу http://livedocs.macromedia.com/flash/8/main/00001686.html.

При написании выражений для определения момента нажатия или отпускания клавиши, отсутствующей среди констант класса `flash.ui.Keyboard`, всегда используйте такую последовательность действий.

1. Запустите приложение `KeyViewer` из листинга 22.11 на компьютере с такой же операционной системой и клавиатурой, как у конечного пользователя.
2. Нажмите желаемую клавишу.
3. Сохраните возвращенный код клавиши в виде константы.
4. Используйте константу, полученную на шаге 3, при определении момента нажатия или отпускания желаемой клавиши.

Предположим, что мы хотим определить момент, когда пользователь нажимает клавишу A на компьютере с операционной системой Mac OS и клавиатурой с раскладкой U.S. English. Мы запускаем приложение `KeyViewer` и нажимаем клавишу A. В ответ приложение `KeyViewer` отображает код клавиши — 65. Мы сохраняем этот код в виде константы пользовательского класса, например с именем `KeyConstants`, как показано в следующем коде:

```
public static const A_KEY:int = 65;
```

Затем, чтобы определить момент нажатия клавиши A, мы используем следующий код:

```
private function keyDownListener (e:KeyboardEvent):void {
  if (e.keyCode == KeyConstants.A_KEY) {
    trace("The A key was pressed");
  }
}
```

Следующий код демонстрирует описанную методику в контексте очень простого тестового приложения:

```
package {
  import flash.display.*;
  import flash.events.*;

  public class AKeySensor extends Sprite {
    //
    public static const A_KEY:int = 65;

    public function AKeySensor ( ) {
      stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
    }

    private function keyDownListener (e:KeyboardEvent):void {
      if (e.keyCode == AKeySensor.A_KEY) {
        trace("The A key was pressed");
      }
    }
  }
}
```

Стоит отметить, что, когда используется редактор методов ввода (IME — Input Method Editor), переменная экземпляра `keyCode` класса `KeyboardEvent` не

поддерживается. Дополнительную информацию о редакторах IME можно найти в описании класса `flash.system.IME` справочника по языку ActionScript корпорации Adobe и в разделе Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Client System Environment ▸ IME class документации корпорации Adobe.

Клавиши, расположенные в нескольких местах. На некоторых клавиатурах определенные клавиши, расположенные в нескольких местах, имеют одинаковые коды. Например, на компьютере с операционной системой Mac OS и клавиатурой с раскладкой U.S. English код клавиши 16 представляет и левую, и правую клавиши Shift; код клавиши 17 представляет и левую, и правую клавиши Ctrl; код клавиши 13 представляет и основную клавишу Enter, и клавишу Enter на числовой клавиатуре. Чтобы отличать подобные клавиши, расположенные в нескольких местах, используется переменная экземпляра `keyLocation` класса `KeyboardEvent`, значение которой обозначает логическое положение клавиши, представленное в виде одной из четырех констант класса `flash.ui.KeyLocation` (`LEFT`, `NUM_PAD`, `RIGHT` и `STANDARD`). Эту методику демонстрирует следующий код. В нем функция-приемник события `KeyboardEvent.KEY_DOWN` выводит одно отладочное сообщение в момент нажатия левой клавиши Shift, а другое — в момент нажатия правой клавиши Shift:

```
private function keyDownListener (e:KeyboardEvent):void {
  if (e.keyCode == Keyboard.SHIFT) {
    if (e.keyLocation == KeyLocation.LEFT) {
      trace("The left Shift key was pressed");
    } else if (e.keyLocation == KeyLocation.RIGHT) {
      trace("The right Shift key was pressed");
    }
  }
}
```

Определение одновременного нажатия нескольких клавиш

Чтобы определить нажатие клавиши Shift или Ctrl (клавиша Command на компьютерах Macintosh) в сочетании с любой другой клавишей, мы используем переменные экземпляра `shiftKey` и `ctrlKey` класса `KeyboardEvent` в функции-приемнике события `KeyboardEvent.KEY_DOWN`. Например, следующее простое приложение определяет нажатие сочетания клавиш Ctrl+S (Command+S на компьютерах Macintosh):

```
package {
  import flash.display.*;
  import flash.events.*;

  public class CtrlSSensor extends Sprite {
    public static const S_KEY:int = 83;

    public function CtrlSSensor ( ) {
      stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
    }

    private function keyDownListener (e:KeyboardEvent):void {
      if (e.keyCode == CtrlSSensor.S_KEY
```

```
            && e.ctrlKey == true) {
            trace("Ctrl+S was pressed");
        }
    }
  }
}
```

В автономной версии приложения Flash Player и в его версии, реализованной в виде модуля расширения браузера, среда выполнения не может определить нажатие клавиши Alt (и клавиши F10).

Чтобы определить одновременное нажатие двух или более обычных клавиш, не являющихся Shift и Ctrl, мы должны вручную отслеживать текущее состояние каждой клавиши. Эту методику демонстрирует следующий код. В нем приложение отображает отладочное сообщение, когда происходит одновременное нажатие клавиш ← и ↑. Подобный код может использоваться для поворота автомобиля или для перемещения по диагонали космического корабля в компьютерной игре.

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.ui.*;

  // Определяет одновременное нажатие клавиш «стрелка вверх» и «стрелка влево»
  public class UpLeftSensor extends Sprite {
    // Отслеживает состояние стрелки «вверх»
    // (true, когда нажата; false в противном случае)
    private var upPressed:Boolean;
    // Отслеживает состояние стрелки «влево»
    // (true, когда нажата; false в противном случае)
    private var leftPressed:Boolean;

    public function UpLeftSensor ( ) {
      // Регистрируем приемники для событий клавиатуры
      stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
      stage.addEventListener(KeyboardEvent.KEY_UP, keyUpListener);
    }

    // Обрабатывает события KeyboardEvent.KEY_DOWN
    private function keyDownListener (e:KeyboardEvent):void {
      // Делаем пометку, была ли нажата клавиша «стрелка вверх» или «стрелка влево»
      if (e.keyCode == Keyboard.UP) {
        upPressed = true;
      } else if (e.keyCode == Keyboard.LEFT) {
        leftPressed = true;
      }

      // Если клавиши «стрелка вверх» и «стрелка влево» были нажаты одновременно...
      if (upPressed && leftPressed) {
        // ...выполняем определенное действие, например поворачиваем
        // космический корабль по диагонали вверх и влево
```

```
        trace("Up Arrow key and Left Arrow key are both pressed");
      }
    }

    // Обрабатывает события KeyboardEvent.KEY_UP
    private function keyUpListener (e:KeyboardEvent):void {
      // Делаем пометку, была ли отпущена клавиша «стрелка вверх»
      // или «стрелка влево»
      if (e.keyCode == Keyboard.UP) {
        upPressed = false;
      } else if (e.keyCode == Keyboard.LEFT) {
        leftPressed = false;
      }
    }
  }
}
```

События мыши и клавиши-модификаторы

Подобно тому как язык ActionScript позволяет легко проверить, были ли нажаты клавиши Shift или Ctrl в процессе диспетчеризации события ввода с клавиатуры, он также дает возможность узнать, были ли нажаты клавиши Shift или Ctrl в процессе диспетчеризации события ввода с помощью мыши. Для этого внутри функции-приемника, зарегистрированной для данного события, используются переменные экземпляра `shiftKey` и `ctrlKey` класса `MouseEvent`. Например, следующий код выводит отладочное сообщение, когда происходит щелчок кнопкой мыши при нажатой клавише Shift. Подобный код может быть использован в программе рисования для ограничения перемещения объекта по горизонтальной или вертикальной оси.

```
package {
  import flash.display.*;
  import flash.events.*;

  public class ControlClickSensor extends Sprite {
    public function ControlClickSensor ( ) {
      stage.addEventListener(MouseEvent.CLICK, clickListener);
    }

    private function clickListener (e:MouseEvent):void {
      if (e.shiftKey) {
        trace("Shift+click detected");
      }
    }
  }
}
```

Определение символа, связанного с клавишей

Ранее рассказывалось, как можно определить код последней нажатой или отпущенной клавиши. Чтобы получить фактический символ, связанный с последней

нажатой или отпущенной клавишей, мы проверяем значение переменной экземпляра `charCode` класса `KeyboardEvent` внутри функции-приемника события `KeyboardEvent.KEY_UP` или `KeyboardEvent.KEY_DOWN`.

Когда используется клавиатура с раскладкой U.S. English, значение переменной `charCode` представляет код символа из таблицы ASCII, который логически соответствует последней нажатой или отпущенной клавише. В некоторых случаях для одной и той же клавиши переменная `charCode` может принимать два возможных значения в зависимости от того, была ли нажата клавиша Shift. Например, код символа для клавиши S на клавиатуре с раскладкой U.S. English соответствует значению 115 при отпущенной клавише Shift и значению 83 при нажатой клавише Shift. Для клавиш, у которых нет соответствующих отображаемых символов в таблице ASCII, переменной `KeyboardEvent.charCode` устанавливается значение `0`.

Когда используется клавиатура с другой раскладкой, отличной от раскладки U.S. English, значение переменной `charCode` представляет код символа в таблице ASCII для эквивалентной клавиши на клавиатуре с раскладкой U.S. English. Например, на японской клавиатуре для клавиши с иероглифом ち, которая находится в той же позиции, что и клавиша A на клавиатуре с раскладкой U.S. English, переменная `charCode` будет по-прежнему возвращать либо значение `97`, либо значение `65` (символы «a» и «A» из таблицы ASCII соответственно), но не значение `12 385` (кодовая точка Unicode для иероглифа ち).

Чтобы преобразовать код символа в фактическую строку, используется метод экземпляра `fromCharCode( )` класса `String`. Эта методика продемонстрирована в листинге 22.12 — измененный класс `KeyViewer` (который был представлен ранее в листинге 22.11) отображает символ, связанный с последней нажатой клавишей.

Листинг 22.12. Получение кода клавиши и кода символа нажатой клавиши

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;
  import flash.ui.*;

  // Отображает код клавиши и код символа для любой нажатой клавиши
  public class KeyViewer extends Sprite {
    private var keyoutput:TextField;
    public function KeyViewer ( ) {
      keyoutput = new TextField( );
      keyoutput.text = "Press any key...";
      keyoutput.autoSize = TextFieldAutoSize.LEFT;
      keyoutput.border     = true;
      keyoutput.background = true;
      addChild(keyoutput);

      stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
    }

    private function keyDownListener (e:KeyboardEvent):void {
      keyoutput.text = "The key code for the key you pressed is: "
```

```
                    + e.keyCode + "\n";
        keyoutput.appendText("The character code for the key you pressed is: "
                    + e.charCode + "\n");
        keyoutput.appendText("The character for the key you pressed is: "
                    + String.fromCharCode(e.charCode));
    }
  }
}
```

Результат выполнения приложения `KeyViewer` из листинга 22.12 для нажатой клавиши **S** на клавиатуре с раскладкой U.S. English выглядит следующим образом:

```
The key code for the key you pressed is: 83
The character code for the key you pressed is: 115
The character for the key you pressed is: s
```

Результат выполнения приложения `KeyViewer` из листинга 22.12 для нажатой клавиши **Shift** в сочетании с клавишей **S** на клавиатуре с раскладкой U.S. English выглядит следующим образом:

```
The key code for the key you pressed is: 83
The character code for the key you pressed is: 83
The character for the key you pressed is: S
```

Как и в случае с переменной экземпляра `keyCode` класса `KeyboardEvent`, переменная `charCode` не поддерживается, если используется редактор методов ввода, и не предназначена для использования в качестве средства получения вводимых текстовых данных. Для получения вводимых текстовых данных применяется событие `TextEvent.TEXT_INPUT` совместно с объектом `TextField`, которые рассматриваются в следующем разделе.

События текстового ввода

Приложение Flash Player выполняет диспетчеризацию событий текстового ввода в следующих ситуациях:

- когда пользователь добавляет новый текст в поле;
- когда пользователь активизирует гипертекстовую ссылку протокола **event:** в поле (щелкая кнопкой мыши на этой ссылке);
- когда происходит прокрутка текстового поля либо программным путем, либо пользователем.

В табл. 22.4 перечислены типы внутренних событий текстового ввода приложения Flash Player. Для каждого типа события столбец «Тип события» содержит константу класса, представляющую официальное строковое название типа события. В столбце «Описание» указывается конкретное действие пользователя, приводящее к возникновению этого события. Столбец «Получатель» описывает объект, который выступает в роли получателя события при его диспетчеризации. В столбце «Поведение по умолчанию» приводится стандартная реакция приложения Flash Player на данное событие. Столбец «Всплывает» содержит информацию о том,

имеет ли данное событие фазу всплытия. Наконец, столбец «Тип данных объекта, передаваемого в функцию-приемник» определяет тип данных объекта, передаваемого в функцию-приемник, обрабатывающую данное событие.

Таблица 22.4. События текстового ввода приложения Flash Player

Тип события	Описание	Получатель	Поведение по умолчанию	Всплывает	Тип данных объекта, передаваемого в функцию-приемник
TextEvent. TEXT_INPUT	Пользователь попытался добавить новый текст в поле	Объект TextField, в который пользователь попытался добавить новый текст	Текст добавляется в поле. Данное поведение по умолчанию может быть отменено методом экземпляра preventDefault( ) класса Event	Да	TextEvent
Event. CHANGE	Новый текст был добавлен пользователем в поле	Объект TextField, в который был добавлен новый текст	Отсутствует	Да	Event
Event. SCROLL	Произошла прокрутка поля либо программным путем, либо пользователем	Объект TextField, в котором произошла прокрутка	Отсутствует	Нет	Event
TextEvent. LINK	Активизирована гипертекстовая ссылка протокола event:	Объект TextField, содержащий активизированную ссылку	Отсутствует	Да	TextEvent

Подобно событиям мыши, клавиатуры и фокуса, событие текстового ввода может обрабатываться приемниками, зарегистрированными в получателе события или в любом отображаемом предке данного получателя. Тем не менее в следующих разделах мы сосредоточимся только на использовании приемников, зарегистрированных в получателе события.

Поближе познакомимся с четырьмя событиями, представленными в табл. 22.4.

События TextEvent.TEXT_INPUT и Event.CHANGE

События `TextEvent.TEXT_INPUT` и `Event.CHANGE` позволяют определить новый введенный пользователем текст. В частности, эти события могут возникать в результате использования таких методик, предназначенных для ввода текста, как:

- нажатие клавиши;
- вставка текста с помощью специальных сочетаний клавиш или встроенного контекстного меню приложения Flash Player (открываемого щелчком вспомогательной (правой) кнопки мыши);
- ввод текста через программное обеспечение для распознавания речи;
- создание текстового содержимого в редакторе методов ввода.

Событие TextEvent.TEXT_INPUT сообщает о том, что пользователь пытается добавить новый текст в поле, и дает приложению возможность либо пресечь эту попытку, либо разрешить ее. Это событие предоставляет удобный способ для обращения к текстовому содержимому, добавляемому в текстовое поле, до того как оно на самом деле будет добавлено туда. В отличие от этого, событие Event.CHANGE сообщает о том, что попытка пользователя добавить новый текст в поле завершилась успешно и приложение Flash Player соответствующим образом обновило содержимое данного текстового поля.

Обобщенный код, необходимый для регистрации приемника в объекте TextField для события TextEvent.TEXT_INPUT, выглядит следующим образом:

```
объектTextField.addEventListener(TextEvent.TEXT_INPUT, textInputListener);
```

Обобщенный код для приемника события TextEvent.TEXT_INPUT выглядит следующим образом:

```
private function textInputListener (e:TextEvent):void {
}
```

Чтобы предотвратить отображение введенного пользователем текста в объекте TextField, используется метод экземпляра preventDefault( ) класса Event, как показано в следующем коде:

```
private function textInputListener (e:TextEvent):void {
  // Предотвращаем отображение введенного пользователем текста на экране
  e.preventDefault( );
}
```

Для обращения к тексту, введенному пользователем, применяется переменная экземпляра text класса TextEvent, как показано в следующем коде:

```
private function textInputListener (e:TextEvent):void {
  // Выводим отладочное сообщение, содержащее введенный пользователем текст
  trace(e.text);
}
```

Событие TextEvent.TEXT_INPUT может быть использовано для автоматического форматирования вводимых пользователем данных при заполнении анкет в приложении, как показано в листинге 22.13. В этом примере весь текст, введенный в поле, преобразуется в верхний регистр. Подобный код может быть использован в разделе «адрес доставки» формы, с помощью которой оформляется заказ товара через Интернет.

Листинг 22.13. Преобразование введенного пользователем текста в верхний регистр

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  public class UppercaseConverter extends Sprite {
    private var inputfield:TextField;

    public function UppercaseConverter ( ) {
      inputfield = new TextField( );
```

```
        inputfield.text = "";
        inputfield.width  = 150;
        inputfield.height = 30;
        inputfield.border     = true;
        inputfield.background = true;
        inputfield.type = TextFieldType.INPUT;
        addChild(inputfield);

        // Регистрируем приемник в объекте inputfield
        // для событий TextEvent.TEXT_INPUT
        inputfield.addEventListener(TextEvent.TEXT_INPUT, textInputListener);
    }

    // Вызывается всякий раз, когда пользователь пытается добавить новый
    // текст в объект inputfield
    private function textInputListener (e:TextEvent):void {
      // Предотвращаем добавление введенного пользователем текста
      // в поле
      e.preventDefault( );
      // Добавляем в поле эквивалентный символ верхнего регистра
      inputfield.replaceText(inputfield.caretIndex,
                             inputfield.caretIndex,
                             e.text.toUpperCase( ));
      // Устанавливаем точку ввода (знак вставки) в конец нового текста,
      // чтобы пользователь думал, что это он ввел данный текст
      var newCaretIndex:int = inputfield.caretIndex + e.text.length;
      inputfield.setSelection(newCaretIndex, newCaretIndex);
    }
  }
}
```

Теперь перейдем к событию `Event.CHANGE`. Обобщенный код, необходимый для регистрации приемника в объекте `TextField` для события `Event.CHANGE`, выглядит следующим образом:

```
объектTextField.addEventListener(Event.CHANGE, changeListener);
```

Обобщенный код для приемника события `Event.CHANGE` выглядит таким образом:

```
private function changeListener (e:Event):void {
}
```

Событие `Event.CHANGE` может быть использовано для синхронизации содержимого двух текстовых полей, как показано в листинге 22.14. Здесь представлен фрагмент из гипотетического элемента управления «панель», содержащего помеченные фотографии. Для простоты код из листинга позволяет добавлять только метку фотографии без соответствующего изображения. В коде событие `Event.CHANGE` применяется для обновления строки заголовка панели в соответствии с названием текущей выбранной фотографии. Для обзора в этом коде также используются события `FocusEvent.FOCUS_IN` и `FocusEvent.FOCUS_OUT`, чтобы обновлять заголовок панели, когда пользователь меняет фокус в приложении.

Листинг 22.14. Синхронизация двух объектов TextField

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  public class PhotoPanel extends Sprite {
    private static const defaultTitle:String =
                                        "Photo Viewer [No photo selected]";
    private static const defaultPhotoName:String =
                                        "Enter Photo Name Here";

    private var title:TextField;
    private var photoname:TextField;

    public function PhotoPanel ( ) {
      // Создаем объект TextField для строки заголовка панели
      title = new TextField( );
      title.text = PhotoPanel.defaultTitle;
      title.width  = 350;
      title.height = 25;
      title.border     = true;
      title.background = true;
      title.selectable = false;
      addChild(title);

      // Создаем объект TextField для заголовка отдельной фотографии
      photoname = new TextField( );
      photoname.text = PhotoPanel.defaultPhotoName;
      photoname.width  = 150;
      photoname.height = 30;
      photoname.x = 100;
      photoname.y = 150;
      photoname.border     = true;
      photoname.background = true;
      photoname.type = TextFieldType.INPUT
      addChild(photoname);

      // Регистрируем приемник в объекте photoname
      // для событий Event.CHANGE
      photoname.addEventListener(Event.CHANGE, changeListener);

      // Регистрируем приемники в объекте photoname для событий получения
      // и потери фокуса
      photoname.addEventListener(FocusEvent.FOCUS_IN, photoFocusInListener);
      photoname.addEventListener(FocusEvent.FOCUS_OUT,
                                 photoFocusOutListener);

      // Регистрируем приемник в объекте stage
      // для событий потери фокуса
      stage.addEventListener(FocusEvent.FOCUS_OUT, panelFocusOutListener);
    }
```

```
    // Вызывается всякий раз при добавлении
    // нового текста в объект photoname
    private function changeListener (e:Event):void {
      // Название фотографии изменилось, поэтому обновляем строку заголовка,
      // чтобы ее содержимое соответствовало тексту объекта photoname
      if (photoname.text.length == 0) {
        title.text = "Photo Viewer [Unnamed Photo]";
      } else {
        title.text = "Photo Viewer [" + photoname.text + "]";
      }
    }

    // Вызывается всякий раз при получении фокуса объектом photoname
    private function photoFocusInListener (e:FocusEvent):void {
      // Если для фотографии еще не было указано название...
      if (photoname.text == PhotoPanel.defaultPhotoName) {
        // ...очищаем поле photoname, чтобы пользователь мог ввести название
        photoname.text = "";
        // Обновляем строку заголовка панели, чтобы показать, что была
        // выбрана фотография без названия
        title.text = "Photo Viewer [Unnamed Photo]";
      } else {
        // ...выбранной фотографии уже присвоено название, поэтому обновляем
        // строку заголовка панели, чтобы отобразить это название
        title.text = "Photo Viewer [" + photoname.text + "]";
      }
    }

    // Вызывается всякий раз, когда объект photoname теряет фокус
    private function photoFocusOutListener (e:FocusEvent):void {
      // Если пользователь не ввел название для фотографии...
      if (photoname.text.length == 0) {
        // ...присваиваем названию фотографии значение по умолчанию
        photoname.text = PhotoPanel.defaultPhotoName;
      }
    }

    // Вызывается всякий раз, когда фокус теряет любой объект
    private function panelFocusOutListener (e:FocusEvent):void {
      // Если на настоящий момент ни один объект не имеет фокуса...
      if (e.relatedObject == null) {
        // ...присваиваем строке заголовка панели значение по умолчанию
        title.text = PhotoPanel.defaultTitle;
      }
    }
  }
}
```

Событие Event.SCROLL

Событие `Event.SCROLL` возникает всякий раз, когда изменяется значение любой из следующих переменных объекта `TextField`: `scrollH`, `scrollV`, `maxscrollH`

или maxscrollV. Другими словами, событие Event.SCROLL сообщает об одном из следующих изменений в текстовом поле:

- произошла прокрутка поля по вертикали или по горизонтали (либо пользователем, либо программным путем с помощью переменных scrollH или scrollV);
- поле получило новое содержимое, которое изменяет максимальный диапазон для вертикальной или горизонтальной прокрутки;
- изменение размеров поля привело к изменению максимального диапазона для вертикальной или горизонтальной прокрутки поля.

Обобщенный код, необходимый для регистрации приемника в объекте TextField для события Event.SCROLL, выглядит следующим образом:

```
объектTextField.addEventListener(Event.SCROLL, scrollListener);
```

Обобщенный код для приемника события Event.SCROLL выглядит следующим образом:

```
private function scrollListener (e: Event):void {
}
```

Обычно событие Event.SCROLL применяется для синхронизации интерфейса полосы прокрутки с содержимым поля, как показано в листинге 22.15. Полоса прокрутки в этом листинге обладает следующими возможностями:

- может применяться к любому объекту TextField;
- может перетаскиваться с помощью мыши для прокрутки поля по вертикали;
- автоматически обновляется в ответ на изменения в размерах поля, его содержимого или позиции прокрутки.

Тем не менее для простоты данная полоса прокрутки не включает кнопки для прокручивания содержимого вниз и вверх. В листинге 22.15 применяются многие методики, которые были рассмотрены в этой главе, а также несколько еще не описанных методик. Там, где это необходимо, указаны перекрестные ссылки на дополнительные темы.

Листинг 22.15. Использование события Event.SCROLL для реализации полосы прокрутки

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;
  import flash.utils.*;
  import flash.geom.*;

  // Простая полоса прокрутки с возможностью перетаскивания, которая
  // автоматически обновляется в ответ на изменения в размерах заданного
  // текстового поля.
  // Использование:
  //   var theTextField:TextField = new TextField( );
  //   someContainer.addChild(theTextField);
  //   var scrollbar:ScrollBar = new ScrollBar(theTextField);
  //   someContainer.addChild(scrollbar);
```

```
public class ScrollBar extends Sprite {
  // Текстовое поле, к которому применяется данная полоса прокрутки
  private var t:TextField;
  // Текущая высота текстового поля. Если высота текстового поля
  // изменяется, мы обновляем высоту данной полосы прокрутки.
  private var tHeight:Number;
  // Фоновое изображение для полосы прокрутки
  private var scrollTrack:Sprite;
  // Перетаскиваемый ползунок полосы прокрутки
  private var scrollThumb:Sprite;
  // Ширина полосы прокрутки
  private var scrollbarWidth:int = 15;
  // Минимальная высота ползунка полосы прокрутки
  private var minimumThumbHeight:int = 10;
  // Флажок, который обозначает, перетаскивает ли пользователь ползунок
  // в данный момент
  private var dragging:Boolean = false;
  // Флажок, который обозначает, нужно ли перерисовывать полосу прокрутки
  // на следующем запланированном этапе обновления экрана
  private var changed:Boolean = false;

  // Конструктор.
  // @param textfield Объект TextField, к которому применяется данная
  //                  полоса прокрутки.
  public function ScrollBar (textfield:TextField) {
    // Сохраняем ссылку на объект TextField, к которому применяется данная
    // полоса прокрутки
    t = textfield;
    // Запоминаем высоту текстового поля, чтобы мы могли отслеживать
    // изменения, требующие перерисовки полосы прокрутки.
    tHeight = t.height;

    // Создаем фон полосы прокрутки
    scrollTrack = new Sprite( );
    scrollTrack.graphics.lineStyle( );
    scrollTrack.graphics.beginFill(0x333333);
    scrollTrack.graphics.drawRect(0, 0, 1, 1);
    addChild(scrollTrack);

    // Создаем перетаскиваемый ползунок на полосе прокрутки
    scrollThumb = new Sprite( );
    scrollThumb.graphics.lineStyle( );
    scrollThumb.graphics.beginFill(0xAAAAAA);
    scrollThumb.graphics.drawRect(0, 0, 1, 1);
    addChild(scrollThumb);

    // Регистрируем приемник события Event.SCROLL, который будет обновлять
    // полосу прокрутки в соответствии с текущей позицией прокрутки поля
    t.addEventListener(Event.SCROLL, scrollListener);

    // Регистрируем в объекте scrollThumb приемник для событий нажатия
    // кнопки мыши, который будет вызывать перетаскивание ползунка
    scrollThumb.addEventListener(MouseEvent.MOUSE_DOWN,mouseDownListener);
```

```
        // Регистрируем приемник для получения уведомлений, когда данный
        // объект добавляется в список отображения или удаляется из него
        // (требуется пользовательский вспомогательный класс StageDetector).
        // Когда данный объект добавляется в список отображения, регистрируем
        // приемники для событий перемещения мыши и отпускания кнопки мыши
        // на уровне объекта stage, которые будут управлять операцией
        // перетаскивания ползунка.
        var stageDetector:StageDetector = new StageDetector(this);
        stageDetector.addEventListener(StageDetector.ADDED_TO_STAGE,
                                       addedToStageListener);
        stageDetector.addEventListener(StageDetector.REMOVED_FROM_STAGE,
                                       removedFromStageListener);

        // Регистрируем приемник для получения уведомлений перед каждым
        // обновлением экрана. Перед обновлением экрана проверяем,
        // требуется ли перерисовка полосы прокрутки. Информацию по событию
        // Event.ENTER_FRAME можно найти в гл. 24.
        addEventListener(Event.ENTER_FRAME, enterFrameListener);

        // Инициируем первоначальную прорисовку полосы прокрутки.
        changed = true;
    }

    // Выполняется всякий раз, когда данный объект добавляется
    // в список отображения
    private function addedToStageListener (e:Event):void {
        // Регистрируем приемники для «глобальных» событий перемещения мыши
        // и отпускания кнопки мыши
        stage.addEventListener(MouseEvent.MOUSE_UP, mouseUpListener);
        stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
    }

    // Выполняется всякий раз, когда данный объект удаляется
    // из списка отображения
    private function removedFromStageListener (e:Event):void {
        // Отменяем регистрацию приемников для «глобальных» событий
        // перемещения мыши и отпускания кнопки мыши
        stage.removeEventListener(MouseEvent.MOUSE_UP, mouseUpListener);
        stage.removeEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
    }

    // Выполняется один раз для каждого обновления экрана. Этот метод
    // проверяет, произошли ли какие-либо изменения в позиции прокрутки
    // текстового поля, его содержимом или размерах с момента последней
    // перерисовки полосы прокрутки. Если да, то перерисовываем полосу
    // прокрутки. Выполняя эту проверку – «рисовать или нет» – всего один
    // раз перед обновлением экрана, мы избавляемся от ненужных вызовов
    // метода updateScrollbar( ), а также избегаем проблем в приложении
    // Flash Player, связанных с задержкой при изменении значения переменной
    // TextField.maxScrollV.
    private function enterFrameListener (e:Event):void {
        // Если высота поля изменилась, запрашиваем перерисовку
        // полосы прокрутки
```

```
    if (t.height != tHeight) {
      changed = true;
      tHeight = t.height;
      // Высота изменилась, поэтому прекращаем любую происходящую
      // в настоящий момент операцию перетаскивания. Пользователь
      // будет вынужден щелкнуть снова, чтобы начать операцию
      // перетаскивания ползунка после перерисовки
      // полосы прокрутки.
      if (dragging) {
        scrollThumb.stopDrag( );
        dragging = false;
      }
    }

    // Если требуется перерисовка полосы прокрутки...
    if (changed) {
      // ...вызываем процедуру отрисовки полосы прокрутки
      updateScrollbar( );
      changed = false;
    }
  }

  // Обрабатывает события Event.SCROLL
  private function scrollListener (e:Event):void {
    // В определенных случаях, когда строки удаляются из текстового поля,
    // приложение Flash Player выполняет диспетчеризацию двух событий:
    // одно событие связано с уменьшением значения переменной maxScrollV
    // (диспетчеризация осуществляется немедленно), а другое –
    // с уменьшением значения переменной scrollV (диспетчеризация
    // осуществляется спустя несколько обновлений экрана). В подобных
    // ситуациях переменная scrollV некоторое время имеет неправильное
    // значение, которое превышает значение переменной maxScrollV.
    // В качестве обходного пути мы игнорируем событие, возникающее
    // после изменения значения переменной maxScrollV, и ожидаем
    // появления события, возникающего в результате изменения значения
    // переменной scrollV (в противном случае отрисованная полоса
    // прокрутки некоторое время не будет соответствовать фактическому
    // содержимому поля).
    if (t.scrollV > t.maxScrollV) {
      return;
    }

    // Если на данный момент пользователь не перетаскивал ползунок полосы
    // прокрутки, помечаем, что данная полоса прокрутки должна быть
    // перерисована на следующем запланированном этапе обновления экрана.
    // (Если в данный момент пользователь перетаскивает ползунок,
    // изменение прокрутки, вызвавшее данное событие, явилось результатом
    // перетаскивания ползунка в новую позицию, поэтому обновлять полосу
    // прокрутки не нужно, поскольку ползунок уже находится в правильной
    // позиции.)
    if (!dragging) {
```

```
      changed = true;
    }
  }
  // Устанавливает размеры и позицию фонового изображения и ползунка
  // полосы прокрутки в соответствии с размерами и содержимым связанного
  // текстового поля. Информацию о переменных scrollV, maxScrollV
  // и bottomScrollV класса TextField можно найти в справочнике
  // по языку ActionScript корпорации Adobe.
  public function updateScrollbar ( ):void {
    // Устанавливаем размеры и позицию фонового изображения полосы прокрутки.
    // Данный код всегда помещает полосу прокрутки справа от поля.
    scrollTrack.x = t.x + t.width;
    scrollTrack.y = t.y;
    scrollTrack.height = t.height;
    scrollTrack.width = scrollbarWidth;

    // Проверяем количество видимых на экране строк текстового поля
    var numVisibleLines:int = t.bottomScrollV - (t.scrollV-1);
    // Если часть строк в текстовом поле не видна на экране...
    if (numVisibleLines < t.numLines) {
      // ...отображаем ползунок
      scrollThumb.visible = true;
      // Теперь устанавливаем размеры ползунка
      // Высота ползунка определяется процентным соотношением
      // отображаемых строк, умноженным на высоту текстового поля
      var thumbHeight:int = Math.floor(t.height *
                                       (numVisibleLines/t.numLines));
      // Высота ползунка не должна быть меньше значения minimumThumbHeight
      scrollThumb.height = Math.max(minimumThumbHeight, thumbHeight);
      scrollThumb.width = scrollbarWidth;

      // Теперь устанавливаем позицию ползунка
      scrollThumb.x = t.x + t.width;
      //   Вертикальная позиция ползунка определяется количеством
      //   прокрученных строк в поле, выраженным в процентах и умноженным
      //   на высоту «пространства полосы» (это высота
      //   полосы прокрутки за вычетом высоты ползунка).
      scrollThumb.y = t.y + (scrollTrack.height-scrollThumb.height)
                      * ((t.scrollV-1)/(t.maxScrollV-1));
    } else {
      // Если в данный момент на экране отображаются все строки текстового
      // поля, прячем ползунок полосы прокрутки
      scrollThumb.visible = false;
    }
  }

  // Устанавливает вертикальную позицию прокрутки текстового поля
  // в соответствии с относительной позицией ползунка
  public function synchTextToScrollThumb ( ):void {
    var scrollThumbMaxY:Number = t.height-scrollThumb.height;
    var scrollThumbY:Number = scrollThumb.y-t.y;
```

```
      t.scrollV = Math.round(t.maxScrollV
                             * (scrollThumbY/scrollThumbMaxY));
    }
    // Выполняется, когда пользователь нажимает основную кнопку мыши
    // над ползунком
    private function mouseDownListener (e:MouseEvent):void {
      // Начинаем операцию перетаскивания ползунка. (Метод startDrag( )
      // наследуется из класса Sprite.)
      var bounds:Rectangle = new Rectangle(t.x + t.width,
                                           t.y,
                                           0,
                                           t.height-scrollThumb.height);
      scrollThumb.startDrag(false, bounds);
      dragging = true;
    }

    // Выполняется, когда пользователь отпускает основную кнопку мыши
    // (в любом месте области отображения приложения Flash Player и даже
    // за ее пределами)
    private function mouseUpListener (e:MouseEvent):void {
      // Если пользователь перетащил ползунок, обновляем вертикальную
      // позицию прокрутки поля, после чего прекращаем операцию
      // перетаскивания ползунка
      if (dragging) {
        synchTextToScrollThumb( );
        scrollThumb.stopDrag( );
        dragging = false;
      }
    }

    // Выполняется при перемещении указателя мыши (над областью отображения
    // приложения Flash Player)
    private function mouseMoveListener (e:MouseEvent):void {
      // Если пользователь перетаскивает ползунок, устанавливаем
      // вертикальную позицию прокрутки в соответствии с относительной
      // позицией ползунка
      if (dragging) {
        synchTextToScrollThumb( );
      }
    }
  }
}
```

В листинге 22.16 представлен простой тестовый класс, который демонстрирует использование класса `ScrollBar` из листинга 22.15.

Листинг 22.16. Демонстрационный класс, представляющий использование класса ScrollBar

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;
  import flash.utils.*;
```

```
  // Демонстрирует использование класса ScrollBar
  public class ScrollBarDemo extends Sprite {
    public function ScrollBarDemo ( ) {
      // Создаем объект TextField
      var inputfield:TextField = new TextField( );
      // Заполняем текстовое поле начальным содержимым
      inputfield.text = "1\n2\n3\n4\n5\n6\n7\n8\n9";
      inputfield.height = 50;
      inputfield.width  = 100;
      inputfield.border     = true;
      inputfield.background = true;
      inputfield.type = TextFieldType.INPUT;
      inputfield.multiline = true;
      addChild(inputfield);

      // Создаем полосу прокрутки и связываем ее
      // с объектом TextField
      var scrollbar:ScrollBar = new ScrollBar(inputfield);
      addChild(scrollbar);
    }
  }
}
```

Событие TextEvent.LINK

Событие `TextEvent.LINK` применяется для вызова кода на языке ActionScript, когда пользователь щелкает кнопкой мыши на гипертекстовой ссылке в объекте `TextField`. Это событие возникает в тех случаях, когда пользователь щелкает кнопкой мыши на гипертекстовой ссылке, адрес URL которой начинается с псевдопротокола **event**.

Введение в использование гипертекстовых ссылок в текстовых полях языка ActionScript можно найти в гл. 27.

Обобщенный код, необходимый для создания гипертекстовой ссылки, которая вызывает код на языке ActionScript, выглядит следующим образом:

объектTextField.htmlText = "*текстСсылки*";

В предыдущем коде *объектTextField* — это объект `TextField`, содержащий ссылку, а *текстСсылки* — текст, который отображается на экране и на котором щелкает кнопкой мыши пользователь. Когда пользователь щелкает кнопкой мыши на тексте *текстСсылки*, приложение Flash Player выполняет все приемники, зарегистрированные в объекте *объектTextField* или в его отображаемых предках для события `TextEvent.LINK`. В каждый приемник передается объект `TextEvent`, значением переменной `text` которого является указанная строка *содержимоеСсылки*. Эта строка обычно обозначает операцию языка ActionScript, которая должна быть выполнена при щелчке кнопкой мыши на ссылке.

Обобщенный код, необходимый для регистрации приемника в объекте `TextField` для события `TextEvent.LINK`, выглядит следующим образом:

```
объектTextField.addEventListener(TextEvent.LINK, приемник);
```

Обобщенный код для приемника события `TextEvent.LINK` выглядит следующим образом:

```
private function приемник (e:TextEvent):void {
}
```

Используя приведенный обобщенный код в качестве подсказки, создадим пример гипертекстовой ссылки, которая запускает игру, когда пользователь щелкает на ней кнопкой мыши. Вот код для этой ссылки: обратите внимание, что задаваемая строка *содержимоеСсылки* обозначает название операции, вызываемой этой ссылкой: `startGame`.

```
var t:TextField = new TextField( );
t.htmlText = "<a href='event:startGame'>Play now!</a>";
t.autoSize = TextFieldAutoSize.LEFT;
addChild(t);
```

Следующий код регистрирует метод `linkListener( )` в предыдущем объекте `TextField` — `t` — для событий `TextEvent.LINK`:

```
t.addEventListener(TextEvent.LINK, linkListener);
```

Наконец, следующий код демонстрирует метод `linkListener( )`, который выполняется при щелчке кнопкой мыши на создаваемой ссылке. Внутри метода `linkListener( )` мы выполняем операцию, определенную в задаваемой строке *содержимоеСсылки*, к которой можно обратиться через переменную экземпляра `text` класса `TextEvent`.

```
private function linkListener (e:TextEvent):void {
  var operationName:String = e.text;
  if (operationName == "startGame") {
    startGame( );
  }
}
```

Теперь попытаемся создать гипертекстовую ссылку, которая не просто вызывает операцию на языке ActionScript, но и передает аргументы для этой операции. Код для этой гипертекстовой ссылки представлен ниже. Обратите внимание, что на этот раз задаваемая строка *содержимоеСсылки* содержит название операции (`displayMsg`) и аргумент (`hello world`), которые отделены друг от друга произвольным разделителем (запятой).

```
var t:TextField = new TextField( );
t.htmlText = "<a href='event:displayMsg,hello world'>click here</a>";
t.autoSize = TextFieldAutoSize.LEFT;
addChild(t);
```

Следующий код регистрирует метод `linkListener( )` в предыдущем объекте `TextField` — `t` — для событий `TextEvent.LINK`:

```
t.addEventListener(TextEvent.LINK, linkListener);
```

Наконец, следующий код демонстрирует метод `linkListener( )`, в котором для отделения названия операции (`displayMsg`) от аргумента (`hello world`) применяется метод экземпляра `split( )` класса `String`.

```
private function linkListener (e:TextEvent):void {
  var linkContent:Array    = e.text.split(",");
  var operationName:String = linkContent[0];
  var argument:String      = linkContent[1];

  if (operationName == "displayMsg") {
    displayMsg(argument);
  }
}
```

В отличие от языка JavaScript, обычный код на языке ActionScript нельзя включить непосредственно в атрибут HREF тега <A>.

В листинге 22.17 продемонстрировано использование события `TextEvent.LINK` в контексте гипотетического приложения для обмена сообщениями (чата), в котором пользователю предоставляется возможность запросить личную беседу, щелкнув кнопкой мыши на имени любого пользователя в поле приложения. Код, представленный в листинге 22.17, сильно упрощен, чтобы можно было сконцентрировать внимание на применении события `TextEvent.LINK`; в частности, опущен код, относящийся к реальному получению и отправке сообщений.

Листинг 22.17. Использование события TextEvent.LINK для создания имен пользователей, активизируемых щелчком кнопкой мыши

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;

  // Демонстрирует использование события TextEvent.LINK в примере упрощенного
  // раздела чата с именами пользователей, активизируемыми щелчком кнопкой мыши.
  public class ChatRoom extends Sprite {
    // Текстовое поле, содержащее сообщения чата
    private var messages:TextField;

    public function ChatRoom ( ) {
      // Создаем текстовое поле со ссылками протокола event:
      messages = new TextField( );
      messages.multiline = true;
      messages.autoSize = TextFieldAutoSize.LEFT;
      messages.border     = true;
      messages.background = true;
      // Создаем тестовые сообщения чата с именами пользователей,
      // активизируемыми щелчком кнопкой мыши
      messages.htmlText =
        "<a href='event:privateChat,user1'>Andy</a> says: What's up?<br>"
      + "<a href='event:privateChat,user2'>Mike</a> says: I'm busy...<br>"
      + "<a href='event:privateChat,user1'>Andy</a> says: Ok see you later";
      addChild(messages);
```

```
      // Регистрируем приемник в объекте
      // TextField 'messages' для событий
      // TextEvent.LINK
      messages.addEventListener(TextEvent.LINK, linkListener);
    }

    // Выполняется всякий раз, когда пользователь щелкает кнопкой мыши
    // на ссылке протокола event: в объекте TextField 'messages'
    private function linkListener (e:TextEvent):void {
      // Содержимое переменной e.text – это вся строка, следующая
      // за названием протокола event: в атрибуте href. Например,
      // "privateChat,user1". Эта строка разбивается
      // на название операции ("privateChat")
      // и соответствующий аргумент ("user1").
      var requestedCommand:Array = e.text.split(",");
      var operationName:String   = requestedCommand[0];
      var argument:String        = requestedCommand[1];


      // Если запрашиваемой операцией
      // является запрос личной беседы,
      // то вызываем метод requestPrivateChat( ).
      if (operationName == "privateChat") {
        requestPrivateChat(argument);
      }
    }

    // Отправляет приглашение указанному пользователю на участие
    // в личной беседе
    private function requestPrivateChat (userID:String):void {
      trace("Now requesting private chat with " + userID);
      // Код для работы с сетью не показан...
    }
  }
}
```

События ввода уровня приложения Flash Player

Как мы уже неоднократно видели на протяжении этой главы, подавляющее большинство событий ввода приложения Flash Player возникает в результате взаимодействия пользователя с определенными объектами в списке отображения. Тем не менее Flash Player также поддерживает небольшой набор событий, которые возникают в результате взаимодействия пользователя с самим приложением. Мы будем называть такие «события приложения» событиями ввода уровня приложения Flash Player. Диспетчеризация событий ввода уровня Flash Player выполняется в следующих ситуациях, когда:

- изменяются размеры области отображения приложения Flash Player;
- происходит перемещение указателя мыши над областью отображения Flash Player;
- приложение получает или теряет фокус операционной системы (автономная версия Flash Player получает системный фокус, когда фокус устанавливается на окно приложения; версия Flash Player, реализованная в виде модуля расширения браузера, получает системный фокус, когда пользователь щелкает кнопкой мыши на области отображения приложения или, если такая возможность поддерживается, когда пользователь переходит на внедренный объект приложения с помощью клавиатуры).

В табл. 22.5 перечислены типы событий ввода уровня приложения Flash Player. Для каждого типа события столбец «Тип события» содержит константу класса `Event`, представляющую официальное строковое название типа события. В столбце «Описание» указано конкретное действие пользователя, приводящее к возникновению этого события. Столбец «Получатель» описывает объект, который выступает в роли получателя события при его диспетчеризации. Столбец «Поведение по умолчанию» описывает стандартную реакцию приложения Flash Player на данное событие (у типов событий ввода уровня приложения Flash Player отсутствует поведение по умолчанию). Столбец «Всплывает» содержит информацию о том, имеет ли данное событие фазу всплытия. Столбец «Тип данных объекта, передаваемого в функцию-приемник» определяет тип данных объекта, передаваемого в функцию-приемник, обрабатывающую данное событие. Наконец, столбец «Примечание» содержит важную информацию, касающуюся использования этого события.

Таблица 22.5. События ввода уровня приложения Flash Player

Тип события	Описание	Получатель	Поведение по умолчанию	Всплывает	Тип данных объекта, передаваемого в функцию-приемник	Примечание
Event.ACTIVATE	Приложение Flash Player получило системный фокус	Отображаемый объект, в котором зарегистрирован приемник события	Отсутствует	Да	Event	Возникает даже в тех случаях, когда получатель не находится в списке отображения
Event.DEACTIVATE	Приложение Flash Player потеряло системный фокус	Отображаемый объект, в котором зарегистрирован приемник события	Отсутствует	Да	Event	Возникает даже в тех случаях, когда получатель не находится в списке отображения
Event.RESIZE	Изменились размеры области отображения приложения	Экземпляр класса Stage	Отсутствует	Да	Event	

Продолжение

Таблица 22.5 (продолжение)

Тип события	Описание	Получатель	Поведение по умолчанию	Всплывает	Тип данных объекта, передаваемого в функцию-приемник	Примечание
Event.MOUSE_LEAVE	Указатель мыши переместился за пределы области отображения приложения	Экземпляр класса Stage	Отсутствует	Да	Event	Сопутствующее событие Event.MOUSE_ENTER отсутствует. Используйте событие MouseEvent.MOUSE_MOVE, чтобы определить, когда указатель мыши вновь появится в пределах области отображения Flash Player

Поближе познакомимся с четырьмя событиями, представленными в табл. 22.5.

События Event.ACTIVATE и Event.DEACTIVATE

События `Event.ACTIVATE` и `Event.DEACTIVATE` обычно применяются для разработки приложений, которые самостоятельно активизируются или деактивизируются в ответ на получение или потерю фокуса операционной системы проигрывателем Flash Player. Например, в ответ на потерю фокуса проигрывателем приложение может заглушить все звуки, закрыть открытое меню или приостановить воспроизведение анимации.

В отличие от других событий ввода, с которыми мы познакомились в этой главе, события `Event.ACTIVATE` и `Event.DEACTIVATE` не имеют фазы захвата и фазы всплытия. Вместо этого события `Event.ACTIVATE` и `Event.DEACTIVATE` могут быть обработаны приемниками, зарегистрированными в любом экземпляре любого класса, унаследованного от класса `EventDispatcher` (примечание: не только классов, унаследованных от класса `DisplayObject`). Более того, когда функция-приемник для событий `Event.ACTIVATE` и `Event.DEACTIVATE` зарегистрирована в отображаемом объекте, она вызывается, даже когда данный объект не находится в списке отображения.

В листинге 22.18 демонстрируются основы использования событий `Event.ACTIVATE` и `Event.DEACTIVATE` — когда проигрыватель Flash Player получает системный фокус, приложение из листинга начинает воспроизведение анимации «вращающийся прямоугольник», а когда Flash Player теряет системный фокус, воспроизведение анимации останавливается (методики создания анимации будут рассмотрены в гл. 24).

Листинг 22.18. Реагирование на события Event.ACTIVATE и Event.DEACTIVATE

```
package {
  import flash.display.*;
  import flash.utils.*;
  import flash.events.*;

  public class Spinner extends Sprite {
    private var timer:Timer;
    private var rect:Sprite;

    public function Spinner ( ) {
      // Создаем изображение прямоугольника
      rect = new Sprite( );
      rect.x = 200;
      rect.y = 200;
      rect.graphics.lineStyle(1);
      rect.graphics.beginFill(0x0000FF);
      rect.graphics.drawRect(0, 0, 150, 75);
      addChild(rect);

      // Регистрируем приемник для получения уведомлений,
      // когда приложение Flash Player
      // получает или теряет системный фокус
      addEventListener(Event.ACTIVATE, activateListener);
      addEventListener(Event.DEACTIVATE, deactivateListener);

      // Создаем таймер,
      // который будет использован для анимации
      timer = new Timer(50, 0);
      timer.addEventListener(TimerEvent.TIMER, timerListener);
    }

    // Вращает изображение прямоугольника
    private function timerListener (e:TimerEvent):void {
      rect.rotation += 10;
    }

    // Обрабатывает события Event.ACTIVATE
    private function activateListener (e:Event):void {
      // Начинаем вращение изображения прямоугольника
      timer.start( );
    }

    // Обрабатывает события Event.DEACTIVATE
    private function deactivateListener (e:Event):void {
      // Останавливаем вращение изображения прямоугольника
      timer.stop( );
    }
  }
```

Событие Event.RESIZE

Событие `Event.RESIZE` обычно используется при разработке приложений с «растягивающимся» содержимым, когда размеры элементов интерфейса изменяются автоматически для соответствия доступному пространству в области отображения проигрывателя Flash Player.

Событие `Event.RESIZE` возникает тогда, когда переменной `scaleMode` экземпляра класса `Stage` Flash Player присвоено значение `StageScaleMode.NO_SCALE` и происходит изменение размеров области отображения проигрывателя. Изменение размеров происходит в любом из следующих случаев:

- изменяются размеры окна автономной версии Flash Player либо пользователем, либо в результате вызова функции `fscommand("fullscreen", "true");`
- пользователь изменяет размеры окна браузера, содержащего SWF-файл, внедренный с применением процентных значений для атрибутов `height` или `width` тегов `<OBJECT>` или `<EMBED>`.

Следующий HTML-код демонстрирует, как можно внедрить SWF-файл с использованием процентных значений для атрибута `width` тегов `<OBJECT>` и `<EMBED>`. Обратите внимание, что для одного измерения задан фиксированный размер в пикселах (`height="75"`), а для другого — размер в процентах (`width="100%"`).

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://fpdownload.macromedia.com/get/flashplayer/current/swflash.cab"
  width="100%"
  height="75">
  <PARAM NAME="movie" VALUE="app.swf">
  <EMBED src="app.swf"
    width="100%"
    height="75"
    type="application/x-shockwave-flash"
    pluginspage="http://www.adobe.com/go/getflashplayer">
  </EMBED>
</OBJECT>
```

Приемники для события `Event.RESIZE` должны быть зарегистрированы в экземпляре класса `Stage` Flash Player, как показано в следующем коде. Обратите внимание, что переменной `scaleMode` обязательно должно быть присвоено значение `StageScaleMode.NO_SCALE`.

```
package {
  import flash.display.*;
  import flash.events.*;

  public class ResizeSensor extends Sprite {
    public function ResizeSensor ( ) {
      stage.scaleMode = StageScaleMode.NO_SCALE;
      stage.addEventListener(Event.RESIZE, resizeListener);
    }
```

```
    private function resizeListener (e:Event):void {
      trace("Flash Player was resized");
    }
  }
}
```

Листинг 22.19 расширяет предыдущий код, показывая, как размещать объект `Sprite rect` в правом верхнем углу области отображения Flash Player при каждом изменении размеров окна приложения. Обратите внимание, что приложение из листинга вручную вызывает код, осуществляющий исходное размещение объекта, поскольку при первоначальной загрузке SWF-файла в приложение Flash Player событие `Event.RESIZE` не возникает.

Листинг 22.19. Растягиваемая форма

```
package {
  import flash.display.*;
  import flash.events.*;

  // Помещает объект Sprite rect в правый верхний угол области отображения
  // приложения Flash Player всякий раз при изменении размеров окна
  // приложения Flash Player
  public class StretchyLayout extends Sprite {
    private var rect:Sprite;
    public function StretchyLayout ( ) {
      // Создаем изображение прямоугольника и помещаем его
      // в список отображения
      rect = new Sprite( );
      rect.graphics.lineStyle( );
      rect.graphics.beginFill(0x0000FF);
      rect.graphics.drawRect(0, 0, 150, 75);
      addChild(rect);

      // Предотвращаем масштабирование содержимого
      stage.scaleMode = StageScaleMode.NO_SCALE;
      // Помещаем SWF-файл в левый верхний угол области отображения
      // приложения Flash Player
      stage.align     = StageAlign.TOP_LEFT;
      // Регистрируем приемник для событий Event.RESIZE
      stage.addEventListener(Event.RESIZE, resizeListener);

      // Вручную вызываем код для исходного размещения объекта
      positionRectangle( );
    }

    // Обрабатывает события Event.RESIZE
    private function resizeListener (e:Event):void {
      positionRectangle( );
    }

    // Помещает объект rect в правый верхний угол области отображения
    // приложения Flash Player
    private function positionRectangle ( ):void {
```

```
      rect.x = stage.stageWidth - rect.width;
      rect.y = 0;
    }
  }
}
```

Событие Event.MOUSE_LEAVE

Событие `Event.MOUSE_LEAVE` обычно применяется для отключения или удаления содержимого, взаимодействующего с мышью, когда указатель покидает пределы области отображения проигрывателя Flash Player. Например, в приложении, которое скрывает системный указатель мыши, заменяя его пользовательским изображением указателя мыши (как было показано ранее в листинге 22.5), изображение пользовательского указателя мыши скрывается, когда он перемещается за пределы области отображения приложения Flash Player.

Как и в случае с `Event.RESIZE`, приемники для события `Event.MOUSE_LEAVE` должны быть зарегистрированы в экземпляре класса `Stage` приложения Flash Player. Следующий пример демонстрирует базовый код, необходимый для обработки событий `Event.MOUSE_LEAVE`:

```
package {
  import flash.display.*;
  import flash.events.*;

  public class MouseLeaveSensor extends Sprite {
    public function MouseLeaveSensor ( ) {
      // Регистрируем приемник для событий Event.MOUSE_LEAVE
      stage.addEventListener(Event.MOUSE_LEAVE, mouseLeaveListener);
    }

    // Обрабатывает события Event.MOUSE_LEAVE
    private function mouseLeaveListener (e:Event):void {
      trace("The mouse has left the building.");
    }
  }
}
```

Из программы на экран

На протяжении нескольких предыдущих глав мы рассмотрели множество методик для создания графического содержимого и его изменения в ответ на действия пользователя. В следующей главе мы узнаем, как происходит автоматическое обновление экрана для отображения текущего графического содержимого программы в средах выполнения Flash. Познакомившись с системой обновления экрана среды выполнения, мы перейдем к гл. 24, где будем использовать цикл обновления экрана для создания программной анимации.

Обновления экрана

Говоря по существу, все обновления экрана в языке ActionScript можно разбить на две категории: происходящие через регулярные интервалы времени *(запланированные обновления)* и происходящие сразу после выполнения определенных функций-приемников событий *(постсобытийные обновления)*. Независимо от категории, все обновления экрана являются автоматическими. В ActionScript отсутствуют универсальные способы, позволяющие запросить немедленное обновление экрана. Вместо этого новое графическое содержимое, создаваемое программным путем или вручную в среде разработки Flash, отображается автоматически на этапе запланированного или постсобытийного обновления. В этой главе рассматриваются две разновидности обновлений экрана в языке ActionScript.

Хотя большая часть этой книги посвящена созданию кода с использованием чистого языка ActionScript, а не рассмотрению конкретных сред разработки SWF-файлов, для чтения следующего материала вам потребуются базовые знания временной шкалы и методик создания сценариев временной шкалы в среде разработки Flash. Если вы незнакомы со средой разработки Flash, вам следует прочитать гл. 29 перед тем, как продолжить изучение предложенного здесь материала.

Запланированные обновления экрана

В ActionScript обновления экрана неразрывно связаны с анимационными возможностями среды выполнения Flash. Даже те приложения, которые созданы исключительно с использованием языка ActionScript в приложении Flex Builder 2 или с помощью консольного компилятора mxmlc, подвержены влиянию системы обновления экрана, реализующей анимационные возможности среды Flash.

Система обновления экрана среды выполнения Flash разработана для поддержки модели среды разработки Flash, предназначенной для создания анимированного содержимого с использованием сценариев. В среде разработки Flash анимированное содержимое создается вручную в виде последовательности кадров на временной шкале аналогично кадрам на обычной киноленте. Каждый визуальный кадр может быть связан с блоком кода, называемым *сценарием кадра*. В самых общих чертах, когда среда выполнения воспроизводит анимацию, созданную в среде разработки, она придерживается следующего цикла обновления экрана.

1. Выполнить код текущего кадра.
2. Обновить экран.

3. Перейти к следующему кадру.
4. Повторить цикл.

Предположим, что у нас есть созданная в среде разработки Flash анимация, которая состоит из трех кадров, а с каждым кадром связан сценарий. Обобщенный процесс, в соответствии с которым среда выполнения Flash воспроизводит анимацию, выглядит следующим образом.

1. Выполнить сценарий кадра 1.
2. Отобразить содержимое кадра 1.
3. Выполнить сценарий кадра 2.
4. Отобразить содержимое кадра 2.
5. Выполнить сценарий кадра 3.
6. Отобразить содержимое кадра 3.

На шагах 1, 3 и 5 сценарий каждого кадра может создать новое или изменить существующее графическое содержимое. Таким образом, более точное описание предыдущих шагов процесса воспроизведения анимации выглядело бы следующим образом.

1. Выполнить сценарий кадра 1.
2. Отобразить содержимое кадра 1 и визуальные результаты выполнения сценария кадра 1.
3. Выполнить сценарий кадра 2.
4. Отобразить содержимое кадра 2 и визуальные результаты выполнения сценария кадра 2.
5. Выполнить сценарий кадра 3.
6. Отобразить содержимое кадра 3 и визуальные результаты выполнения сценария кадра 3.

В приведенном описании процесса обратите внимание, что перед тем, как отобразить визуальные результаты выполнения некоторого сценария кадра, среда Flash всегда полностью завершает выполнение данного сценария.

Среда выполнения Flash никогда не прерывает выполнение сценария кадра для того, чтобы обновить экран.

Скорость, с которой выполняются шесть предыдущих шагов, определяется скоростью кадров среды Flash, которая измеряется количеством кадров в секунду. Предположим, что для предыдущей анимации была установлена очень медленная скорость кадров — один кадр в секунду. Кроме того, предположим, что выполнение сценария каждого кадра занимает ровно 100 мс, а визуализация содержимого каждого кадра — ровно 50 мс. Относительно момента начала воспроизведения анимации теоретические времена выполнения шести описанных шагов выглядели бы следующим образом:

```
0мс:    Начало выполнения сценария кадра 1
100мс:  Завершение выполнения сценария кадра 1
1000мс: Начало визуализации содержимого кадра 1 и результатов
        выполнения сценария кадра
```

```
1050мс: Завершение визуализации содержимого кадра 1 и результатов
        выполнения сценария кадра

1051мс: Начало выполнения сценария кадра 2
1151мс: Завершение выполнения сценария кадра 2
2000мс: Начало визуализации содержимого кадра 2 и результатов выполнения
        сценария кадра
2050мс: Завершение визуализации содержимого кадра 2 и результатов
        выполнения сценария кадра

2051мс: Начало выполнения сценария кадра 3
2151мс: Завершение выполнения сценария кадра 3
3000мс: Начало визуализации содержимого кадра 3 и результатов выполнения
        сценария кадра
3050мс: Завершение визуализации содержимого кадра 3 и результатов выполнения
        сценария кадра
```

Обратите внимание, что после завершения выполнения сценария каждого кадра среда Flash не осуществляет немедленное обновление экрана. Вместо этого она выводит результаты выполнения сценария на следующем запланированном этапе визуализации кадра. Следовательно, обновления экрана среды выполнения Flash могут считаться *запланированными обновлениями экрана*, поскольку они происходят в соответствии с установленным графиком, который определяется скоростью кадров.

Таким образом, еще более точное описание шагов предыдущего процесса воспроизведения анимации выглядело бы следующим образом.

1. Выполнить сценарий кадра 1.
2. Дождаться следующего запланированного этапа визуализации кадра.
3. Отобразить содержимое кадра 1 и визуальные результаты выполнения сценария кадра 1.
4. Выполнить сценарий кадра 2.
5. Дождаться следующего запланированного этапа визуализации кадра.
6. Отобразить содержимое кадра 2 и визуальные результаты выполнения сценария кадра 2.
7. Выполнить сценарий кадра 3.
8. Дождаться следующего запланированного этапа визуализации кадра.
9. Отобразить содержимое кадра 3 и визуальные результаты выполнения сценария кадра 3.

Теперь давайте представим, что сценарий кадра 1 регистрирует функцию-приемник `clickListener( )` в экземпляре класса `Stage` для событий `MouseEvent.CLICK`. Всякий раз, когда выполняется функция `clickListener( )`, из точки, где на настоящий момент находится указатель мыши, рисуется красная линия. Рассмотрим код сценария кадра для кадра 1:

```
import flash.events.*;
import flash.display.*;
stage.addEventListener(MouseEvent.CLICK, clickListener);
```

```
function clickListener (e:MouseEvent):void {
  graphics.lineStyle(2, 0xFF0000);
  graphics.lineTo(e.stageX, e.stageY);
}
```

Сразу после завершения выполнения сценария кадра 1 метод `clickListener( )` может получать уведомления о возникновении событий `MouseEvent.CLICK`.

Теперь предположим, что пользователь щелкнул кнопкой мыши в области отображения среды выполнения Flash через 500 мс после начала воспроизведения анимации (то есть в период ожидания, который представлен шагом 2 в предыдущем описании процесса). Метод `clickListener( )` выполняется немедленно, однако визуальные результаты этого не будут отображены до момента наступления следующего запланированного этапа визуализации кадра. На следующем этапе визуальные результаты выполнения метода `clickListener( )` будут отображены вместе с содержимым кадра 1 и результатами выполнения сценария кадра 1.

Таким образом, еще более точное описание шагов предыдущего процесса воспроизведения анимации выглядело бы следующим образом.

1. Выполнить сценарий кадра 1.
2. Дождаться следующего запланированного этапа визуализации кадра. В процессе ожидания, если возникают какие-либо события, выполнять соответствующие приемники событий.
3. Отобразить содержимое кадра 1, визуальные результаты выполнения сценария кадра 1, визуальные результаты выполнения любых приемников событий на шаге 2.
4. Выполнить сценарий кадра 2.
5. Дождаться следующего запланированного этапа визуализации кадра. В процессе ожидания, если возникают какие-либо события, выполнять соответствующие приемники событий.
6. Отобразить содержимое кадра 2, визуальные результаты выполнения сценария кадра 2, визуальные результаты выполнения любых приемников событий на шаге 5.
7. Выполнить сценарий кадра 3.
8. Дождаться следующего запланированного этапа визуализации кадра. В процессе ожидания, если возникают какие-либо события, выполнять соответствующие приемники событий.
9. Отобразить содержимое кадра 3, визуальные результаты выполнения сценария кадра 3, визуальные результаты выполнения любых приемников событий на шаге 8.

Описанные шаги отражают поведение, по умолчанию используемое средой выполнения Flash для обновления экрана. Однако для некоторых типов событий среда Flash может выполнять обновление экрана более быстро. Дополнительную информацию можно получить в разд. «Постсобытийные обновления экрана» далее в этой главе.

Теперь предположим, что содержимое кадра 2 идентично содержимому кадра 1, сценарий кадра 2 не генерирует никаких визуальных результатов и никакие приемники событий не вызываются в промежутке между кадром 1 и кадром 2. В таком случае среда Flash не выполняет визуализацию области отображения. Вместо этого, когда наступает этап визуализации кадра для кадра 2, среда выполнения попросту проверяет, требуется ли обновление экрана. Кадр 2 не содержит визуальных изменений, поэтому повторная визуализация экрана не происходит.

Таким образом, еще более точное описание шагов предыдущего процесса воспроизведения анимации выглядело бы следующим образом.

1. Выполнить сценарий кадра 1.
2. Дождаться следующего запланированного этапа визуализации кадра. В процессе ожидания, если возникают какие-либо события, выполнять все зарегистрированные приемники событий.
3. На этапе визуализации кадра проверить, требуется ли обновление экрана. Обновление экрана требуется в том случае, когда справедливо любое из следующих условий:
   - кадр 1 включает изменения содержимого экземпляра класса `Stage`, созданные вручную в среде разработки Flash;
   - код сценария кадра 1 создал новое или изменил существующее графическое содержимое;
   - код функции-приемника, которая была выполнена на шаге 2, создал новое или изменил существующее графическое содержимое.
4. Если необходимо, обновить экран, чтобы отразить все изменения, произошедшие на шаге 3.
5. Повторить шаги 1–4 для кадров 2 и 3.

Для вашего сведения, в оставшейся части этой главы и в следующей главе мы будем называть проверку необходимости обновления экрана, происходящую на шаге 3, *проверкой запланированного обновления экрана*. Всякий раз, когда среда выполнения Flash проводит проверку запланированного обновления экрана, она осуществляет диспетчеризацию события `Event.ENTER_FRAME` (даже в тех случаях, когда обновления экрана на самом деле не происходит). Реагируя на событие `Event.ENTER_FRAME`, объекты могут выполнять повторяющиеся задачи, синхронизированные с операцией обновления экрана. В гл. 24 будет рассказано, как использовать событие `Event.ENTER_FRAME` для создания анимированного содержимого полностью программным путем.

Вы готовы рассмотреть последний гипотетический сценарий? Предположим, что мы удаляем кадры 2 и 3 из нашей анимации, оставляя только кадр 1. Как и раньше, сценарий кадра 1 определяет приемник события `MouseEvent.CLICK` — `clickListener( )`. Как только содержимое кадра 1 и результаты выполнения сценария кадра будут визуализированы (шаг 4 в предыдущем описании процесса), воспроизведение анимации завершится. Тем не менее для продолжения обработки событий цикл обновления экрана среды выполнения Flash должен оставаться активным. Таким образом, для SWF-файла, содержащего всего один кадр, цикл

обновления экрана выглядит следующим образом (описанные шаги также применимы к многокадровому SWF-файлу, воспроизведение которого было просто приостановлено на кадре 1).

1. Выполнить сценарий кадра 1.
2. Дождаться следующего запланированного этапа визуализации кадра. В процессе ожидания, если возникают какие-либо события, выполнять все зарегистрированные приемники событий.
3. На этапе визуализации кадра проверить, требуется ли обновление экрана. Обновление экрана требуется в том случае, когда справедливо любое из следующих условий:
   - кадр 1 включает изменения содержимого экземпляра класса `Stage`, созданные вручную в среде разработки Flash;
   - код сценария кадра 1 создал новое или изменил существующее графическое содержимое;
   - код функции-приемника, которая была выполнена на шаге 2, создал новое или изменил существующее графическое содержимое.
4. Если необходимо, обновить экран, чтобы отразить все изменения, произошедшие на шаге 3.
5. Дождаться следующего запланированного этапа визуализации кадра. В процессе ожидания, если возникают какие-либо события, выполнять все зарегистрированные приемники событий.
6. На этапе визуализации кадра проверить, требуется ли обновление экрана. Обновление экрана требуется в том случае, когда код функции-приемника, выполненной на шаге 5, создал новое или изменил существующее графическое содержимое.
7. Если необходимо, обновить экран, чтобы отразить все изменения, произошедшие на шаге 6.
8. Повторить шаги 5–7.

Шаги 5–8 из предыдущего описания процесса постоянно повторяются до тех пор, пока SWF-файл выполняется в среде Flash, привязывая тем самым последующее выполнение кода к циклу обновления экрана, который зависит от скорости кадров.

В гл. 20 мы узнали, что, когда пустая среда выполнения Flash открывает новый SWF-файл, она находит основной класс этого SWF-файла, создает его экземпляр и добавляет созданный экземпляр в список отображения в качестве первого ребенка экземпляра класса `Stage`. Для программ, разработанных исключительно с использованием языка ActionScript, обновление экрана происходит сразу после завершения выполнения метода-конструктора основного класса. Все последующие обновления экрана происходят в соответствии с циклом обновления экрана, описанном на шагах 5–8 предыдущего списка и зависящем от скорости кадров.

Например, рассмотрим следующую чрезвычайно простую программу рисования, которая увеличивает частоту обновления экрана среды выполнения Flash путем установки скорости кадров, равной одному кадру в секунду:

```
package {
  import flash.display.*;
  import flash.events.*;

  public class SimpleScribble extends Sprite {
    public function SimpleScribble ( ) {
      stage.frameRate = 1;
      graphics.moveTo(stage.mouseX, stage.mouseY);
      stage.addEventListener(MouseEvent.MOUSE_MOVE, mouseMoveListener);
    }

    private function mouseMoveListener (e:MouseEvent):void {
      graphics.lineStyle(2, 0xFF0000);
      graphics.lineTo(e.stageX, e.stageY);
    }
  }
}
```

Метод-конструктор класса `SimpleSribble` не создает никакого графического содержимого, но регистрирует приемник `mouseMoveListener( )` для события `MouseEvent.MOUSE_MOVE`. При перемещении мыши метод `mouseMoveListener( )` рисует линию в точку, где на настоящий момент находится указатель мыши. Однако нарисованная линия не отображается на экране до следующего этапа обновления экрана, которое происходит один раз в секунду. Следовательно, каждую секунду среда выполнения Flash обновляет экран набором линий, отражающих траекторию перемещения указателя мыши в области отображения с момента последнего обновления экрана. Для более «плавного» рисования мы могли бы увеличить скорость кадров до 30 в секунду или инициировать немедленные обновления экрана, используя методики, описываемые далее, в разд. «Постсобытийные обновления экрана».

Подведем промежуточные итоги, перечислив рассмотренные ключевые моменты.

- Система обновления экрана языка ActionScript является полностью автоматической.
- Для приложений, разработанных исключительно с использованием ActionScript, скорость кадров можно представить как количество автоматических проверок необходимости обновления экрана, проводимых средой выполнения Flash за одну секунду. Например, если скорость кадров среды выполнения Flash равна 1, все визуальные изменения, внесенные программой, будут автоматически отображаться один раз в секунду; если скорость кадров равна 10, визуальные изменения будут автоматически отображаться 10 раз в секунду (каждые 100 мс).
- Если скорость кадров очень низкая (скажем, 1–10 кадров в секунду), то между выполнением кода, генерирующего визуальное содержимое, и отображением этого содержимого на экране могут происходить заметные задержки.
- Всякий раз, когда среда Flash выполняет проверку запланированного обновления экрана, она осуществляет диспетчеризацию события `Event.ENTER_FRAME`.
- Приложение Flash Player никогда не прерывает выполнение блока кода для того, чтобы обновить экран.

Последний из перечисленных моментов является чрезвычайно важным, поэтому рассмотрим его более подробно.

Никаких обновлений экрана внутри блоков кода

Повторим еще раз, что среда Flash никогда не прерывает выполнение блока кода для того, чтобы обновить экран. До того как произойдет запланированное обновление экрана, все функции в стеке вызовов и весь код в текущем кадре должны завершить свое выполнение. Подобным образом, перед тем как произойдет постсобытийное обновление экрана, приемник события, внутри которого был вызван метод `updateAfterEvent( )`, должен завершить свое выполнение.

На самом деле, даже если обновление экрана запланировано на данное время, это обновление будет отложено до тех пор, пока не завершится выполнение текущего кода. Обновления экрана и выполнение кода являются взаимно исключающими задачами для среды Flash; они всегда происходят последовательно, но ни в коем случае не одновременно.

В качестве золотого правила запомните, что в ActionScript обновление экрана не может произойти между двумя строками кода. Например, следующая функция `displayMsg( )` создает объект `TextField` и дважды устанавливает его горизонтальное положение: сначала 50, а затем 100:

```
public function displayMsg ( ):void {
  var t:TextField = new TextField( );
  t.text = "Are we having fun yet?";
  t.autoSize = TextFieldAutoSize.LEFT;
  addChild(t);
  t.x = 50;
  t.x = 100;
}
```

При выполнении функции `displayMsg( )` экран никогда не будет и не может быть обновлен между двумя последними строками в этой функции. В результате объект `TextField` никогда не появится на экране в горизонтальной позиции 50. Вместо этого выполнение функции полностью завершится перед визуализацией экрана и объект `TextField` будет отображен в горизонтальной позиции 100. Хотя переменной `x` на самом деле кратковременно присваивается значение `50`, визуальный результат этого изменения никогда не будет отображен на экране.

В некоторых случаях выполнение кода может задерживать обновления экрана на много секунд, вплоть до максимального значения, определяемого параметром компилятора `max-execution-time`, которому по умолчанию присваивается значение `15`. Любой сценарий, выполнение которого не завершается в течение времени, определяемого параметром `max-execution-time`, генерирует исключение `ScriptTimeoutError`. Информацию по обработке этого исключения можно найти в описании исключения `flash.errors.ScriptTimeoutError` в справочнике по языку ActionScript корпорации Adobe.

Чтобы избежать появления исключений `ScriptTimeoutError`, весь код должен быть разработан таким образом, чтобы его выполнение завершалось в течение ин-

тервала, определяемого параметром компилятора `max-execution-time`. Для выполнения задачи, которой требуется больше времени, чем позволяет разрешенный лимит времени, разбейте ее на части, которые могут быть выполнены в течение времени, определяемого параметром `max-execution-time`, а затем используйте класс `Timer`, чтобы организовать выполнение этих частей кода.

Установка скорости кадров

Установить скорость кадров приложения Flash Player можно одним из следующих способов:

- используя аргумент `default-frame-rate` компилятора mxmlc;
- в окне Document Properties (Свойства документа) среды разработки Flash;
- используя переменную экземпляра `frameRate` класса `Stage` внутри выполняемого SWF-файла.

Первый SWF-файл, загруженный в среду Flash, устанавливает начальную скорость кадров для всех SWF-файлов, которые будут загружены в дальнейшем.

Независимо от способа установки скорости кадров, она будет использована всеми загруженными в дальнейшем SWF-файлами (переопределяя их собственную заданную скорость кадров). Однако после загрузки первого SWF-файла установленная скорость кадров среды выполнения Flash может быть изменена переменной экземпляра `frameRate` класса `Stage`, принимающей значения в диапазоне от `0.01` (один кадр каждые 100 с) до `1000` (1000 кадров в секунду). Например, следующий код устанавливает скорость кадров, равную 150 кадрам в секунду:

```
package {
  import flash.display.*;
  public class SetFrameRate extends Sprite {
    public function SetFrameRate ( ) {
      stage.frameRate = 150;
    }
  }
}
```

Хотя скорость кадров может быть установлена через любой объект, имеющий доступ к экземпляру класса `Stage`, отображаемые объекты не могут выполняться с различными скоростями кадров. Визуализация всех объектов в списке отображения осуществляется одновременно, с учетом единой скорости кадров среды Flash.

До появления языка ActionScript 3.0 скорость кадров среды выполнения нельзя было изменять программным путем.

Назначенная скорость кадров в сравнении с реальной скоростью

Хотя среда Flash выполняет запланированные обновления экрана в соответствии со скоростью кадров, количество обновлений экрана в секунду, которое может быть

достигнуто на самом деле *(реальная скорость кадров)*, зачастую оказывается меньше, чем скорость кадров, указанная программистом *(назначенная скорость кадров)*. Реальная скорость кадров существенно зависит от таких факторов, как производительность компьютера, доступные системные ресурсы, физическая частота обновления устройства отображения и сложность содержимого, исполняемого в среде Flash. Следовательно, назначенную скорость кадров необходимо считать нижним пределом скорости. Реальная скорость никогда не будет меньше назначенной скорости, однако в некоторых условиях она не сможет достичь назначенной скорости кадров.

Приложение Flash Player не всегда сможет достигать назначенной скорости кадров.

Чтобы определить текущую назначенную скорость кадров, мы проверяем значение переменной экземпляра `frameRate` класса `Stage`. Например, следующий класс отображает назначенную скорость кадров в объекте `TextField`:

```
package {
  import flash.display.*;
  import flash.text.*;

  public class ShowFrameRate extends Sprite {
    public function ShowFrameRate ( ) {
      var t:TextField = new TextField( );
      t.autoSize = TextFieldAutoSize.LEFT;
      t.text = stage.frameRate.toString( );
      addChild(t);
    }
  }
}
```

Для того чтобы определить *реальную* скорость кадров, мы используем событие `Event.ENTER_FRAME`, применяемое для измерения времени, прошедшего между проверками запланированных обновлений экрана среды выполнения Flash. Эта методика продемонстрирована в листинге 23.1. Более подробно с событием `Event.ENTER_FRAME` мы познакомимся в гл. 24.

Листинг 23.1. Измерение реальной скорости кадров

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.utils.*;
  import flash.text.*;

  public class FrameRateMeter extends Sprite {
    private var lastFrameTime:Number;
    private var output:TextField;

    public function FrameRateMeter( ) {
      output = new TextField( );
      output.autoSize = TextFieldAutoSize.LEFT;
      output.border = true;
```

```
      output.background = true;
      output.selectable = false;
      addChild(output);

      addEventListener(Event.ENTER_FRAME, enterFrameListener);
    }

    private function enterFrameListener (e:Event):void {
      var now:Number = getTimer( );
      var elapsed:Number = now - lastFrameTime;
      var framesPerSecond:Number = Math.round(1000/elapsed);
      output.text = "Time since last frame: " + elapsed
            + "\nExtrapolated actual frame rate: " + framesPerSecond
            + "\nDesignated frame rate: " + stage.frameRate;
      lastFrameTime = now;
    }
  }
}
```

Реальная скорость кадров в отладочной версии среды выполнения Flash зачастую оказывается гораздо меньше реальной скорости кадров в рабочей версии.

Постсобытийные обновления экрана

Из предыдущего раздела мы узнали, что запланированные обновления экрана происходят автоматически через интервалы, определяемые скоростью кадров. Мы также узнали, что визуальные изменения, вносимые приемниками событий, не отображаются до следующего запланированного этапа обновления экрана. При использовании стандартной скорости кадров, равной 24 кадрам в секунду, задержка между выполнением приемника события и отображением его визуальных результатов обычно незаметна. Тем не менее для визуальных изменений, возникающих в результате взаимодействия пользователя с мышью или клавиатурой, даже небольшие задержки могут привести к тому, что приложение будет выглядеть дрожащим или замедленным. В связи с этим язык ActionScript дает каждой функции-приемнику событий мыши и клавиатуры специальную возможность инициировать *постсобытийное обновление экрана*. Это обновление, которое происходит непосредственно после диспетчеризации события, перед следующим запланированным обновлением.

Для того чтобы запросить постсобытийное обновление экрана в ответ на событие мыши, мы вызываем метод `MouseEvent.updateAfterEvent( )` над объектом `MouseEvent`, передаваемым во все функции-приемники событий мыши. Например, следующий код вызывает постсобытийное обновление экрана в ответ на событие `MouseEvent.MOUSE_MOVE`:

```
private function mouseMoveListener (e:MouseEvent):void {
  e.updateAfterEvent( );  // Вызываем обновление
}
```

Чтобы запросить постсобытийное обновление экрана в ответ на событие клавиатуры, мы вызываем метод `KeyboardEvent.updateAfterEvent( )` над объектом `KeyboardEvent`, передаваемым во все функции-приемники событий клавиатуры. Например, следующий код вызывает постсобытийное обновление экрана в ответ на событие `KeyboardEvent.KEY_DOWN`:

```
private function keyDownListener (e:KeyboardEvent):void {
  e.updateAfterEvent( );  // Вызываем обновление
}
```

В обоих случаях вызов метода `updateAfterEvent( )` заставляет среду выполнения Flash обновить экран сразу после диспетчеризации события, перед следующим запланированным обновлением экрана. Тем не менее, хотя постсобытийное обновление экрана осуществляется перед следующим запланированным обновлением экрана, оно не произойдет до тех пор, пока все приемники событий, вызванные в процессе диспетчеризации события, не завершат свое выполнение.

Как и в случае с запланированными обновлениями экрана, среда Flash никогда не прерывает выполнение блока кода, чтобы осуществить постсобытийное обновление экрана.

Пример использования метода `updateAfterEvent( )` в реальном сценарии продемонстрирован в классе пользовательского указателя мыши `CustomMousePointer`, который был представлен в подразд. «Определение позиции указателя мыши» разд. «События мыши» гл. 22. Класс `CustomMousePointer` рисует синий треугольник в объекте `Sprite`, представляющем указатель, и использует приемник события `MouseEvent.MOUSE_MOVE`, чтобы реализовать перемещение этого объекта `Sprite` за мышью. Метод `updateAfterEvent( )` применяется внутри метода `mouseMoveListener( )` для вызова постсобытийного обновления экрана, чтобы обеспечить плавное перемещение указателя, не зависящее от скорости кадров.

Рассмотрим код метода `mouseMoveListener( )` из класса `CustomMousePointer`. Обратите внимание на вызов метода `updateAfterEvent( )`, выделенный полужирным шрифтом:

```
private function mouseMoveListener (e:MouseEvent):void {
  // При перемещении мыши обновляем позицию пользовательского указателя
  // мыши, чтобы она соответствовала позиции системного указателя
  var pointInParent:Point = parent.globalToLocal(new Point(e.stageX,
                                                           e.stageY));
  x = pointInParent.x;
  y = pointInParent.y;

  // Запрашиваем постсобытийное обновление экрана, чтобы анимация
  // указателя была максимально плавной
  e.updateAfterEvent( );

  // Убеждаемся, что пользовательский указатель мыши отображается на экране
  // (он может быть скрыт, поскольку системный указатель мог покидать
  // пределы области отображения приложения Flash Player).
  if (!visible) {
```

```
    visible = true;
  }
}
```

Стоит отметить, что, когда среда Flash обновляет экран в ответ на вызов метода `updateAfterEvent( )`, она отображает не только те изменения, которые были внесены функцией-приемником события, запросившей данное обновление, но и все визуальные изменения, произошедшие с момента последнего обновления экрана.

Постсобытийные обновления для событий таймера

Для того чтобы разрешить обновление экрана сразу после истечения некоторого произвольного интервала времени, язык ActionScript предоставляет метод `TimerEvent.updateAfterEvent( )`, который вызывает постсобытийное обновление экрана после возникновения события `TimerEvent.TIMER`.

Указанный метод точно так же используется внутри функций-приемников события `TimerEvent.TIMER`, как и методы `MouseEvent.updateAfterEvent( )` и `KeyboardEvent.updateAfterEvent( )` внутри функций-приемников событий мыши и клавиатуры.

Для демонстрации использования метода `TimerEvent.updateAfterEvent( )` создадим расширенный пример, который генерирует событие `TimerEvent.TIMER` в десять раз чаще, чем скорость кадров среды выполнения Flash. Мы начнем с установки скорости, равной одному кадру в секунду:

```
stage.frameRate = 1;
```

Далее мы создаем объект `Timer`, который осуществляет диспетчеризацию события `TimerEvent.TIMER` каждые 100 мс (10 раз в секунду):

```
var timer:Timer = new Timer(100, 0);
```

Затем мы регистрируем функцию-приемник `timerListener( )` в объекте `timer` для событий `TimerEvent.TIMER`, как показано в следующем коде:

```
timer.addEventListener(TimerEvent.TIMER, timerListener);
```

После этого мы запускаем таймер:

```
timer.start( );
```

Теперь внутри функции `timerListener( )` рисуем прямоугольник и помещаем его в случайное место на экране. Для того чтобы гарантировать, что прямоугольник появится на экране сразу после завершения процесса диспетчеризации события `TimerEvent.TIMER` (а не на следующем запланированном этапе обновления экрана), мы используем метод `TimerEvent.updateAfterEvent( )` для запроса постсобытийного обновления экрана. Рассмотрим получившийся код для метода `timerListener( )`:

```
private function timerListener (e:TimerEvent):void {
  // Создаем прямоугольник
  var rect:Sprite = new Sprite( );
  rect.graphics.lineStyle(1);
  rect.graphics.beginFill(0x0000FF);
  rect.graphics.drawRect(0, 0, 150, 75);
```

```
    rect.x = Math.floor(Math.random( )*stage.stageWidth);
    rect.y = Math.floor(Math.random( )*stage.stageHeight);

    // Добавляем прямоугольник на экран
    addChild(rect);

    // Запрашиваем постсобытийное обновление экрана
    e.updateAfterEvent( );
}
```

В результате такого вызова метода `TimerEvent.updateAfterEvent( )` отображение визуальных изменений, вносимых внутри метода `timerListener( )`, происходит приблизительно каждые 100 мс, а не раз в секунду.

Для информации следующий код демонстрирует наш предыдущий сценарий с таймером в контексте класса `RandomRectangles`:

```
package {
  import flash.display.*;
  import flash events.*;
  import flash.utils.*;

  public class RandomRectangles extends Sprite {
    public function RandomRectangles ( ) {
      stage.frameRate = 1;
      var timer:Timer = new Timer(100, 0);
      timer.start( );
      timer.addEventListener(TimerEvent.TIMER, timerListener);
    }

    private function timerListener (e:TimerEvent):void {
      var rect:Sprite = new Sprite( );
      rect.graphics.lineStyle(1);
      rect.graphics.beginFill(0x0000FF);
      rect.graphics.drawRect(0, 0, 150, 75);
      rect.x = Math.floor(Math.random( )*stage.stageWidth);
      rect.y = Math.floor(Math.random( )*stage.stageHeight);

      addChild(rect);

      e.updateAfterEvent( )
    }
  }
}
```

В гл. 24 мы продолжим наше изучение класса `Timer`, применяя его для создания движения и других форм анимации.

Метод setInterval( ) языка ActionScript 2.0 может также использовать метод updateAfterEvent( ) для вызова постсобытийного обновления экрана, однако вместо метода setInterval( ) предпочтительнее использовать класс flash.utils.Timer, поскольку он предоставляет возможность запускать и останавливать события таймера, а также уведомлять о таймерных событиях сразу несколько приемников. Старайтесь избегать использования метода setInterval( ) в языке ActionScript 3.0.

Автоматические постсобытийные обновления экрана

В приложении Flash Player 9 определенные «кнопочные» взаимодействия с объектами любого класса, наследуемого от класса `Sprite`, приводят к автоматическому постсобытийному обновлению экрана (аналогично тому, как программист вызывает метод `updateAfterEvent( )`). В частности, автоматическое постсобытийное обновление экрана будут вызывать следующие взаимодействия:

- перемещение указателя мыши над экземпляром класса, который наследуется от класса `Sprite`, или за пределы этого экземпляра;
- нажатие или отпускание основной кнопки мыши, когда указатель мыши находится над экземпляром класса, наследуемого от `Sprite`;
- использование клавиши Пробел или Enter для активизации экземпляра класса, наследуемого от класса `Sprite`.

Существует небольшая вероятность, что в будущих версиях приложения Flash Player такое особое поведение, относящееся к автоматическому обновлению экрана, будет применяться только к объектам SimpleButton. В связи с этим вы, вероятно, не захотите полагаться на него в своем коде.

Область перерисовки

Как уже известно из разд. «Запланированные обновления экрана», среда выполнения Flash обновляет экран только в тех случаях, когда это необходимо (то есть когда графическое содержимое было изменено или добавлено). Точнее говоря, когда среда выполнения обновляет экран, она визуализирует только те области, которые изменились с момента последнего обновления. Например, представим анимацию, состоящую из двух кадров, первый из которых содержит круг, а второй — тот же круг, но вместе с треугольником. Когда среда Flash визуализирует второй кадр, она перерисовывает прямоугольную область, содержащую треугольник, но не перерисовывает круг. Прямоугольная область, включающая все содержимое, которое было изменено, называется *областью перерисовки* (в программировании графики данная область иногда называется измененным прямоугольником).

Чтобы отобразить область перерисовки в отладочных версиях среды выполнения, можно щелкнуть правой кнопкой мыши в окне приложения Flash Player и в контекстном меню выбрать пункт Show Redraw Regions (Показать области перерисовки).

Оптимизация с использованием события Event.RENDER

Событие `Event.RENDER` — это особый тип события обновления экрана, используемый в сложных ситуациях, когда графическая производительность имеет первостепенное значение. Его основное назначение заключается в предоставлении

программисту возможности отложить выполнение всех пользовательских процедур рисования точно до того момента, когда произойдет визуализация экрана, позволяя тем самым избежать повторного выполнения процедур рисования. В отличие от остальных внутренних событий среды разработки Flash, событие `Event.RENDER` должно быть запрошено программистом вручную. Среда Flash осуществляет диспетчеризацию события `Event.RENDER`, когда выполняются два следующих условия:

- среда выполнения Flash собирается проверить необходимость обновления экрана (либо при прохождении какого-либо кадра, либо в результате вызова метода `updateAfterEvent( )`);
- программист вызвал метод `stage.invalidate( )` (с его помощью программист может попросить среду выполнения Flash осуществить диспетчеризацию события `Event.RENDER` в следующий раз, когда произойдет проверка обновления экрана).

Рассмотрим пример, который демонстрирует, как событие `Event.RENDER` может быть использовано для улучшения производительности. Предположим, что мы создаем класс `Ellipse`, который представляет фигуру эллипса на экране. Для простоты предположим, что эллипс всегда заполняется белым цветом и имеет контур черного цвета толщиной 1 пиксел. Наш класс `Ellipse` должен делать следующее:

- управлять концептуальными данными эллипса (то есть хранить ширину и высоту эллипса);
- рисовать эллипс на экране, используя данные концептуального эллипса, и перерисовывать эллипс на экране при изменении данных концептуального эллипса.

В соответствии с описанными назначениями, в листинге 23.2 представлен вариант реализации класса `Ellipse`, который приемлем в том случае, когда производительность не имеет решающего значения:

Листинг 23.2. Простейший класс Ellipse

```
package {
  import flash.display.Shape;

  public class Ellipse extends Shape {
    private var w:Number;
    private var h:Number;

    public function Ellipse (width:Number, height:Number) {
      w = width;
      h = height;
      draw( );
    }

    public function setWidth (newWidth:Number):void {
      w = newWidth;
      draw( );
    }
    public function getWidth ( ):Number {
```

```
      return w;
    }

    public function setHeight (newHeight:Number):void {
      h = newHeight;
      draw( );
    }

    public function getHeight ( ):Number {
      return h;
    }

    private function draw ( ):void {
      graphics.lineStyle(1);
      graphics.beginFill(0xFFFFFF, 1);
      graphics.drawEllipse(0, 0, w, h);
    }
  }
}
```

Обратите внимание, что в классе `Ellipse` есть три места, где происходит изменение данных концептуального эллипса: метод `setWidth( )`, метод `setHeight( )` и метод-конструктор класса `Ellipse`. Чтобы обеспечить соответствие между концептуальным эллипсом и эллипсом, отображаемым на экране, мы должны убедиться, что перерисовка эллипса, отображаемого на экране, осуществляется в каждом из трех перечисленных мест. Код, представленный в листинге 23.2, для удовлетворения этого требования использует метод решения «в лоб»; он просто вызывает метод `draw( )` всякий раз, когда выполняются методы `getWidth( )`, `getHeight( )` или метод-конструктор класса `Ellipse`. Разумеется, если внутри одного цикла обновления экрана эти функции вызываются несколько раз, дублирующие вызовы метода `draw( )` являются избыточными. Это демонстрирует следующий код:

```
var e:Ellipse = new Ellipse (100, 200);  // метод draw( ) вызывается здесь
e.setWidth(25);   // метод draw( ) вызывается здесь снова
e.setHeight(50);  // метод draw( ) вызывается здесь снова
```

При выполнении трех предыдущих строк кода метод `draw( )` вызывается три раза, однако на экране будут отображены результаты лишь последнего вызова. Первые два вызова являются избыточными и нерациональными. В простейшем приложении такая избыточность незаметна и, следовательно, может оказаться допустимой. Тем не менее в сложных приложениях подобная избыточность может приводить к тому, что процедуры рисования будут бесполезно выполняться сотни или тысячи раз, представляя собой потенциальный источник серьезных проблем, связанных с производительностью.

Чтобы избавиться от избыточности в классе `Ellipse`, мы должны изменить подход, применяемый для рисования фигуры. Вместо того чтобы вызывать метод `draw( )` всякий раз, когда изменяются данные концептуального эллипса, мы будем откладывать вызов этого метода до момента обновления экрана. Эта новая стратегия приведет к усложнению кода класса `Ellipse`, но при этом улучшит его производительность.

Первый шаг в реализации новой стратегии «один вызов метода `draw( )`» заключается в удалении вызова метода `draw( )` из методов `setWidht( )` и `setHeight( )`, а также вызова метода-конструктора класса `Ellipse`. Вместо непосредственного вызова метода `draw( )` эти функции будут вызывать метод `stage.invalidate( )`, который приказывает среде выполнения Flash осуществить диспетчеризацию события `Event.RENDER` при очередной проверке необходимости обновления экрана. Затем из функции-приемника события `Event.RENDER` мы вызовем метод `draw( )`. В листинге 23.3 представлен измененный класс `Ellipse` — отличия от кода из листинга 23.2 выделены полужирным шрифтом. Стоит отметить, что метод `draw( )` не должен вызываться, когда объект `Ellipse` не находится в списке отображения, поэтому вызов метода `stage.invalidate( )` происходит только в том случае, когда объект `Ellipse` находится в списке отображения. Чтобы определить, находится ли объект `Ellipse` в списке отображения, мы проверяем значение унаследованной переменной экземпляра `stage` этого объекта. Когда значение переменной `stage` равно `null`, объект `Ellipse` не находится в списке отображения.

Объект, запрашивающий уведомление о возникновении события Event.RENDER, получит это уведомление даже в том случае, если он не находится в списке отображения.

Обратите внимание, что на данном промежуточном этапе нашей разработки класс `Ellipse` не является полнофункциональным, поскольку он не регистрирует приемники для событий `Event.RENDER`. Эту проблему мы решим в ближайшее время.

Листинг 23.3. Измененный класс Ellipse, часть 1

```
package {
  import flash.display.Shape;

  public class EllipseInterim extends Shape {
    private var w:Number;
    private var h:Number;

    public function EllipseInterim (width:Number, height:Number) {
      w = width;
      h = height;

      // Если этот объект находится в списке отображения...
      if (stage != null) {
        // ...запрашиваем диспетчеризацию события Event.RENDER
        stage.invalidate( );
      }
    }

    public function setWidth (newWidth:Number):void {
      w = newWidth;

      if (stage != null) {
        stage.invalidate( );
      }
    }
```

```
    public function getWidth ( ):Number {
      return w;
    }

    public function setHeight (newHeight:Number):void {
      h = newHeight;

      if (stage != null) {
        stage.invalidate( );
      }
    }

    public function getHeight ( ):Number {
      return h;
    }

    // Приемник события вызывается перед обновлением
    // экрана, если был вызван метод
    // stage.invalidate( )
    private function renderListener (e:Event):void {
      draw( );
    }

    private function draw ( ):void {
      graphics.clear( );
      graphics.lineStyle(1);
      graphics.beginFill(0xFFFFFF, 1);
      graphics.drawEllipse(0, 0, w, h);
    }
  }
}
```

Чтобы метод `renderListener( )` выполнялся всякий раз, когда среда Flash осуществляет диспетчеризацию события `Event.RENDER`, мы должны зарегистрировать метод `renderListener( )` в экземпляре класса `Stage` для событий `Event.RENDER`. Однако когда объект `Ellipse` не находится в списке отображения, его переменной экземпляра `stage` присвоено значение `null` и, следовательно, она не может применяться для регистрации события. Для того чтобы обойти эту проблему, мы определим в классе `Ellipse` две функции-приемника событий — `addedToStageListener( )` и `removedFomStageListener( )`, которые получают уведомления о возникновении пользовательских событий `StageDetector.ADDED_TO_STAGE` и `StageDetector.REMOVED_FROM_STAGE`. Диспетчеризация первого события происходит, когда объект добавляется в список отображения, и при получении этого события класс `Ellipse` будет регистрировать метод `renderListener( )` для события `Event.RENDER`. Диспетчеризация второго события происходит в том случае, когда объект удаляется из списка отображения, и при получении этого события класс `Ellipse` будет отменять регистрацию метода `renderListener( )` для событий `Event.RENDER`.

В листинге 23.4 продемонстрирован измененный класс `Ellipse` (изменения, как и раньше, выделены полужирным шрифтом). Обратите внимание, что метод

addedToStageListener( ) вызывает метод stage.invalidate( ), гарантируя, что любые изменения, внесенные в объект Ellipse за то время, пока он не находился в списке отображения, будут визуализированы при добавлении этого объекта в список отображения.

В листинге 23.4 применяются пользовательские события StageDetector.ADDED_TO_STAGE и StageDetector.REMOVED_FROM_STAGE, диспетчеризация которых осуществляется пользовательским классом StageDetector. Детальное рассмотрение данного класса можно найти в подразд. «События ADDED_TO_STAGE и REMOVED_FROM_STAGE» разд. «События контейнеров» гл. 20.

Листинг 23.4. Измененный класс Ellipse, часть 2

```
package {
  import flash.display.Shape;
  import flash.events.*;

  public class EllipseInterim extends Shape {
    private var w:Number;
    private var h:Number;

    public function EllipseInterim (width:Number, height:Number) {
      // Регистрируем приемники для получения уведомлений,
      // когда данный объект добавляется в список
      // отображения или удаляется из него
      var stageDetector:StageDetector = new StageDetector(this);
      stageDetector.addEventListener(StageDetector.ADDED_TO_STAGE,
                                     addedToStageListener);
      stageDetector.addEventListener(StageDetector.REMOVED_FROM_STAGE,
                                     removedFromStageListener);

      w = width;
      h = height;
      if (stage != null) {
        stage.invalidate( );
      }
    }

    public function setWidth (newWidth:Number):void {
      w = newWidth;
      if (stage != null) {
        stage.invalidate( );
      }
    }

    public function getWidth ( ):Number {
      return w;
    }

    public function setHeight (newHeight:Number):void {
      h = newHeight;
```

```
      if (stage != null) {
        stage.invalidate( );
      }
    }

    public function getHeight ( ):Number {
      return h;
    }

    // Приемник события вызывается при добавлении этой фигуры
    // в список отображения
    private function addedToStageListener (e:Event):void {
      // Регистрируем приемник для получения уведомлений
      // об обновлениях экрана
      stage.addEventListener(Event.RENDER, renderListener);

      // Гарантируем, что любые изменения, внесенные в данный объект
      // за то время, пока он не отображался на экране, будут
      // визуализированы при его добавлении в список отображения.
      stage.invalidate( );
    }

    // Приемник события вызывается при удалении этой фигуры из списка
    // отображения
    private function removedFromStageListener (e:Event):void {
      // Нет необходимости в получении событий
      // обновления экрана, когда данный объект
      // не находится в списке отображения
      stage.addEventListener(Event.RENDER, renderListener);
    }

    private function renderListener (e:Event):void {
      draw( );
    }

    private function draw ( ):void {
      graphics.clear( );
      graphics.lineStyle(1);
      graphics.beginFill(0xFFFFFF, 1);
      graphics.drawEllipse(0, 0, w, h);
    }
  }
}
```

Класс `Ellipse`, представленный в листинге 23.4, теперь является полностью функциональным, однако в нем все еще остаются две существенные избыточности. Во-первых, метод `addedToStageListener( )` всегда вызывает метод `stage.invalidate( )`, когда объект `Ellipse` добавляется в список отображения. В результате перерисовка фигуры происходит даже в тех случаях, когда в этом нет необходимости, поскольку за то время, пока эллипс не отображался на экране, данные концептуального эллипса не изменялись.

Во-вторых, не забывайте, что событие `Event.RENDER` возникает в том случае, когда *любой* объект — не обязательно текущий — вызывает метод `stage.invalidate( )`. Таким образом, в текущем состоянии метод `renderListener( )` будет вызывать метод `draw( )` всякий раз, когда любой объект в приложении вызывает метод `stage.invalidate( )`. В приложении с множеством объектов такая избыточность может привести к серьезным проблемам, связанным с производительностью.

Чтобы решить эти две оставшиеся проблемы, мы внесем последний набор изменений в класс `Ellipse` — добавим новую логику, позволяющую определить, требуется ли перерисовка эллипса при вызове методов `addedToStageListener( )` и `renderListener( )`. Во-первых, добавим новую переменную экземпляра `changed`, которая будет сообщать о том, требуется ли перерисовка объекта `Ellipse` при очередном обновлении экрана. Затем, чтобы устанавливать и сбрасывать значение переменной `changed`, а также проверять ее статус, добавим три новых метода: `setChanged( )`, `clearChanged( )` и `hasChanged( )`. Наконец, всякий раз при изменении эллипса (то есть всякий раз при вызове метода `setWidth( )`, `setHeight( )` или метода-конструктора) мы будем присваивать переменной `changed` значение `true`.

В листинге 23.5 представлена окончательная версия класса `Ellipse` с комментариями, которые помогут вам понять код (изменения выделены полужирным шрифтом). Как отмечалось ранее, класс, представленный в листинге 23.5, безусловно, является более сложным, чем его первоначальная, неоптимизированная версия из листинга 23.2. Однако в приложениях, требующих максимальной графической производительности, подход с отложенной визуализацией, применяемый в листинге 23.5, является просто незаменимым. Дополнительную информацию по классам, представляющим фигуры и использующим отложенную визуализацию, можно найти в гл. 25.

Листинг 23.5. Окончательная, оптимизированная версия класса Ellipse

```
package {
  import flash.display.Shape;
  import flash.events.*;

  public class Ellipse extends Shape {
    private var w:Number;
    private var h:Number;
    private var changed:Boolean;

    public function Ellipse (width:Number, height:Number) {
      // Регистрируем приемники для получения уведомлений, когда данный
      // объект добавляется в список отображения или удаляется из него
      var stageDetector:StageDetector = new StageDetector(this);
      stageDetector.addEventListener(StageDetector.ADDED_TO_STAGE,
                                     addedToStageListener);
      stageDetector.addEventListener(StageDetector.REMOVED_FROM_STAGE,
                                     removedFromStageListener);
      // Устанавливаем ширину и высоту
      w = width;
      h = height;
```

```
    // Помечаем, что объект изменился
    setChanged( );
  }

  public function setWidth (newWidth:Number):void {
    w = newWidth;
    setChanged( );
  }

  public function getWidth ( ):Number {
    return w;
  }

  public function setHeight (newHeight:Number):void {
    h = newHeight;
    setChanged( );
  }

  public function getHeight ( ):Number {
    return h;
  }

  // Помечает, что в данной фигуре что-то изменилось
  private function setChanged ( ):void {
    changed = true;
    if (stage != null) {
      stage.invalidate( );
    }
  }

  // Помечает, что самые последние изменения
  // были отображены на экране
  private function clearChanged ( ):void {
    changed = false;
  }

  // Сообщает о том, содержит ли данная фигура изменения,
  // которые еще не были отображены на экране
  protected function hasChanged ( ):Boolean {
    return changed;
  }

  // Приемник события вызывается при добавлении этой фигуры
  // в список отображения
  private function addedToStageListener (e:Event):void {
    // Регистрируем приемник для получения уведомлений
    // об обновлениях экрана
    stage.addEventListener(Event.RENDER, renderListener);
    // Если за то время, пока объект отсутствовал в списке отображения,
    // произошли какие-либо изменения, нарисуем эти изменения
    // при следующей визуализации экрана. Однако если с момента,
    // когда объект находился в списке отображения, никаких изменений
```

```
      // не произошло, в его отрисовке нет необходимости.
      if (hasChanged( )) {
        stage.invalidate( );
      }
    }

    // Приемник события вызывается при удалении этой фигуры из списка
    // отображения
    private function removedFromStageListener (e:Event):void {
      // Нет необходимости в получении событий обновления экрана, когда
      // данный объект не находится в списке отображения
      stage.addEventListener(Event.RENDER, renderListener);
    }

    // Приемник события вызывается перед обновлением экрана, если был вызван
    // метод stage.invalidate( )
    private function renderListener (e:Event):void {
      // Вызываем метод draw ( ), если изменения, внесенные в данную фигуру,
      // еще не были отображены.
      // Если это событие вызвал другой объект, но данный объект не был
      // изменен, перерисовка данного объекта выполняться не будет.
      if (hasChanged( )) {
        draw( );
      }
    }

    private function draw ( ):void {
      graphics.clear( );
      graphics.lineStyle(1);
      graphics.beginFill(0xFFFFFF, 1);
      graphics.drawEllipse(0, 0, w, h);
    }
  }
}
```

Заставим его двигаться!

Теперь, когда мы познакомились с системой обновления экрана среды выполнения Flash, можно рассмотреть, как с помощью кода создается анимация и движение. Звучит забавно? Увидимся в следующей главе.

Дополнительную информацию о системе обновления экрана среды выполнения Flash можно найти в дневнике Тиника Уро (Tinic Uro), инженера, работающего над созданием приложения Flash Player. Этот дневник расположен по адресу: http://www.kaourantin.net/2006/05/frame-rates-in-flash-player.html.

Программная анимация

В этой главе описываются базовые методики создания анимации с помощью языка ActionScript. Основное внимание уделяется интеграции кода для создания анимации с автоматическими обновлениями экрана среды выполнения Flash. Однако в этой главе не рассматриваются углубленные вопросы, связанные с анимацией, например программирование движения, подчиняющегося законам физики, перемещение по траектории, обнаружение столкновения, эффекты растровой анимации или преобразование цветов.

Никаких циклов

Чтобы создать анимацию на ActionScript, мы постоянно изменяем графическое содержимое с течением времени, создавая иллюзию движения. Например, чтобы анимировать объект `TextField`, перемещая его по экрану в горизонтальном направлении, мы будем постоянно увеличивать или уменьшать значение переменной экземпляра `x` этого объекта. В некоторых языках программирования естественным механизмом для постоянного изменения значения переменной экземпляра является инструкция цикла. Таким образом, новички в программировании на языке ActionScript могут подумать, что анимация создается с помощью цикла `while`, как та, что показана в следующем коде:

```
public class TextAnimation extends Sprite {
  public function TextAnimation ( ) {
    // Создаем объект TextField
    var t:TextField = new TextField( );
    t.text          = "Hello";
    t.autoSize      = TextFieldAutoSize.LEFT;
    addChild(t);

    // Циклически обновляем горизонтальную позицию объекта TextField
    // и останавливаемся в тот момент, когда объект достигнет
    // координаты 300 по оси x
    while (t.x <= 300) {
      t.x += 10;
    }
  }
}
```

Предыдущий цикл `while` многократно увеличивает значение переменной экземпляра `x` объекта `TextField`, но, как в варианте создания анимации, в нем допущена роковая ошибка: при выполнении тела цикла обновление экрана не происходит.

На каждой итерации цикла горизонтальное положение объекта `TextField` изменяется, однако визуальный эффект данного изменения не отображается на экране. Визуализация изображения происходит только после завершения последней итерации цикла и выхода из функции-конструктора класса `TextAnimation`. Следовательно, в тот момент, когда произойдет визуализация изображения, объект `TextField` уже будет находиться в точке с координатой 300 по оси X.

В языке ActionScript инструкции цикла не могут быть использованы для создания анимации. Помните, что обновление экрана никогда не происходит внутри блока кода. Дополнительную информацию можно найти в подразд. «Никаких обновлений экрана внутри блоков кода» разд. «Запланированные обновления экрана» гл. 23.

В языке ActionScript анимация создается не с помощью циклов, а с помощью многократного вызова функций, которые вносят визуальные изменения и затем завершаются, позволяя выполнить обновление экрана. Существует два механизма для многократного вызова подобных функций: события `Event.ENTER_FRAME` и `TimerEvent.TIMER`.

Создание анимации с помощью события ENTER_FRAME

Среда Flash осуществляет диспетчеризацию события `Event.ENTER_FRAME` всякий раз, когда выполняется проверка запланированного обновления экрана (как было описано в гл. 23). Любая функция, зарегистрированная для получения уведомлений о возникновении события `Event.ENTER_FRAME`, выполняется многократно с частотой, определяемой текущей скоростью кадров среды выполнения Flash. Визуальные изменения, вносимые любой функцией-приемником события `Event.ENTER_FRAME`, отображаются сразу после завершения этой функции.

Функция может быть зарегистрирована для получения уведомлений о возникновении события `Event.ENTER_FRAME` из любого экземпляра класса `DisplayObject`, независимо от того, находится в настоящий момент этот экземпляр в списке отображения или нет. В качестве примера используем событие `Event.ENTER_FRAME`, чтобы реализовать анимацию, рассмотренную в предыдущем разделе, — объект `TextField` перемещается по экрану в горизонтальном направлении до точки с координатой 300 по оси X. Начнем с создания класса `TextAnimation`, который создает объект `TextField` и затем добавляет его в список отображения.

```
public class TextAnimation extends Sprite {
  private var t:TextField;

  public function TextAnimation ( ) {
    // Создаем объект TextField
    t = new TextField( );
    t.text          = "Hello";
    t.autoSize      = TextFieldAutoSize.LEFT;
```

```
    addChild(t);
  }
}
```

Теперь создадим функцию-приемник `moveTextRight( )` для события `Event.ENTER_FRAME`, которая перемещает объект `TextField t` вправо на 10 пикселов. Многократный вызов функции `moveTextRight( )` создаст эффект анимации. Обратите внимание, что, поскольку функция `moveTextRight( )` является функцией-приемником события `Event.ENTER_FRAME`, она определяет единственный обязательный параметр, типом данных которого является `Event`.

```
public function moveTextRight (e:Event):void {
  t.x += 10;
}
```

Наконец, внутри функции-конструктора класса `TextAnimation` мы регистрируем функцию `moveTextRight( )` для событий `Event.ENTER_FRAME`. Как только она будет зарегистрирована в качестве функции-приемника событий `Event.ENTER_FRAME`, она будет многократно вызываться с течением времени, в соответствии со скоростью кадров среды выполнения Flash. Новый код выделен полужирным шрифтом:

```
public function TextAnimation ( ) {
  // Создаем объект TextField
  t = new TextField( );
  t.text          = "Hello";
  t.autoSize      = TextFieldAutoSize.LEFT;
  addChild(t);

  // Регистрируем функцию moveTextRight( )
  // для получения уведомлений
  // о возникновении события Event.ENTER_FRAME
  addEventListener(Event.ENTER_FRAME, moveTextRight);
}
```

В листинге 24.1 продемонстрирован класс `TextAnimation`, дополненный методом `moveTextRight( )`.

Листинг 24.1. Анимация объекта TextField по горизонтали

```
public class TextAnimation extends Sprite {
  private var t:TextField;

  public function TextAnimation ( ) {
    // Создаем объект TextField
    t = new TextField( );
    t.text          = "Hello";
    t.autoSize      = TextFieldAutoSize.LEFT;
    addChild(t);

    // Регистрируем функцию moveTextRight( ) для получения уведомлений
    // о возникновении события Event.ENTER_FRAME
    addEventListener(Event.ENTER_FRAME, moveTextRight);
  }
```

```
  public function moveTextRight (e:Event):void {
    t.x += 10;
  }
}
```

Всякий раз, когда при выполнении кода из листинга 24.1 среда Flash проверяет запланированное обновление экрана, она осуществляет диспетчеризацию события `Event.ENTER_FRAME`. В результате вызывается функция `moveTextRight( )`, и среда выполнения Flash обновляет экран. С течением времени многократный вызов функции `moveTextRight( )` создает анимацию объекта `TextField`, перемещая его по экрану. Однако пока функция `moveTextRight( )` перемещает объект `TextField t` вправо до бесконечности. Чтобы объект `TextField` не перемещался дальше позиции с координатой 300 по оси X, мы должны изменить метод `moveTextRight( )` — значение `10` должно прибавляться к значению переменной `t.x` только в том случае, когда значение переменной `t.x` меньше или равно `300`, как показано ниже.

```
public function moveTextRight (e:Event):void {
  // Добавляем 10 к значению переменной t.x только в том случае, когда
  // значение переменной t.x меньше или равно 300
  if (t.x <= 300) {
    t.x += 10;
    // Не допускаем, чтобы значение переменной t.x превысило 300
    if (t.x > 300) {
      t.x = 300;
    }
  }
}
```

Предыдущий код справляется с задачей остановить объект `TextField` в точке с координатой 300 по оси X, но при этом он позволяет методу `moveTextRight( )` бесполезно продолжать свое выполнение даже после того, как объект `TextField` достигнет заданной координаты. Чтобы избежать ненужных вызовов функции, мы отменяем регистрацию приемника `moveTextRight( )` для событий `Event.EVENT_FRAME`, когда объект достигает точки с координатой 300 по оси X. Вот этот код:

```
public function moveTextRight (e:Event):void {
  if (t.x <= 300) {
    t.x += 10;
    if (t.x > 300) {
      t.x = 300;
    }
  } else {
    // Прекращаем получать уведомления о возникновении события
    // Event.ENTER_FRAME
    removeEventListener(Event.ENTER_FRAME, moveTextRight);
  }
}
```

Теперь, когда мы изменили метод `moveTextRight( )`, чтобы получение уведомлений о возникновении события `Event.ENTER_FRAME` происходило только при необходимости, наш простой класс `TextAnimation` завершен. В листинге 24.2 показан окончательный код.

Листинг 24.2. Анимация объекта TextField по горизонтали до координаты 300 по оси X

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  public class TextAnimation extends Sprite {
    private var t:TextField;

    public function TextAnimation ( ) {
      // Создаем объект TextField
      t = new TextField( );
      t.text          = "Hello";
      t.autoSize      = TextFieldAutoSize.LEFT;
      addChild(t);

      // Регистрируем функцию moveTextRight( ) для получения уведомлений
      // о возникновении события Event.ENTER_FRAME
      addEventListener(Event.ENTER_FRAME, moveTextRight);
    }

    public function moveTextRight (e:Event):void {
      if (t.x <= 300) {
        t.x += 10;
        if (t.x > 300) {
          t.x = 300;
        }
      } else {
        // Прекращаем получать уведомления о возникновении события
        // Event.ENTER_FRAME
        removeEventListener(Event.ENTER_FRAME, moveTextRight);
      }
    }
  }
}
```

Обратите внимание, что в листинге 24.2 анимация одного объекта (`TextField`) осуществляется под управлением другого объекта (класса `TextAnimation`). Структура, в которой «один объект управляет анимацией другого объекта», является типичной для приложений с централизованным управлением анимацией. В подобных приложениях один класс выступает в роли режиссера для всех анимаций в приложении, регистрируя единственный метод для уведомлений о возникновении события `Event.ENTER_FRAME` и вызывая процедуры для создания анимации над всеми подчиняющимися объектами. В отличие от этого, в приложении с децентрализованным управлением анимацией отдельные классы управляют своей собственной анимацией самостоятельно, определяя свои собственные методы-приемники события `Event.ENTER_FRAME`. Для сравнения в листинге 24.3 показан подкласс `TextTo300` класса `TextField`. Здесь, как в предыдущем примере, объект перемещается в точку с координатой 300 по оси X, но делает это самостоятельно. Обратите внимание, что класс `TextTo300`

определяет методы `start( )` и `stop( )`, которые могут быть использованы для воспроизведения и остановки анимации.

Листинг 24.3. Децентрализованное управление анимацией

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  public class TextTo300 extends TextField {
    public function TextTo300 ( ) {
    }

    public function moveTextRight (e:Event):void {
      if (x <= 300) {
        x += 10;
        if (t.x > 300) {
          t.x = 300;
        }
      } else {
        stop( );
      }
    }

    public function start ( ):void {
      // Начинаем воспроизведение анимации
      addEventListener(Event.ENTER_FRAME, moveTextRight);
    }

    public function stop ( ):void {
      // Останавливаем анимацию
      removeEventListener(Event.ENTER_FRAME, moveTextRight);
    }
  }
}
```

Следующий код демонстрирует основной класс SWF-файла, в котором применяется класс `TextTo300` из листинга 24.3:

```
package {
  import flash.display.*;
  import flash.text.*;

  public class TextAnimation extends Sprite {
    private var t:TextTo300;

    public function TextAnimation ( ) {
      // Создаем экземпляр класса TextTo300
      t = new TextTo300( );
      t.text          = "Hello";
      t.autoSize      = TextFieldAutoSize.LEFT;
      addChild(t);
```

```
        // Запускаем анимацию
        t.start( );
    }
  }
}
```

Влияние скорости кадров на анимации, создаваемые с помощью события Event.ENTER_FRAME. Поскольку событие `Event.ENTER_FRAME` синхронизируется со скоростью кадров, для более высокой скорости функции-приемники события `Event.ENTER_FRAME` будут вызываться чаще по сравнению с более низкой скоростью. Таким образом, если мы перемещаем объект по экрану, используя функцию-приемник события `Event.ENTER_FRAME`, увеличение скорости кадров может предвещать увеличение скорости анимации.

Например, при программировании перемещения объекта `t` в предыдущем разделе мы неявно задавали его скорость относительно скорости кадров. Наш код говорит: «При прохождении каждого кадра перемещаем объект `t` на 10 пикселов вправо»:

```
ball_mc._x += 10;
```

Следовательно, скорость объекта `t` зависит от скорости кадров. Если скорость кадров равна 12 кадрам в секунду, то скорость перемещения объекта `t` составит 120 пикселов в секунду. Если скорость кадров равна 30 кадрам в секунду, скорость перемещения объекта `t` составит 300 пикселов в секунду!

При создании анимаций на базе сценариев вы можете поддаться искушению рассчитать расстояние перемещения элемента с учетом назначенной скорости кадров. Например, если назначенная скорость кадров составляет 20 кадров в секунду и мы хотим, чтобы элемент перемещался со скоростью 100 пикселов в секунду, естественно предположить, что скорость объекта должна составлять 5 пикселов за один кадр (5 пикселов × 20 кадров в секунду = 100 пикселов в секунду).

Однако у этого подхода есть два существенных недостатка.

- Всякий раз, изменяя назначенную скорость кадров, мы должны модифицировать весь код, который вычисляет скорость с учетом данной скорости кадров.
- Как было сказано в гл. 23, среда выполнения Flash не всегда достигает назначенной скорости кадров. Если компьютер слишком медленный, чтобы отображать кадры со скоростью, соответствующей назначенной, анимация будет замедляться. Это замедление может варьироваться даже в зависимости от загрузки системы; если запущены другие программы или если среда Flash выполняет некую задачу с интенсивной загрузкой процессора, то скорость кадров может упасть на короткий промежуток времени, после чего будет восстановлен нормальный темп.

В некоторых случаях анимация, воспроизводимая со слегка различающимися скоростями, может считаться допустимой. Но когда визуальная точность имеет не последнее значение или когда мы создаем, скажем, игру с активными действиями, гораздо более подходящим вариантом является определение расстояния для перемещения объекта с учетом прошедшего времени, а не в зависимости от назначенной скорости кадров. Дополнительную информацию можно найти далее, в разд. «Анимация, основанная на скорости».

Создание анимации с использованием события TimerEvent.TIMER

В предыдущем разделе рассказывалось о том, каким образом использовать событие `Event.ENTER_FRAME` для создания анимаций, которые синхронизируются со скоростью кадров среды выполнения Flash. В этом разделе мы познакомимся с тем, как синхронизировать анимации с произвольным промежутком времени, задаваемым с помощью класса `flash.utils.Timer`.

Класс `Timer` — это универсальный служебный класс, предназначенный для выполнения кода через указанный промежуток времени. Каждый объект `Timer` осуществляет диспетчеризацию событий `TimerEvent.TIMER` с частотой, устанавливаемой программистом. Функции, которые желают выполняться с данной частотой, регистрируются в объекте `Timer` для событий `TimerEvent.TIMER`.

Класс Timer не гарантирует частоту, с которой выполняются его функции-приемники. Если система или среда выполнения Flash оказалась занята в тот момент, когда объект Timer запланировал выполнить свои функции-приемники, выполнение функций будет отложено. Информацию о том, как избежать подобных задержек в анимации, можно найти в разд. «Анимация, основанная на скорости».

Для использования класса `Timer` необходимо выполнить следующую общую последовательность действий.

1. Создать новый объект `Timer`:

   ```
   var timer:Timer = new Timer( );
   ```

2. Установить частоту (в миллисекундах), с которой должны генерироваться события `TimerEvent.TIMER`. Например, следующий код устанавливает частоту, равную 100 мс (за одну секунду осуществляется диспетчеризация десяти событий `TimerEvent.TIMER`):

   ```
   timer.delay = 100;
   ```

3. Установить общее количество генерируемых событий `TimerEvent.TIMER`. Например, следующий код говорит таймеру сгенерировать всего пять событий `TimerEvent.TIMER`:

   ```
   timer.repeatCount = 5;
   ```

   Значение `0` сбрасывает ограничение (генерировать события `TimerEvent.TIMER` постоянно или до тех пор, пока не скажут остановиться):

   ```
   timer.repeatCount = 0;  // Бесконечные события TimerEvent.TIMER
   ```

4. Создать одну или несколько функций, которые будут периодически вызываться в ответ на события `TimerEvent.TIMER` объекта `Timer`. Функции, регистрируемые для событий `TimerEvent.TIMER`, должны определять единственный параметр с типом данных `TimerEvent`.

   ```
   function timerListener (e:TimerEvent):void {
     // Этот код будет выполняться при возникновении события TimerEvent.TIMER
   }
   ```

5. Зарегистрировать функцию, созданную на шаге 4, в объекте `Timer` для событий `TimerEvent.TIMER`:

   ```
   timer.addEventListener(TimerEvent.TIMER, timerListener);
   ```

6. Использовать метод экземпляра `start( )` класса `Timer`, чтобы запустить таймер. Как только таймер будет запущен, он начнет генерировать события `TimerEvent.TIMER` в соответствии со значениями, указанными для переменных `delay` и `repeatCount`:

   ```
   timer.start( );
   ```

В качестве альтернативы шагам 2 и 3 из приведенного списка установить значения переменных `delay` и `repeatCount` можно через параметры конструктора класса `Timer`, как показано в следующем коде:

```
var timer:Timer = new Timer(100, 5);  // Переменной delay присваиваем
                                      // значение 100, а переменной
                                      // repeatCount – значение 5
```

Следующий код демонстрирует применение предыдущих шагов в простой реальной ситуации: отображение слова «GO!» после трехсекундной задержки. Обратите внимание, что вторым аргументом функции-конструктора класса `Timer` является значение `1`. Это значит, что произойдет диспетчеризация только одного события `TimerEvent.TIMER`. Подобный код может быть использован в начале теста с ограниченным временем выполнения или гоночной игры.

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.utils.*;
  import flash.text.*;

  public class Race extends Sprite {
    private var startMsg:TextField;

    public function Race ( ) {
      // Выполнить приемники события TimerEvent.TIMER один раз спустя
      // 3 секунды
      var timer:Timer = new Timer(3000, 1);
      timer.addEventListener(TimerEvent.TIMER, timerListener);
      timer.start( );

      startMsg = new TextField( );
      startMsg.autoSize = TextFieldAutoSize.LEFT;
      startMsg.text = "Get Ready!";
      addChild(startMsg);
    }
    private function timerListener (e:TimerEvent):void {
      startMsg.text = "GO!";
    }
  }
}
```

Теперь, когда в целом мы чувствуем себя комфортно при работе с классом Timer, воспользуемся им для воссоздания класса TextAnimation из листинга 24.2. На этот раз вместо события Event.ENTER_FRAME для создания анимации будет использован объект Timer. Как и раньше, мы начинаем работу с определения конструктора и свойств класса. На этот раз мы добавим новую переменную экземпляра timer, которая будет ссылаться на наш объект Timer.

```
public class TextAnimation extends Sprite {
  private var t:TextField;
  private var timer:Timer;

  public function TextAnimation ( ) {
    // Создаем объект TextField
    t = new TextField( );
    t.text          = "Hello";
    t.autoSize      = TextFieldAutoSize.LEFT;
    addChild(t);
  }
}
```

Далее внутри конструктора класса TextAnimation мы создаем объект Timer, который будет генерировать события TimerEvent.TIMER каждые 50 мс (то есть 20 раз в секунду). После создания объекта Timer мы вызываем его метод start( ), после чего этот объект начинает генерировать события TimerEvent.TIMER.

```
public function TextAnimation ( ) {
  // Создаем объект TextField
  t = new TextField( );
  t.text          = "Hello";
  t.autoSize      = TextFieldAutoSize.LEFT;
  addChild(t);

  timer = new Timer(50, 0);
  timer.start( );
}
```

Затем мы создаем нашу функцию-приемник moveTextRight( ), которая перемещает объект TextField t вправо на 10 пикселов до тех пор, пока он не достигнет координаты 300 по оси X. На этот раз функция moveTextRight( ) принимает события TimerEvent.TIMER, а не Event.ENTER_FRAME, поэтому она определяет единственный обязательный параметр, типом данных которого является TimerEvent.

```
public function moveTextRight (e:TimerEvent):void {
}
```

Как и раньше, мы хотим остановить анимацию, когда объект t достигнет координаты 300 по оси X. Для этого мы вызываем метод stop( ) объекта Timer, который прекращает генерацию событий TimerEvent.TIMER:

```
public function moveTextRight (e:TimerEvent):void {
  if (t.x <= 300) {
    t.x += 10;
    if (t.x > 300) {
```

```
      t.x = 300;
    }
  } else {
    // Останавливаем таймер,
    // когда объект TextField достигнет
    // своего места назначения
    timer.stop( );
  }
}
```

Чтобы обновить экран сразу после выхода из функции `moveTextRight( )`, мы используем метод экземпляра `updateAfterEvent( )` класса `TimerEvent` (метод `updateAfterEvent( )` был подробно рассмотрен в гл. 23).

```
public function moveTextRight (e:TimerEvent):void {
  if (t.x <= 300) {
    t.x += 10;
    if (t.x > 300) {
      t.x = 300;
    }
    // Обновляем экран после выхода из этой функции
    e.updateAfterEvent( );
  } else {
    // Останавливаем таймер,
    // когда объект TextField достигнет
    // своего места назначения
    timer.stop( );
  }
}
```

Наконец, мы регистрируем функцию `moveTextRight( )` в объекте `Timer` для событий `TimerEvent.TIMER`. Мы выполняем регистрацию этой функции непосредственно перед запуском таймера в конструкторе класса `TextAnimation`, как показано в следующем коде:

```
public function TextAnimation ( ) {
  // Создаем объект TextField
  t = new TextField( );
  t.text          = "Hello";
  t.autoSize      = TextFieldAutoSize.LEFT;
  addChild(t);

  timer = new Timer(50, 0);
  timer.addEventListener(TimerEvent.TIMER, moveTextRight);
  timer.start( );
}
```

В листинге 24.4 представлен законченный код для версии класса `TextAnimation`, реализованной с использованием объекта `Timer`.

Листинг 24.4. Анимация объекта TextField по горизонтали до координаты 300 по оси X, версия с таймером

```
package {
  import flash.display.*;
```

```
  import flash.events.*;
  import flash.utils.*;
  import flash.text.*;

  public class TextAnimation extends Sprite {
    private var t:TextField;
    private var timer:Timer;

    public function TextAnimation ( ) {
      // Создаем объект TextField
      t = new TextField( );
      t.text          = "Hello";
      t.autoSize      = TextFieldAutoSize.LEFT;
      addChild(t);

      timer = new Timer(50, 0);
      timer.addEventListener(TimerEvent.TIMER, moveTextRight);
      timer.start( );
    }

    public function moveTextRight (e:TimerEvent):void {
      if (t.x <= 300) {
        t.x += 10;
        if (t.x > 300) {
          t.x = 300;
        }
        e.updateAfterEvent( );  // Обновляем экран после выхода
                                // из этой функции
      } else {
        // Останавливаем таймер, когда объект TextField достигнет
        // своего места назначения
        timer.stop( );
      }
    }
  }
}
```

Приведем еще одно сравнение между анимацией на основе объекта `Timer` и анимацией на основе события `Event.ENTER_FRAME` — в листинге 24.5 представлена аналогичная версия класса `TextTo300`, реализованная с использованием объекта `Timer` (класс `TextTo300` был приведен ранее в листинге 24.3).

Листинг 24.5. Класс TextTo300, версия с таймером

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.utils.*;
  import flash.text.*;

  public class TextTo300 extends TextField {
    private var timer:Timer;
```

```
    public function TextTo300 ( ) {
      timer = new Timer(50, 0);
      timer.addEventListener(TimerEvent.TIMER, moveTextRight);
    }

    public function moveTextRight (e:Event):void {
      if (x <= 300) {
        x += 10;
        if (t.x > 300) {
          t.x = 300;
        }
      } else {
        stop( );
      }
    }

    public function start ( ):void {
      // Начинаем воспроизведение анимации
      timer.start( );
    }

    public function stop ( ):void {
      // Останавливаем анимацию
      timer.stop( );
    }
  }
}
```

В листинге 24.6 показан еще один пример анимации, созданной с использованием объекта `Timer`. Это версия мигающего текста, реализованная на языке ActionScript 3.0.

Листинг 24.6. Мигающий текст

```
package {
  import flash.display.TextField;
  import flash.util.Timer;
  import flash.events.*;

  public class BlinkText extends TextField {
    private var timer:Timer;

    public function BlinkText (delay:Number = 1000) {
      timer = new Timer(delay, 0);
      timer.addEventListener(TimerEvent.TIMER, timerListener);
      timer.start( );
    }

    private function timerListener (e:TimerEvent):void {
      // Скрываем данный объект, если в настоящее время он отображается
      // на экране; или отображаем данный объект, если в настоящее время
      // он скрыт.
      visible = !visible;
```

```
      e.updateAfterEvent( );
    }

    public function setDelay (newDelay:Number):void {
      timer.delay = newDelay;
    }

    public function startBlink ( ):void {
      timer.start( );
    }

    public function stopBlink ( ):void {
      visible = true;
      timer.stop( );
    }
  }
}
```

Влияние скорости кадров на таймеры. Хотя на первый взгляд может показаться, что класс `Timer` предоставляет абсолютно независимый способ для выполнения некоторой функции через определенный интервал времени, он, как бы удивительно это ни звучало, все равно зависит от скорости кадров среды выполнения Flash. Событие `TimerEvent.TIMER` может возникать не чаще десяти раз для каждой проверки запланированного обновления экрана (то есть в десять раз чаще скорости кадров). Например, если скорость кадров равна 1 кадру в секунду, событие `TimerEvent.TIMER` может возникать в лучшем случае каждые 100 мс, даже когда для параметра `delay` объекта `Timer` указано меньшее значение. Для скорости, равной 10 кадрам в секунду, событие `TimerEvent.TIMER` может возникать 100 раз в секунду (каждые 10 мс). Для скорости, равной 100 кадрам в секунду, событие `TimerEvent.TIMER` может возникать 1000 раз в секунду (каждую 1 мс).

Когда установленная частота возникновения события `TimerEvent.TIMER` больше скорости кадров, функция будет выполняться на этапе следующего запланированного обновления экрана после истечения интервала времени. Чтобы запросить обновление экрана до следующего запланированного обновления, используйте метод экземпляра `updateAfterEvent( )` класса `TimerEvent`, рассмотренный в гл. 23.

Выбор между классом Timer и событием Event.ENTER_FRAME

Как мы уже знаем, и класс `Timer`, и событие `Event.ENTER_FRAME` могут применяться для создания анимации. Так что же из них больше подходит для ваших целей? Ниже перечислены основные факторы, влияющие на выбор.

Скорость кадров может изменяться. Когда SWF-файл *загружается другим приложением*, скорость кадров этого приложения может значительно отличаться от

назначенной скорости кадров данного SWF-файла. Это может приводить к слишком быстрому или слишком медленному воспроизведению анимации. Загружаемый SWF-файл может, конечно, установить собственную скорость кадров, однако такое изменение, возможно, окажет нежелательное влияние на воспроизведение в родительском приложении. Определенную независимость от скорости кадров обеспечивает класс `Timer` (ограничения были рассмотрены в разд. «Создание анимации с использованием события TimerEvent.TIMER»).

Использование множества объектов Timer требует больше памяти. В архитектурах с децентрализованным управлением анимацией использование отдельного объекта `Timer` для управления анимацией каждого объекта требует больше памяти, чем это необходимо для аналогичной реализации с использованием события `Event.ENTER_FRAME`.

Использование множества объектов Timer может приводить к избыточному количеству запросов на обновление экрана. В архитектурах с децентрализованным управлением анимацией использование отдельного объекта `Timer` вместе с методом `updateAfterEvent( )` для управления анимацией каждого объекта ведет к возникновению нескольких независимых запросов на обновление экрана, что может привести к проблемам с производительностью.

Исходя из перечисленных факторов, мы рекомендуем следующее.

- В приложениях, которые должны синхронизировать отображение содержимого, создаваемого программным путем, с отображением кадрового содержимого, создаваемого вручную в среде разработки Flash, используйте событие `Event.ENTER_FRAME`.
- В приложениях, где должно минимизироваться влияние колебаний скорости кадров среды Flash, применяйте один объект `Timer` для управления всеми анимациями и создавайте анимации, основанные на скорости (дополнительную информацию можно найти в разд. «Анимация, основанная на скорости»).
- Когда колебания скорости кадров среды выполнения Flash считаются приемлемыми, используйте событие `Event.ENTER_FRAME` (поскольку код для анимации, создаваемой с помощью события `Event.ENTER_FRAME`, в основном проще его эквивалента, реализуемого с помощью объекта `Timer`).
- Избегайте использования независимых объектов `Timer` для анимации отдельных отображаемых объектов. Там, где это возможно, используйте один объект `Timer` для управления всеми анимациями. С другой стороны, если вы хотите обновлять различные объекты через различные интервалы времени, подходящим решением могут оказаться независимые объекты `Timer`.

Обобщенный класс Animator

Во многих приложениях, которые были представлены в этой главе, код для создания графического объекта и код, занимающийся анимацией этого объекта, объединялись в одном классе. Для реальных приложений разумнее выносить код, создающий анимацию, во внешние классы и повторно применять его в других классах.

Простой класс Animator, представленный в листинге 24.7, демонстрирует один из возможных путей абстрагировать реализацию возможностей для создания анимации от анимируемых объектов. Каждый экземпляр класса Animator может перемещать указанный объект DisplayObject в заданную позицию за определенный промежуток времени. Стоит отметить, что в листинге 24.7 основное внимание уделяется структуре класса, который используется для вынесения кода, управляющего анимацией. В связи с этим класс Animator не реализует сложные анимационные возможности. Их можно реализовать в классе mx.effects.Tween платформы разработки Flex и классах fl.transitions.Tween и fl.motion.Animator среды разработки Flash CS3.

Ниже представлен листинг класса Animator. Понять код помогут комментарии.

Листинг 24.7. Обобщенный вспомогательный класс для создания анимации

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.utils.*;

  // Очень простой класс для создания анимации, который демонстрирует, как
  // отделить реализацию анимационных возможностей от анимируемого объекта
  public class Animator extends EventDispatcher {
    // Ссылается на анимируемый объект
    private var target:DisplayObject;
    // Всякий раз, когда начинается воспроизведение анимации, мы записываем
    // время начала в переменную startTime
    private var startTime:Number;
    // Длительность анимации, в миллисекундах
    private var duration:Number;
    // Всякий раз, когда начинается воспроизведение анимации, мы сохраняем
    // начальную позицию объекта target в переменных startX и startY
    private var startX:Number;
    private var startY:Number;
    // Всякий раз, когда начинается воспроизведение анимации, мы сохраняем
    // разницу между начальной и конечной позициями в переменных deltaX
    // и deltaY
    private var deltaX:Number;
    private var deltaY:Number;

    // Функция-конструктор принимает ссылку на анимируемый объект
    public function Animator (target:DisplayObject) {
      this.target = target;
    }

    // Начинает воспроизведение анимации, которая заключается в перемещении
    // объекта target из его текущей позиции в новую позицию с координатами
    // (toX, toY) за 'duration' миллисекунд
    public function animateTo (toX:Number, toY:Number,
                               duration:Number):void {
```

```
        // Запоминаем, где находился объект target в момент начала
        // воспроизведения данной анимации
        startX = target.x;
        startY = target.y;

        // Вычисляем разницу между начальной и конечной позициями
        // объекта target
        deltaX = toX - target.x;
        deltaY = toY - target.y;
        startTime = getTimer( );

        // Запоминаем длительность данной анимации
        this.duration = duration;

        // Начинаем получать уведомления о возникновении событий
        // Event.ENTER_FRAME. Всякий раз, когда происходит
        // запланированное обновление экрана, мы будем
        // обновлять позицию объекта target
        target.addEventListener(Event.ENTER_FRAME, enterFrameListener);
    }

    // Обрабатывает события Event.ENTER_FRAME.
    private function enterFrameListener (e:Event):void {
        // Вычисляем время, прошедшее с момента начала
        // воспроизведения анимации
        var elapsed:Number = getTimer( )-startTime;
        // Вычисляем, сколько времени прошло с момента начала воспроизведения
        // анимации в процентах от ее общей длительности
        var percentDone:Number = elapsed/duration;
        // Если анимация еще не завершена...
        if (percentDone < 1) {

            // ...обновляем позицию объекта target
            updatePosition(percentDone);
        } else {
            // ...в противном случае помещаем объект target в его конечное
            // положение и прекращаем получать уведомления о возникновении
            // событий Event.ENTER_FRAME
            updatePosition(1);
            target.removeEventListener(Event.ENTER_FRAME, enterFrameListener);
        }
    }

    // Устанавливает позицию объекта target, выраженную в процентах
    // от расстояния между начальной и конечной точками анимации
    private function updatePosition (percentDone:Number):void {
        target.x = startX + deltaX*percentDone;
        target.y = startY + deltaY*percentDone;
    }
}
```

Класс `SlidingText`, представленный далее в листинге 24.8, демонстрирует применение класса `Animator`. Каждый объект `SlidingText` — это текстовое поле, которое может быть перемещено в указанную позицию с использованием анимационных возможностей.

Платформа разработки Flex включает текстовые компоненты, которые могут быть анимированы и стилизованы с помощью множества настраиваемых эффектов. Дополнительную информацию можно получить в описании компонентов mx.controls.Text и mx.controls.TextArea справочника по платформе разработки Flex 2 корпорации Adobe.

Листинг 24.8. Класс, представляющий перемещающийся текст

```
package {
  import flash.text.*;

  public class SlidingText extends TextField {
    private var animator:Animator;

    public function SlidingText (toX:Number, toY:Number, duration:Number) {
      animator = new Animator(this);
      slideTo(toX, toY, duration);
    }

    public function slideTo (toX:Number, toY:Number, duration:Number):void {
      animator.animateTo(toX, toY, duration);
    }
  }
}
```

Класс `AnimationLibDemo`, представленный в листинге 24.9, демонстрирует, как использовать классы `Animator` и `SlidingText`. Он создает круг, который перемещается в текущую позицию указателя мыши, когда пользователь щелкает кнопкой мыши в области отображения среды выполнения Flash. Этот класс также перемещает текст «Hello» в позицию (300; 0) и закрепляет его там.

Листинг 24.9. Демонстрация использования класса Animator

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;

  // Демонстрирует использование класса Animator
  public class AnimationLibDemo extends Sprite {
    private var circleAnimator:Animator;

    public function AnimationLibDemo ( ) {
      // Создаем сообщение, которое с использованием анимационных
      // возможностей перемещается в позицию (300, 0)
      // за одну секунду (1000 мс)
      var welcome:SlidingText = new SlidingText(300, 0, 1000);
      welcome.text       = "Welcome!";
```

```
        welcome.autoSize   = TextFieldAutoSize.LEFT;
        addChild(welcome);

        // Создаем круг для анимации
        var circle:Shape = new Shape( );
        circle.graphics.lineStyle(10, 0x666666);
        circle.graphics.beginFill(0x999999);
        circle.graphics.drawCircle(0, 0, 25);
        addChild(circle);

        // Создаем объект Animator для анимации объекта circle
        circleAnimator = new Animator(circle);

        // Регистрируем приемник для событий мыши
        stage.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownListener);
      }

      // Когда пользователь щелкает кнопкой мыши на сцене, перемещаем
      // объект circle в точку, где произошел щелчок.
      private function mouseDownListener (e:MouseEvent):void {
        // Преобразуем точку из системы координат экземпляра класса Stage
        // в систему координат данного экземпляра класса
        var mousePt:Point = globalToLocal(new Point(e.stageX, e.stageY));
        circleAnimator.animateTo(mousePt.x, mousePt.y, 500);
      }
    }
}
```

При создании элементов управления пользовательского интерфейса с эффектами анимации рассмотрите возможность расширения класса mx.core.UIComponent платформы разработки Flex вместо разработки собственной библиотеки, реализующей анимационные возможности. В классе UIComponent предусмотрен расширенный набор инструментов для добавления эффектов к собственным элементам управления пользовательского интерфейса.

Анимация, основанная на скорости

В среде Flash невозможно гарантировать определенный временной интервал, через который будет выполняться проверка запланированных обновлений экрана или будут возникать события `TimerEvent.TIMER`. Событие `Event.ENTER_FRAME` обычно возникает позднее запланированного времени, которое определяется назначенной скоростью кадров, а события `TimerEvent.TIMER` — позднее времени, определяемого переменной `delay` объекта `Timer`.

Подобные задержки могут приводить к непредсказуемым эффектам в анимации. Чтобы гарантировать, что некоторый объект пройдет указанную дистанцию за определенный промежуток времени, мы должны устанавливать позицию этого объекта в соответствии с его скоростью (то есть в соответствии с его ускорением в определенном направлении). В листинге 24.10 продемонстрирована базовая методика.

Листинг 24.10. Вычисление позиции с учетом скорости

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.utils.*;

  // Перемещает объект на указанное количество пикселов в секунду,
  // независимо от скорости кадров
  public class Animation extends Sprite {
    private var distancePerSecond:int = 50;  // Количество пикселов,
                                             // на которое должен
                                             // перемещаться объект
                                             // за одну секунду
    private var now:int;                     // Текущее время
    private var then:int;                    // Время последнего обновления
                                             // экрана
    private var circle:Shape;                // Анимируемый объект

    public function Animation ( ) {
      // Создаем объект для анимации
      circle = new Shape( );
      circle.graphics.beginFill(0x0000FF, 1);
      circle.graphics.lineStyle(1);
      circle.graphics.drawEllipse(0, 0, 25, 25);
      addChild(circle);

      // Инициализируем значения времени
      then = getTimer( );
      now = then;

      // Регистрируем приемник для получения уведомлений о проверках
      // запланированных обновлений экрана
      addEventListener(Event.ENTER_FRAME, enterFrameListener);
    }

    // Обрабатываем события Event.ENTER_FRAME
    private function enterFrameListener (e:Event):void {
      // Вычисляем, сколько времени прошло с момента последнего перемещения
      then = now;
      now = getTimer( );
      var elapsed:int = now - then;
      var numSeconds:Number = elapsed / 1000;

      // Определяем количество пикселов, на которое должен переместиться
      // объект, учитывая промежуток времени, прошедший с момента
      // последнего перемещения
      var moveAmount:Number = distancePerSecond * numSeconds;

      // Перемещаем объект. В данном случае объект перемещается
      // под углом 0 градусов.
```

```
      circle.x += moveAmount;
    }
  }
}
```

Класс `Animator`, представленный ранее в листинге 24.7, использует скорость аналогичным образом, чтобы гарантировать, что анимируемый объект пройдет заданное расстояние за указанный промежуток времени.

Переходим к контурам и заливкам

На протяжении нескольких последних глав большую часть времени мы потратили на рассмотрение вопросов, связанных с интерактивностью и анимацией. В последующих трех главах мы уделим особое внимание созданию трех конкретных типов графического содержимого: векторов, растровых изображений и текстовых полей.

Рисование с помощью векторов

В языке ActionScript элементарные векторы, линии и фигуры рисуются с помощью класса `Graphics`. Тем не менее экземпляры класса `Graphics` никогда не создаются напрямую; вместо этого каждый класс языка ActionScript, поддерживающий программное векторное рисование, создает экземпляр класса `Graphics` автоматически и предоставляет к нему доступ через переменную экземпляра `graphics`. К отображаемым классам, поддерживающим векторное рисование, относятся классы `Sprite`, `MovieClip` и `Shape`.

Объекты Shape потребляют меньше памяти, чем объекты Sprite и MovieClip. Таким образом, чтобы уменьшить потребление памяти, векторное содержимое лучше рисовать в объектах Shape, когда нет необходимости в контейнерных и интерактивных возможностях классов Sprite и MovieClip.

Обзор класса Graphics

Как видно из табл. 25.1, инструменты рисования класса `Graphics` можно разбить на пять общих категорий: рисование линий, рисование фигур (также называемых *заливками*), определение стилей линий, перемещение чертежного пера и удаление графики.

Таблица 25.1. Обзор класса Graphics

Назначение	Метод класса Graphics
Рисование линий	curveTo( ), lineTo( )
Рисование фигур	beginBitmapFill( ), beginFill( ), beginGradientFill( ), drawCircle( ), drawEllipse( ), drawRect( ), drawRoundRect( ), drawRoundRectComplex( ), endFill( )
Определение стилей линий	lineGradientStyle( ), lineStyle( )
Перемещение чертежного пера	moveTo( )
Удаление графики	clear( )

По существу, в языке ActionScript линии и кривые рисуются гипотетическим «чертежным пером». Для всех новых объектов `Sprite`, `MovieClip` и `Shape` перо первоначально устанавливается в положение (0; 0). По мере рисования линий и кривых (с помощью методов `lineTo( )` и `curveTo( )`) перо перемещается по координатному пространству объекта, останавливаясь в конечной точке последней нарисованной линии или кривой. Текущая позиция пера всегда обозначает начальную точку следующей рисуемой линии или кривой. Чтобы установить начальную точку линии или кривой в произвольную позицию, мы используем метод `moveTo( )`, который

«поднимает» чертежное перо и перемещает его в новую позицию, не рисуя линию к указанной точке.

Рисование линий

Для рисования прямых линий используется метод `lineTo( )`, который рисует линию из текущей позиции чертежного пера в указанную точку (x; y). Например, следующий код создает новый объект `Shape` и рисует линию из точки (0; 0) в точку (25, 35):

```
var canvas:Shape = new Shape( );
canvas.graphics.lineTo(25, 35);
addChild(canvas);
```

Однако если вы попытаетесь выполнить этот код таким, как есть, то можете быть удивлены, обнаружив, что на экране не появилось никакой линии! По умолчанию все рисуемые линии и фигуры не имеют контура. Чтобы отобразить контур, мы должны использовать метод `lineStyle( )`, устанавливающий визуальные характеристики (толщина, цвет и т. д.) для всех линий и фигур, которые будут нарисованы в дальнейшем. Для справки ниже представлена сигнатура метода `lineStyle( )`, которая демонстрирует доступные визуальные параметры и их значения по умолчанию. Подробную информацию по каждому параметру можно найти в справочнике по языку ActionScript корпорации Adobe.

```
lineStyle(thickness:Number = 1.0,
          color:uint = 0,
          alpha:Number = 1.0,
          pixelHinting:Boolean = false,
          scaleMode:String = "normal",
          caps:String = null,
          joints:String = null,
          miterLimit:Number = 3)
```

Метод `lineStyle( )` должен вызываться явно для каждого нового объекта `Sprite`, `MovieClip` и `Shape`, иначе никакой контур отображен не будет (хотя заполняемые области могут рисоваться без контура).

Рассмотрим несколько примеров, демонстрирующих различные способы изменения стиля линий объекта `canvas`. Следующий код удаляет стиль линий (нарисованные в дальнейшем линии, кривые и заливки не будут иметь контура):

```
canvas.graphics.lineStyle( )
```

Следующий код устанавливает для стиля линий толщину 1 пиксел и сплошной черный цвет:

```
canvas.graphics.lineStyle(1)
```

Представленный далее код устанавливает для стиля линий толщину 1 пиксел и сплошной зеленый цвет:

```
canvas.graphics.lineStyle(1, 0x00FF00)
```

Следующий код устанавливает для стиля линий толщину 2 пиксела и зеленый цвет с прозрачностью 50 %:

```
canvas.graphics.lineStyle(1, 0x00FF00, 50)
```

Теперь нарисуем линию из точки (0; 0) в точку (25; 35), как раньше, но на этот раз применим контур синего цвета толщиной 3 пиксела, заставив эту линию появиться на экране:

```
var canvas:Shape = new Shape( );
canvas.graphics.lineStyle(3, 0x0000FF); // Применяем контур синего цвета
canvas.graphics.lineTo(25, 35);
addChild(canvas);
```

Стоит отметить, что если бы предыдущая линия была нарисована в объекте `Sprite` или `MovieClip`, содержащем дочерние отображаемые объекты, она бы отображалась позади этих объектов.

Дочерние отображаемые объекты всегда выводятся перед своим родителем и, следовательно, всегда заслоняют векторное содержимое, нарисованное с помощью языка ActionScript в этом родительском объекте.

Например, следующий код добавляет объект `TextField` в новый объект `Sprite` и затем рисует линию в этом объекте `Sprite`. Объект `TextField` перекрывает линию, поскольку является ребенком объекта `Sprite`.

```
// Создаем объект Sprite и помещаем его на экран
var container:Sprite = new Sprite( );
addChild(container);

// Создаем объект TextField
var msg:TextField = new TextField( );
msg.text = "Hello";
msg.border = true;
msg.background = true;
msg.autoSize = TextFieldAutoSize.LEFT;
container.addChild(msg);

// Рисуем линию
container.graphics.lineStyle(3, 0x0000FF);
container.graphics.lineTo(25, 35);
```

На рис. 25.1 показан результат выполнения предыдущего кода.

Рис. 25.1. Векторное содержимое позади объекта TextField

Содержимое, нарисованное с помощью объекта graphics в объекте Sprite или MovieClip, всегда отображается позади любых дочерних объектов данного объекта Sprite или MovieClip (то есть заслоняется).

При рисовании нескольких линий стиль контура можно установить для каждой в отдельности, вызывая метод `lineStyle( )` перед рисованием каждой линии. Например, следующий код рисует квадрат с помощью постепенно утолщаемых линий, окрашенных в черный, красный, зеленый и синий цвет:

```
var canvas:Shape = new Shape( );
canvas.graphics.lineStyle(1, 0x000000);
canvas.graphics.lineTo(100, 0);
canvas.graphics.lineStyle(5, 0xFF0000);
canvas.graphics.lineTo(100, 100);
canvas.graphics.lineStyle(10, 0x00FF00);
canvas.graphics.lineTo(0, 100);
canvas.graphics.lineStyle(15, 0x0000FF);
canvas.graphics.lineTo(0, 0);
addChild(canvas);
```

На рис. 25.2 показаны результаты выполнения предыдущего кода.

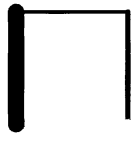

Рис. 25.2. Линии с изменяющейся толщиной

Обратите внимание, что на рис. 25.2 конечные точки линий (называемые *концами* линии) по умолчанию закруглены. Чтобы вместо закругленных концов линий выбрать квадратные, используйте параметр `caps` метода `lineStyle( )`. Например, следующий код создает линию зеленого цвета толщиной 10 пикселов с квадратными концами:

```
canvas.graphics.lineStyle(10, 0x00FF00, 1, false,
                          LineScaleMode.NORMAL, CapsStyle.SQUARE);
```

При нулевой толщине контур превращается в визирную линию (ее толщина составляет 1 пиксел и не изменяется даже при увеличении масштаба объекта или области отображения среды выполнения Flash). Другие варианты масштабирования линий можно установить параметром `scaleMode` метода `lineStyle( )`.

Чтобы полностью отключить отображение контура, установите толщине значение `undefined` или вызовите метод `lineStyle( )` без параметров. Например:

```
canvas.graphics.lineStyle(undefined);  // Выключаем линии в объекте canvas
canvas.graphics.lineStyle( );          // То же самое
```

Чтобы переместить чертежное перо, не рисуя при этом вообще никаких линий, используйте метод `moveTo( )`. Предположим, что мы хотим нарисовать в новом объекте `Shape` одну прямую линию из точки (100; 100) в точку (200; 200). Мы сначала перемещаем перо из точки (0; 0) в точку (100; 100), используя метод `moveTo( )`, а затем рисуем линию из этой точки в точку (200; 200) с помощью метода `lineTo( )`, как показано в следующем коде:

```
var canvas:Shape = new Shape( );  // Создаем объект Shape для рисования
canvas.graphics.lineStyle(1);     // Устанавливаем контур черного цвета
                                  // толщиной 1 пиксел
```

```
canvas.graphics.moveTo(100, 100); // Перемещаем перо, не рисуя линию
canvas.graphics.lineTo(200, 200); // Рисуем линию (в результате этого
                                  // действия происходит перемещение пера)
```

Рисование кривых

Для рисования кривых используется метод `curveTo( )`, который имеет следующую сигнатуру:

```
curveTo(controlX:Number, controlY:Number, anchorX:Number, anchorY:Number)
```

Метод `curveTo( )` рисует кривую Безье второго порядка из текущей позиции чертежного пера в точку (`anchorX`; `anchorY`), используя управляющую точку (`controlX`; `controlY`), которая находится за пределами кривой. В каждой конечной точке кривая касается линии, проходящей через данную конечную точку и управляющую точку. Кривая Безье содержится в выпуклой оболочке, построенной по ее трем управляющим точкам.

Говоря по существу, прямая линия, проходящая из позиции пера в конечную точку (`anchorX`; `anchorY`), натягивается управляющей точкой (`controlX`; `controlY`), формируя кривую. Если любой из аргументов метода `curveTo( )` будет опущен, операция завершится без сообщения об ошибке и позиция чертежного пера останется неизменной. Как и в случае с методом `lineTo( )`, характеристики контура кривой (толщина, цвет, прозрачность и т. д.) определяются самым последним вызовом функции `lineStyle( )`.

Следующий код рисует кривую черного цвета толщиной в 4 пиксела из позиции по умолчанию (0; 0) чертежного пера в якорную точку (100; 0), используя управляющую точку (50; 100). Кривая, получающаяся в результате выполнения этого кода, изображена на рис. 25.3.

```
var canvas:Shape = new Shape( );
addChild(canvas);

canvas.graphics.lineStyle(4);  // Устанавливаем контур толщиной 4 пиксела
                               // черного цвета
canvas.graphics.curveTo(50, 100, 100, 0);  // Рисуем кривую
```

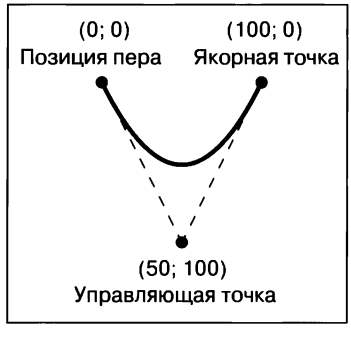

Рис. 25.3. Кривая Безье второго порядка

После того как кривая будет нарисована, чертежное перо останется в ее конечной точке. Таким образом, многократно вызывая функцию `curveTo( )` и/или `lineTo( )`, можно нарисовать сложные кривые или замкнутые фигуры, например окружности или многоугольники, — эта методика будет рассмотрена в следующем разделе.

Кривые, нарисованные на дробных пикселах, зачастую выглядят размытыми. Для увеличения резкости размытых, сглаженных линий присвойте параметру pixelHinting метода lineStyle( ) значение true.

Иногда бывает более удобно указать три точки на кривой, нежели две якорные точки и управляющую точку. Следующий код определяет пользовательский метод `curveThrough3Pts( )`, который принимает координаты трех точек в качестве аргументов и рисует кривую второго порядка, проходящую через эти точки. Предполагается, что вторая точка находится посередине кривой (`t = .5`):

```
// Адаптированный метод drawCurve3Pts( ) Роберта Пеннера (Robert Penner)
public function curveThrough3Pts (g:Graphics,startX:Number, startY:Number,
                                  throughX:Number, throughY:Number,
                                  endX:Number, endY:Number) {
  var controlX:Number = (2 * throughX) - .5 * (startX + endX);
  var controlY:Number = (2 * throughY) - .5 * (startY + endY);
  g.moveTo(startX, startY);
  g.curveTo(controlX, controlY, endX, endY);
}

// Использование
var canvas:Shape = new Shape( );
addChild(canvas);
canvas.graphics.lineStyle(2, 0x0000FF);
curveThrough3Pts(canvas.graphics, 100, 100, 150, 50, 200, 100);
```

Дополнительную информацию по теории кривых в языке ActionScript можно найти в статье «TechNotes» Джима Армстронга (Jim Armstrong) по адресу http://www.algorithmist.net/technotes.html.

Рисование фигур

Чтобы нарисовать фигуру произвольной формы (то есть закрасить некоторым цветом геометрическую область, образованную тремя или более точками), выполняйте такую последовательность действий.

1. Выберите начальную точку фигуры (либо точку по умолчанию (0, 0), либо точку, указанную методом `moveTo( )`).
2. Начните рисование с вызова метода `beginBitmapFill( )`, `beginFill( )` или `beginGradientFill( )`.
3. Нарисуйте контур фигуры, последовательно вызывая функцию `lineTo( )` и/или `curveTo( )`. Конечная точка последнего сегмента контура должна совпадать с начальной точкой, указанной на шаге 1.
4. Завершите рисование фигуры вызовом метода `endFill( )`.

Метод `beginFill( )` позволяет заполнить фигуру сплошным цветом; метод `beginGradientFill( )` — градиентом (переходом между двумя или более цветами); в то время как метод `beginBitmapFill( )` использует для заполнения фигуры указанное растровое изображение (при желании его можно разместить в виде мозаики).

Например, следующий код рисует треугольник красного цвета с контуром толщиной 5 пикселов черного цвета. Обратите внимание, что начальная точка контура (0; 0), устанавливаемая по умолчанию, совпадает с конечной точкой контура:

```
var triangle:Shape = new Shape( );
triangle.graphics.beginFill(0xFF0000, 1);
triangle.graphics.lineStyle(20);
triangle.graphics.lineTo(125, 125); // Рисует линию вниз и вправо
triangle.graphics.lineTo(250, 0);   // Рисует линию вверх и вправо
triangle.graphics.lineTo(0, 0);     // Рисует линию влево
triangle.graphics.endFill( );
addChild(triangle);
```

На рис. 25.4 показаны результаты выполнения предыдущего кода.

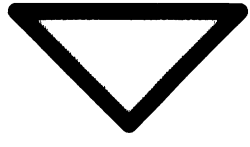

Рис. 25.4. Треугольник

Обратите внимание, что углы треугольника на рис. 25.4 закруглены. Стиль отображения углов задается параметром `joints` метода `lineStyle( )`. Например, следующий код изменяет стиль отображения углов, чтобы они выглядели «скошенными», — для этого в качестве параметра `joints` передается константа `JointStyle.MITER`:

```
triangle.graphics.lineStyle(20, 0, 1, false, LineScaleMode.NORMAL,
                            CapsStyle.ROUND, JointStyle.MITER);
```

На рис. 25.5 показан результат данного изменения. Обратите особое внимание на новую форму углов треугольника.

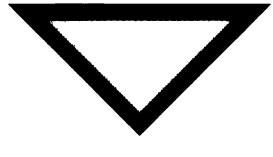

Рис. 25.5. Треугольник со скошенными соединениями

Для рисования различных видов прямоугольников и эллипсов класс `Graphics` предоставляет следующие удобные методы: `drawCircle( )`, `drawEllipse( )`, `drawRect( )`, `drawRoundRect( )` и `drawRoundRectComplex( )`. Методы round rect рисуют прямоугольники с закругленными углами. Все методы, упрощающие рисование фигур, используются совместно с уже знакомыми вам методами `lineStyle( )` и `beginFill( )`. Тем не менее вызывать метод `endFill( )`

после завершения рисования фигуры необязательно, поскольку каждый из перечисленных методов делает это автоматически.

Следующий код демонстрирует общий подход к использованию методов для рисования фигур. В нем применяется метод `drawRect( )` для рисования прямоугольника синего цвета с контуром толщиной 1 пиксел черного цвета:

```
var canvas:Shape = new Shape( );
addChild(canvas);

// Устанавливаем толщину, равную одному пикселу
canvas.graphics.lineStyle(1);
// Устанавливаем синий цвет заливки
canvas.graphics.beginFill(0x0000FF);
// Рисуем фигуру
canvas.graphics.drawRect(0, 0, 150, 75);
// Обратите внимание на отсутствие вызова метода endFill( )
```

Удаление векторного содержимого

Для удаления всего векторного содержимого из объекта применяется метод экземпляра `clear( )` класса `Graphics`. Например:

```
var canvas:Shape = new Shape( );
// Рисуем линию
canvas.graphics.lineStyle(3, 0x0000FF);  // Применяем контур синего цвета
canvas.graphics.lineTo(25, 35);
addChild(canvas);

// Удаляем линию
canvas.graphics.clear( );
```

Когда происходит вызов метода `clear( )`, стиль линий объекта возвращается к значению `undefined` (без контура). После вызова метода `clear( )` метод `lineStyle( )` должен быть вызван снова, иначе у линий и фигур будет отсутствовать контур. Вызов метода `clear( )` также сбрасывает позицию чертежного пера в точку (0; 0). Стоит отметить, что этот метод влияет на векторное содержимое только одного объекта. Если данный объект является экземпляром класса `Sprite` или `MovieClip`, то метод `clear( )` не удалит ни векторное содержимое в его дочерних объектах, ни сами дочерние объекты.

Следующий код каждые 250 мс рисует линию со случайными координатами и случайным стилем контура. Для удаления предыдущей нарисованной линии в нем используется метод `clear( )`.

```
package {
  import flash.display.*;
  import flash.utils.*;
  import flash.events.*;

  public class RandomLines extends Sprite {
    private var s:Shape;
```

```
    public function RandomLines ( ) {
      s = new Shape( );
      addChild(s);


      var t:Timer = new Timer(250);
      t.addEventListener(TimerEvent.TIMER, timerListener);
      t.start( );
    }

    private function timerListener (e:TimerEvent):void {
      s.graphics.clear( );
      s.graphics.lineStyle(random(1, 10), random(0, 0xFFFFFF));
      s.graphics.moveTo(random(0, 550), random(0, 400));
      s.graphics.lineTo(random(0, 550), random(0, 400));
    }

    // Возвращает число в диапазоне от minVal до maxVal включительно
    public function random (minVal:int, maxVal:int):int {
      return minVal + Math.floor(Math.random( ) * (maxVal + 1 - minVal));
    }
  }
}
```

Упражнение: для сравнения попробуйте удалить вызов метода `clear( )` в первой строке метода `timerListener( )`.

Пример: библиотека объектно-ориентированных фигур

Графическое содержимое, создаваемое с помощью внутренних методов рисования фигур (`drawEllipse( )`, `drawRect( )` и т. д.), не связано ни с каким объектом. После того как оно будет нарисовано, его невозможно изменить или удалить независимо от другого содержимого. В ActionScript отсутствуют классы, предоставляющие объектно-ориентированный доступ к соответствующим фигурам на экране. В этом разделе продемонстрирован один из способов устранения данного недостатка — пример реализации небольшой библиотеки классов для создания и управления фигурами как объектами.

В нашей библиотеке фигур представлены следующие классы: `BasicShape`, `Rectange`, `Ellipse`, `Star` и `Polygon`. Класс `BasicShape` — это базовый класс библиотеки. Он расширяет внутренний класс `Shape`, который предоставляет базовую поверхность с минимальным потреблением ресурсов для рисования фигур. Класс `BasicShape` управляет стилями контура и заливки для всех фигур и определяет, когда фигура должна быть перерисована. Экземпляры остальных классов представляют геометрические фигуры, которые могут добавляться в список отображения или удаляться из него. Класс каждой фигуры реализует свою собственную процедуру рисования, учитывающую особенности построения

данной фигуры. После создания объекта фигуры ее штриховка, заливка и контур могут быть легко изменены.

В следующих шести примерах применяются на практике многие методики, которые были рассмотрены в этой книге, — особенно те, которые были рассмотрены в этой главе. Обращайте пристальное внимание на многочисленные комментарии; они помогут вам понять код.

В листинге 25.1 представлен класс `BasicShape` — класс абстрактного типа, который определяет базовую функциональность для всех фигур в нашей библиотеке классов. Его основные возможности реализуются с помощью следующих методов:

- `setStrokeStyle( )` и `setFillStyle( )` — сохраняют визуальные характеристики фигуры;
- `draw( )` — отображает фигуру, но делегирует конкретное рисование линий методу `drawShape( )` (абстрактному методу, реализуемому всеми подклассами);
- `setChanged( )`, `clearChanged( )` и `hasChanged( )` — позволяют отслеживать, изменились ли параметры фигуры с момента ее последнего отображения на экране;
- `requestDraw( )`, `addedListener( )`, `removedListener( )`, а также `renderListener( )` — в сочетании эти методы определяют момент, когда необходимо перерисовать фигуру.

Листинг 25.1. Класс BasicShape

```
package org.moock.drawing {
  import flash.display.*;
  import flash.events.*;
  import flash.errors.IllegalOperationError;

  // Базовый класс для отображаемых геометрических фигур
  public class BasicShape extends Shape {
    // Стиль заливки
    protected var fillColor:Number = 0xFFFFFF;
    protected var fillAlpha:Number = 1;

    // Стиль линий. Используются скошенные соединения вместо закругленных
    // (настройка по умолчанию в языке ActionScript). Все остальные
    // первоначальные настройки совпадают с настройками по умолчанию языка
    // ActionScript.
    protected var lineThickness:Number = 1;
    protected var lineColor:uint = 0;
    protected var lineAlpha:Number = 1;
    protected var linePixelHinting:Boolean = false;
    protected var lineScaleMode:String = LineScaleMode.NORMAL;
    protected var lineJoints:String = JointStyle.MITER;
    protected var lineMiterLimit:Number = 3;

    // Флажок, указывающий на необходимость перерисовки объекта.
    // Предотвращает перерисовку данного объекта в тех случаях, когда
```

```
// событие RENDER было запрошено другим объектом.
protected var changed:Boolean = false;

// Конструктор
public function BasicShape ( ) {
  // Регистрируем приемники для получения уведомлений, когда данный
  // объект добавляется в список отображения или удаляется из него
  // (требуется пользовательский вспомогательный класс StageDetector)
  var stageDetector:StageDetector = new StageDetector(this);
  stageDetector.addEventListener(StageDetector.ADDED_TO_STAGE,
                                 addedToStageListener);
  stageDetector.addEventListener(StageDetector.REMOVED_FROM_STAGE,
                                 removedFromStageListener);
}

// Устанавливает визуальные характеристики линии,
// отображаемой вокруг фигуры
public function setStrokeStyle (thickness:Number = 1,
                                color:uint = 0,
                                alpha:Number = 1,
                                pixelHinting:Boolean = false,
                                scaleMode:String = "normal",
                                joints:String = "miter",
                                miterLimit:Number = 10):void {
  lineThickness = thickness;
  lineColor = color;
  lineAlpha = alpha;
  linePixelHinting = pixelHinting;
  lineScaleMode = scaleMode;
  lineJoints = joints;
  lineMiterLimit = miterLimit;

  // Стиль линий изменился, поэтому просим уведомить нас о следующем
  // обновлении экрана. В тот момент перерисовываем фигуру.
  setChanged( );
}

// Устанавливает визуальные характеристики заливки фигуры
public function setFillStyle (color:uint = 0xFFFFFF,
                              alpha:Number = 1):void {
  fillColor = color;
  fillAlpha = alpha;

  // Стиль заливки изменился, поэтому просим уведомить нас о следующем
  // обновлении экрана. В тот момент перерисовываем фигуру.
  setChanged( );
}

// Создает графику фигуры, делегируя специфические операции рисования
// линий конкретным подклассам класса BasicShape. Для повышения
// производительности метод draw( ) вызывается только в тот момент,
```

```
// когда объект stage рассылает событие Event.RENDER.
private function draw ( ):void {
  // Удаляет всю графику из данного объекта.
  graphics.clear( );
  // Стиль концов линий не имеет значения для замкнутой фигуры, поэтому
  // передаем значение null для этого аргумента.
  graphics.lineStyle(lineThickness, lineColor, lineAlpha,
                     linePixelHinting, lineScaleMode, null,
                     lineJoints, lineMiterLimit);
  graphics.beginFill(fillColor, fillAlpha);
  drawShape( );  // Вызываем процедуру рисования, реализуемую подклассом
  graphics.endFill( );

  // Делаем пометку, что на экране отображены самые последние изменения.
  clearChanged( );
}


// Рисует фактические линии для фигуры каждого типа. Должен быть
// реализован всеми подклассами класса BasicShape.
protected function drawShape ( ):void {
  // Предотвращает непосредственный вызов данного метода абстрактного типа
  throw new IllegalOperationError("The drawShape( ) method can be "
                      + "invoked on BasicShape subclasses only.")
}

// Помечает, что в фигуре изменился какой-то параметр. Если на настоящий
// момент фигура отображается на сцене, заставляет ее перерисоваться
// на этапе следующей визуализации экрана
protected function setChanged ( ):void {
  changed = true;
  requestDraw( );
}

// Помечает, что самые последние изменения были отображены на экране
protected function clearChanged ( ):void {
  changed = false;
}

// Сообщает о том, были ли изменены какие-либо параметры фигуры, которые
// еще не были визуализированы
protected function hasChanged ( ):Boolean {
  return changed;
}

// Если данная фигура отображается на экране, метод requestDraw( )
// заставляет ее перерисоваться на этапе следующего обновления экрана
protected function requestDraw ( ):void {
  if (stage != null) {
    stage.invalidate( );
  }
}
```

```
    // Приемник события, вызываемый при добавлении этой фигуры
    // в список отображения
    private function addedToStageListener (e:Event):void {
      // Регистрируем приемник для получения уведомлений
      // об обновлениях экрана
      stage.addEventListener(Event.RENDER, renderListener);

      // Если объект изменился за то время, пока он отсутствовал в списке
      // отображения, отображаем эти изменения на этапе следующей
      // визуализации экрана. Однако если объект не изменился с момента,
      // когда он находился в списке отображения последний раз,
      // нет необходимости перерисовывать его.
      if (hasChanged( )) {
        requestDraw( );
      }
    }

    // Приемник события, вызываемый при удалении данной фигуры
    // из списка отображения
    private function removedFromStageListener (e:Event):void {
      // Нет необходимости получать уведомления о возникновении событий
      // Event.RENDER, когда объект не находится в списке отображения
      stage.removeEventListener(Event.RENDER, renderListener);
    }

    // Приемник события, вызываемый перед обновлением экрана, если был
    // вызван метод stage.invalidate( ).
    private function renderListener (e:Event):void {
      // Вызываем метод draw( ), если изменения, внесенные в данную фигуру,
      // не были отображены на экране.
      // Если событие для визуализации было запрошено другим объектом,
      // но данный объект не изменялся, он не будет перерисован.
      if (hasChanged( )) {
        draw( );
      }
    }
  }
}
```

В листинге 25.2 представлен класс `Ellipse` — подкласс класса `BasicShape`. Обратите внимание, что специфический код, предназначенный для рисования эллипса, содержится в методе `drawShape( )`. Более того, изменение размеров объекта `Ellipse` не приводит к немедленной перерисовке эллипса. Вместо этого, когда вызывается метод `setSize( )`, объект вызывает метод `setChanged( )`, сообщая о том, что данный объект должен быть перерисован на этапе следующей визуализации экрана, осуществляемой средой выполнения Flash.

Листинг 25.2. Класс Ellipse

```
package org.moock.drawing {
  // Представляет эллипс, который может быть нарисован на экране
  public class Ellipse extends BasicShape {
    // Ширина и высота эллипса
```

```
    protected var w:Number;
    protected var h:Number;

    // Конструктор
    public function Ellipse (width:Number = 100, height:Number = 100) {
      super( );
      setSize(width, height);
    }

    // Процедура рисования эллипса
    override protected function drawShape ( ):void {
      graphics.drawEllipse(0, 0, w, h);
    }

    // Устанавливает ширину и высоту эллипса
    public function setSize (newWidth:Number, newHeight:Number):void {
      w = newWidth;
      h = newHeight;

      // Поскольку установка ширины и высоты эллипса вызывает изменение его
      // формы, он должен быть перерисован на этапе следующей визуализации
      // экрана.
      setChanged( );
    }
  }
}
```

В листинге 25.3 представлен класс `Polygon` — еще один подкласс класса `BasicShape`. С помощью класса `Polygon` можно нарисовать любую многогранную фигуру. Он выступает в роли суперкласса для определенных типов многоугольников, например `Rectangle` и `Star`. Как и `Ellipse`, класс `Polygon` предоставляет свою собственную специфическую процедуру рисования в методе `drawShape( )`. Всякий раз, когда задаются точки объекта `Polygon` (через метод `setPoints( )`), вызывается метод `setChanged( )`, сообщающий о том, что данный объект должен быть перерисован на этапе следующей визуализации экрана, осуществляемой средой выполнения Flash.

Листинг 25.3. Класс Polygon

```
package org.moock.drawing {
  // Представляет многоугольник, который может быть нарисован на экране
  public class Polygon extends BasicShape {
    // Точки многоугольника.
    // Чтобы сократить потребление памяти, точки хранятся в виде двух
    // массивов целочисленных значений, а не в виде одного массива объектов
    // flash.geom.Point.
    private var xpoints:Array;
    private var ypoints:Array;

    // Конструктор
    public function Polygon (xpoints:Array = null, ypoints:Array = null) {
      super( );
```

```
      setPoints(xpoints, ypoints);
    }

    // Процедура рисования многоугольника
    override protected function drawShape ( ):void {
      // Рисуем линии в каждую точку многоугольника
      graphics.moveTo(xpoints[0], ypoints[0]);
      for (var i:int = 1; i < xpoints.length; i++) {
        graphics.lineTo(xpoints[i], ypoints[i]);
      }
      // Замыкаем фигуру, возвращаясь в первую точку
      graphics.lineTo(xpoints[0], ypoints[0]);
    }

    // Присваивает точки многоугольника
    public function setPoints (newXPoints:Array, newYPoints:Array):void {
      if (newXPoints == null || newYPoints == null) {
        return;
      }

      if (newXPoints.length != newYPoints.length) {
        throw new Error("setPoints( ) requires a matching "
                        + "number of x and y points");
      }

      xpoints = newXPoints;
      ypoints = newYPoints;

      // Поскольку присваивание новых точек многоугольника вызывает
      // изменение его формы, он должен быть перерисован на этапе
      // следующей визуализации экрана.
      setChanged( );
    }
  }
}
```

В листинге 25.4 представлен класс `Rectangle` — подкласс класса `Polygon`. По структуре он похож на класс `Ellipse`, однако использует процедуру рисования из метода экземпляра `drawShape ( )` класса `Polygon`, а не реализует собственную.

Листинг 25.4. Класс Rectangle

```
package org.moock.drawing {
  // Представляет прямоугольник, который может быть нарисован на экране
  public class Rectangle extends Polygon {
    // Ширина и высота прямоугольника
    protected var w:Number;
    protected var h:Number;

    // Конструктор
    public function Rectangle (width:Number = 100, height:Number = 100) {
      super( );
```

```
      setSize(width, height);
    }

    // Устанавливает ширину и высоту многоугольника
    public function setSize (newWidth:Number, newHeight:Number):void {
      w = newWidth;
      h = newHeight;

      // Переводим ширину и высоту в точки многоугольника
      setPoints([0,w,w,0],[0,0,h,h]);
    }
  }
}
```

В листинге 25.5 представлен последний класс из библиотеки: `Star`, еще один подкласс класса `Polygon`. Как и `Rectangle`, класс `Star` использует метод экземпляра `drawShape( )` класса `Polygon` для отображения своего контура. Визуальные характеристики каждого объекта `Star` задаются через метод `setStar( )`.

Листинг 25.5. Класс Star

```
package org.moock.drawing {
  // Представляет фигуру звезды, которая может быть нарисована на экране
  public class Star extends Polygon {
    // Конструктор
    public function Star (numTips:int,
                          innerRadius:Number,
                          outerRadius:Number,
                          angle:Number = 0) {
      super( );
      setStar(numTips, innerRadius, outerRadius, angle);
    }

    // Устанавливает физические характеристики звезды.
    // За основу взяты методы рисования Рика Эвинга (Ric Ewing) для языка
    // ActionScript 1.0, доступные по адресу:
    // http://www.adobe.com/devnet/flash/articles/adv_draw_methods.html
    //   numTips      Количество концов (должно быть 3 или больше)
    //   innerRadius  Радиус основания концов
    //   outerRadius  Радиус вершин концов
    //   angle        Начальный угол в градусах (по умолчанию 0)
    public function setStar (numTips:int,
                             innerRadius:Number,
                             outerRadius:Number,
                             angle:Number = 0):void {
      // Вычисляем точки многоугольника-звезды
      if (numTips > 2) {
        // Инициализируем переменные
        var pointsX:Array = [];
        var pointsY:Array = [];
        var centerX:Number = outerRadius;
        var centerY:Number = outerRadius;
```

```
        var step:Number, halfStep:Number,
            startAngle:Number, dx:Number, dy:Number;
        // Вычисляем расстояние между концами
        step = (Math.PI*2)/numTips;
        halfStep = step/2;
        // Вычисляем начальный угол, в радианах
        startAngle = (angle/180)*Math.PI;
        // Задаем начальную точку
        pointsX[0] = centerX+(Math.cos(startAngle)*outerRadius);
        pointsY[0] = centerY-(Math.sin(startAngle)*outerRadius);
        // Добавляем оставшиеся точки
        for (var i:int=1; i <= numTips; i++) {
          dx = centerX+Math.cos(startAngle+(step*i)-halfStep)*innerRadius;
          dy = centerY-Math.sin(startAngle+(step*i)-halfStep)*innerRadius;
          pointsX.push(dx);
          pointsY.push(dy);
          dx = centerX+Math.cos(startAngle+(step*i))*outerRadius;
          dy = centerY-Math.sin(startAngle+(step*i))*outerRadius;
          pointsX.push(dx);
          pointsY.push(dy);
        }
        // Сохраняем вычисленные точки звезды
        setPoints(pointsX,pointsY);
      }
    }
  }
}
```

Наконец, в листинге 25.6 представлен класс `ShapeRandomizer`, который демонстрирует использование библиотеки классов фигур из пяти предыдущих примеров. Метод-конструктор класса `ShapeRandomizer` создает четыре фигуры. Щелчок кнопкой мыши на сцене случайным образом изменяет штриховку, заливку и контуры данных фигур.

Листинг 25.6. Класс ShapeRandomizer

```
package {
  import flash.display.Sprite;
  import flash.events.MouseEvent;

  import org.moock.drawing.*;

  // Демонстрация использования библиотеки org.moock.drawing.
  // Создает случайные фигуры, когда пользователь
  // щелкает кнопкой мыши на сцене.
  public class ShapeRandomizer extends Sprite {

    // Фигуры
    private var rect:Rectangle;
    private var ell:Ellipse;
    private var poly:Polygon;
    private var star:Star;
    // Конструктор
```

```
public function ShapeRandomizer ( ) {
  // Создаем прямоугольник
  rect = new Rectangle(50, 100);
  rect.setStrokeStyle(1, 0xFF0000);
  rect.setFillStyle(0x0000FF);

  // Создаем эллипс
  ell = new Ellipse(250, 50);
  ell.setStrokeStyle(2, 0xFFFF00);
  ell.setFillStyle(0xED994F);

  // Создаем треугольник (то есть трехгранный объект Polygon)
  poly = new Polygon([0, 50, 100], [50, 0, 50]);
  poly.setStrokeStyle(4, 0x333333);
  poly.setFillStyle(0x00FF00);

  // Создаем звезду
  star = new Star(5, 30, 80);
  star.setStrokeStyle(4, 0x666666);
  star.setFillStyle(0xFF0000);

  // Добавляем фигуры в список отображения
  addChild(rect);
  addChild(ell);
  addChild(poly);
  addChild(star);

  // Регистрируем приемник для событий
  // щелчка кнопкой мыши
  stage.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownListener);
}

// Приемник событий, вызываемый в тот момент, когда пользователь щелкает
// в области отображения среды выполнения Flash Player
private function mouseDownListener (e:MouseEvent):void {
  // Изменяем фигуры случайным образом
  rect.width = random(1, 300);
  rect.height = random(1, 300);
  rect.setStrokeStyle(random(1, 10), random(0, 0xFFFFFF));
  rect.setFillStyle(random(0, 0xFFFFFF), Math.random( ));

  ell.width = random(1, 300);
  ell.height = random(1, 300);
  ell.setStrokeStyle(random(1, 10), random(0, 0xFFFFFF));
  ell.setFillStyle(random(0, 0xFFFFFF), Math.random( ));

  poly.setPoints([random(1, 300), random(1, 300), random(1, 300)],
              [random(1, 300), random(1, 300), random(1, 300)]);
  poly.setStrokeStyle(random(1, 10), random(0, 0xFFFFFF));
  poly.setFillStyle(random(0, 0xFFFFFF), Math.random( ));
  star.setStar(random(3, 15), random(10, 20), random(30, 80));
```

```
      star.setStrokeStyle(random(1, 10), random(0, 0xFFFFFF));
      star.setFillStyle(random(0, 0xFFFFFF), Math.random( ));
    }

    // Возвращает число в диапазоне от minVal до maxVal включительно
    public function random (minVal:int, maxVal:int):int {
      return minVal + Math.floor(Math.random( ) * (maxVal + 1 - minVal));
    }
  }
}
```

На рис. 25.6 продемонстрирован набор случайных фигур, созданных классом `ShapeRandomizer`.

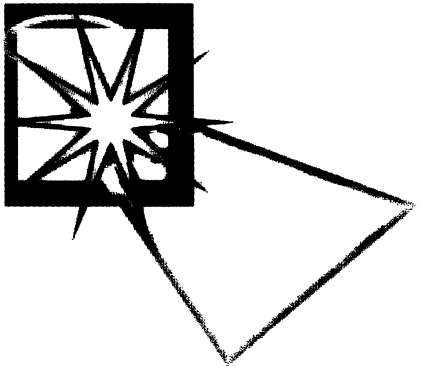

Рис. 25.6. Фигуры, созданные классом ShapeRandomizer

От линий к пикселам

В этой главе вы узнали, как создавать векторное графическое содержимое и управлять им. В следующей главе мы рассмотрим методики создания и управления *растровым* графическим содержимым.

Дополнительную информацию по векторной графике можно найти в документации по классу Graphics в справочнике по языку ActionScript корпорации Adobe.

Программирование растровой графики

В терминах программирования *растровое изображение* — это изображение, которое хранится в растровом формате данных. Растровый формат данных представляет изображение в виде прямоугольной сетки пикселов, в которой каждому пикселу присваивается число, обозначающее его цвет.

Например, если изображение, ширина и высота которого составляют 16 пикселов, представить в растровом формате данных, то оно будет сохранено в виде списка, состоящего из 256 чисел, каждое из которых обозначает конкретный цвет. Этот пример проиллюстрирован на рис. 26.1 — изображение размером 16 × 16 пикселов увеличено с целью отображения его отдельных пикселов. В правой части рисунка указаны позиции в таблице и значения цвета для трех выбранных в изображении пикселов. Обратите внимание, что позиции в таблице начинаются с нуля, поэтому координатой левого верхнего пиксела является (0; 0), а правого нижнего — (15; 15).

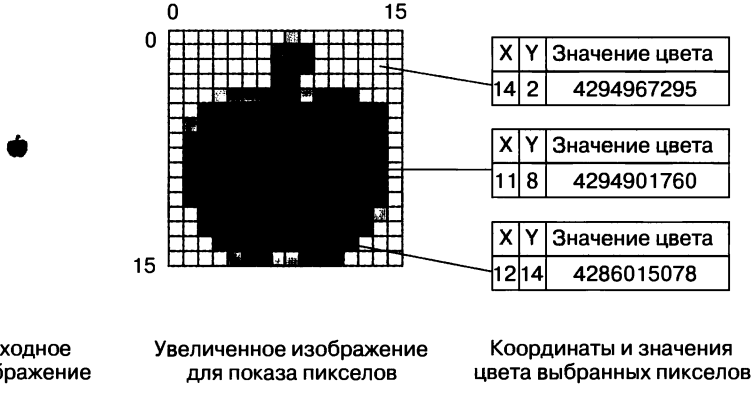

Рис. 26.1. Пример растрового изображения

В этой главе мы рассмотрим ряд распространенных методик, связанных с программированием растровой графики. Тем не менее имейте в виду, что исчерпывающему описанию программирования растровой графики в языке ActionScript могла бы быть посвящена целая книга.

В качестве источника информации для дальнейшего изучения можно использовать справочник по языку ActionScript корпорации Adobe.

Классы BitmapData и Bitmap

В языке ActionScript класс `BitmapData` представляет данные изображения в растровом формате наподобие того, которое показано на рис. 26.1. Каждый экземпляр класса `BitmapData` содержит список значений цвета пикселов и переменные экземпляра `width` и `height`, которые управляют размещением этих пикселов на экране. С помощью класса `BitmapData` мы можем создать данные для совершенно нового растрового изображения или просмотреть и изменить данные любого существующего растрового изображения, включая растровые изображения, загружаемые извне.

Класс `BitmapData` предоставляет широкий набор инструментов для установки или получения значения цвета некоторого пиксела или группы пикселов, а также для создания распространенных графических эффектов, например эффекта размытия или падающей тени. Как мы увидим далее, класс `BitmapData` может быть использован даже для создания анимированных эффектов и определения столкновений на основе растровых изображений. Для работы с большей частью инструментов класса `BitmapData` мы должны получить представление о формате, используемом языком ActionScript для описания значения цвета пиксела, который рассматривается в следующем разделе.

Как видно из названия, объект `BitmapData` сам по себе не является изображением; он лишь хранит представляющие изображение данные в растровом формате. Чтобы создать реальное, отображаемое на экране изображение, используя информацию, хранящуюся в объекте `BitmapData`, мы должны соединить этот объект с экземпляром класса `Bitmap`, как описано далее в разд. «Создание нового растрового изображения». Класс `Bitmap` является потомком класса `DisplayObject`, осуществляющим отображение объекта `BitmapData` на экране.

При работе с растровыми изображениями класс Bitmap используется для управления изображением как отображаемым объектом, а класс BitmapData — для управления нижележащими пиксельными данными изображения.

Благодаря разделению представления изображения (класс `Bitmap`) и хранилища данных (класс `BitmapData`), архитектура растровых изображений языка ActionScript позволяет нескольким различным объектам `Bitmap` одновременно отображать один и тот же объект `BitmapData`, при этом каждый объект `Bitmap` может иметь собственные визуальные характеристики (то есть различный масштаб, угол поворота, обрезку, фильтры, преобразования и прозрачность).

Перед тем как познакомиться с методикой создания нового растрового изображения, бегло рассмотрим способ представления цветов в языке ActionScript.

Значения цвета пикселов

В языке ActionScript значения цвета пикселов, формирующих растровые изображения, сохраняются в виде 32-битных беззнаковых целых чисел, обеспечивая огром-

ный диапазон из 4 294 967 296 возможных значений цветов. Каждое конкретное значение цвета в объекте `BitmapData` концептуально состоит из четырех отдельных значений, которые представляют четыре различных компонента цвета — Alpha (Альфа) (то есть прозрачность), Red (Красный), Green (Зеленый) и Blue (Синий). Эти четыре компонента называются цветовыми *каналами*. Величина каждого канала в любом цвете варьируется от 0 до 255. Соответственно в двоичном виде каждый канал занимает 8 из 32 бит значения цвета, как показано далее:

- Alpha — биты 24–31 (самый старший байт);
- Red — биты 16–23;
- Green — биты 8–15;
- Blue — биты 0–7.

Чем выше значение канала Red, Green или Blue, тем больший вклад вносит каждый цвет в результирующий цвет. Если значения всех трех каналов RGB равны, будет получен оттенок серого цвета; если значения всех каналов равны 0, будет получен черный цвет; если значения всех каналов равны 255, будет получен белый цвет. Данный 32-битный формат цвета позволяет задать 16 777 216 возможных цветов, при этом каждый цвет может иметь отдельный уровень канала Alpha, находящийся в диапазоне от 0 (прозрачный) до 255 (непрозрачный).

Например, чистый красный цвет описывается следующими значениями каналов:

```
Alpha: 255, Red: 255, Green: 0, Blue: 0
```

Эти значения соответствуют следующим установкам битов в 32-битном беззнаковом целочисленном значении цвета (пробелы определяют границы четырех байтов):

```
11111111 11111111 00000000 00000000
```

В десятичном виде предыдущее целочисленное значение будет выглядеть следующим образом:

```
4294901760
```

Безусловно, когда значение цвета представлено в виде одного десятичного числа, уровни различных каналов в этом значении цвета неочевидны. В связи с этим значения цвета пикселов для удобочитаемости обычно записываются в шестнадцатеричном виде: `0xAARRGGBB`, где `AA`, `RR`, `GG` и `BB` — это двухразрядные шестнадцатеричные числа, представляющие каналы Alpha, Red, Green и Blue. Например, предыдущее значение для чисто красного цвета (A:255, R:255, G:0, B:0) в шестнадцатеричном виде записывается следующим образом:

```
0xFFFF0000  // Гораздо проще читать!
```

Для сравнения на рис. 26.2 представлено изображение, заимствованное из рис. 26.1, но на этот раз значения цвета трех выбранных пикселов разбиты на соответствующие цветовые каналы, записанные в шестнадцатеричном виде.

Пример перевода чисел в шестнадцатеричную систему счисления можно найти по адресу http://www.moock.org/asdg/technotes/basePrimer.

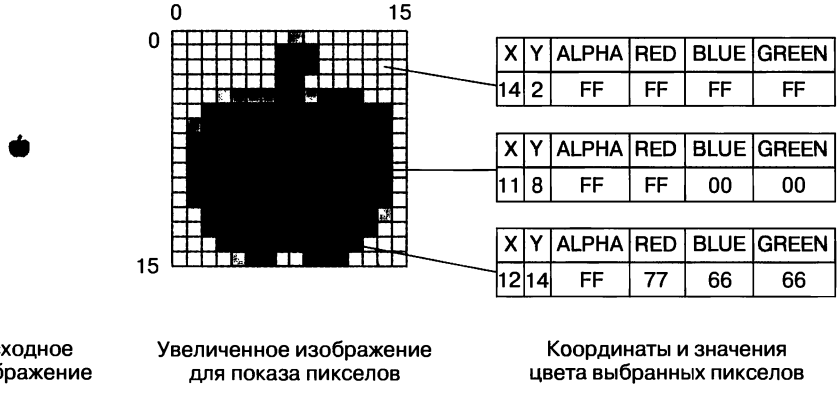

Рис. 26.2. Растровое изображение с шестнадцатеричными значениями цвета

Чтобы получить значение отдельного канала из 32-битного значения цвета, мы можем использовать совместно оператор сдвига вправо и оператор побитового И, как показано в следующем коде:

```
var colorValue:uint = 0xFFFFCC99;  // Выбранный цвет
var alpha:uint = (colorValue >> 24) & 0xFF;  // Выделяем канал Alpha
var red:uint = (colorValue >> 16) & 0xFF;    // Выделяем канал Red
var green:uint = (colorValue >> 8) & 0xFF;   // Выделяем канал Green
var blue:uint = colorValue & 0xFF;           // Выделяем канал Blue

trace(alpha, red, green, blue);  // Отображает: 255 255 204 153
```

Пример использования побитовых операций можно найти в технических записках по адресу http://www.moock.org/asdg/technotes/bitwise.

Хотя непосредственно изменить значение отдельного канала в существующем 32-битном значении цвета невозможно, мы можем достичь этого эффекта, используя комбинацию бинарных операций. Сначала мы очищаем байт нужного канала в существующем значении цвета; затем мы объединяем получившееся число с новым значением цветового канала. Эта методика продемонстрирована в следующем коде. При его выполнении создается число, которое является результатом вставки шестнадцатеричного значения AA в существующее значение цвета:

```
var colorValue:uint = 0xFFFFCC99;  // Выбранный цвет
// Очищаем байт канала Red в исходном значении цвета и присваиваем результат
// обратно переменной colorValue
colorValue &= 0xFF00FFFF;
// Объединяем значение переменной colorValue с новым значением канала Red
colorValue |= (0xAA<<16);
trace(colorValue.toString(16));  // Выводит: ffaacc99
```

Внимательно посмотрите на последнюю строку кода:

```
trace(colorValue.toString(16));  // Выводит: ffaacc99
```

Этот код генерирует строку в шестнадцатеричном формате для числового значения цвета, вызывая над этим значением метод `toString( )` и передавая в качестве

параметра `radix` значение `16`. Строка в шестнадцатеричном формате проще для чтения, чем ее числовой десятичный эквивалент. Тем не менее в качестве средства обеспечения удобочитаемости значения цвета метод `toString( )` неидеален — он опускает ведущие нули. Например, для числа

```
var n:uint = 0x0000CC99;
```

выражение `n.toString(16)` вернет значение `cc99`, опустив четыре ведущих нуля. Чтобы улучшить удобочитаемость значений цветов на этапе отладки, мы можем написать собственный код, наподобие представленного в листинге 26.1. В этом листинге представлен класс `Pixel`, предназначенный для работы со значениями цветов. Класс `Pixel` помещает бинарные операции в удобные методы, например `setRed( )` и `setAlpha( )`. Методы этого класса могут получать и устанавливать уровни отдельных цветовых каналов в значении цвета и генерировать строки, описывающие значения цвета с использованием различных удобочитаемых форматов. Описание класса `Pixel` доступно в Интернете по адресу http://www.moock.org/eas3/examples.

Листинг 26.1. Класс Pixel

```
package {
  public class Pixel {
    private var value:uint;  // Значение цвета пиксела

    public function Pixel (n1:uint, n2:int=0, n3:int=0, n4:int=0) {
      if (arguments.length == 1) {
        value = n1;
      } else {
        value = n1<<24 | n2<<16 | n3<<8 | n4;
      }
    }

    public function setAlpha (n:int):void {
      if (n < 0 || n > 255) {
        throw new RangeError("Supplied value must be in the range 0-255.");
      }
      value &= (0x00FFFFFF);
      value |= (n<<24);
    }

    public function setRed (n:int):void {
      if (n < 0 || n > 255) {
        throw new RangeError("Supplied value must be in the range 0-255.");
      }
      value &= (0xFF00FFFF);
      value |= (n<<16);
    }

    public function setGreen (n:int):void {
      if (n < 0 || n > 255) {
        throw new RangeError("Supplied value must be in the range 0-255.");
      }
```

```
    value &= (0xFFFF00FF);
    value |= (n<<8);
  }

  public function setBlue (n:int):void {
    if (n < 0 || n > 255) {
      throw new RangeError("Supplied value must be in the range 0-255.");
    }
    value &= (0xFFFFFF00);
    value |= (n);
  }

  public function getAlpha ( ):int {
    return (value >> 24) & 0xFF;
  }


  public function getRed ( ):int {
    return (value >> 16) & 0xFF;
  }

  public function getGreen ( ):int {
    return (value >> 8) & 0xFF;
  }

  public function getBlue ( ):int {
    return value & 0xFF;
  }

  public function toString ( ):String {
    return toStringARGB( );
  }

  public function toStringARGB (radix:int = 16):String {
    var s:String =
         "A:" + ((value >> 24)&0xFF).toString(radix).toUpperCase( )
      + " R:" + ((value >> 16)&0xFF).toString(radix).toUpperCase( )
      + " G:" + ((value >> 8)&0xFF).toString(radix).toUpperCase( )
      + " B:" + (value&0xFF).toString(radix).toUpperCase( );
    return s;
  }

  public function toStringAlpha (radix:int = 16):String {
    return ((value >> 24)&0xFF).toString(radix).toUpperCase( );
  }

  public function toStringRed (radix:int = 16):String {
    return ((value >> 16)&0xFF).toString(radix).toUpperCase( );
  }

  public function toStringGreen (radix:int = 16):String {
    return ((value >> 8)&0xFF).toString(radix).toUpperCase( );
  }
```

```
    public function toStringBlue (radix:int = 16):String {
      return (value&0xFF).toString(radix).toUpperCase( );
    }
  }
}

// Примеры использования:
var p:Pixel = new Pixel(0xFFFFCC99);  // Выбранный цвет
p.setRed(0xAA);
trace(p);                  // Выводит: A:FF R:AA G:CC B:99
trace(p.getRed( ));        // Выводит: 170
trace(p.toStringRed( ));   // Выводит: AA

var p2:Pixel = new Pixel(0x33,0x66,0x99,0xCC);
trace(p2.toStringARGB(10));  // Выводит: A:51 R:102 G:153 B:204
```

Создание нового растрового изображения

Для создания и отображения совершенно нового растрового изображения нужно выполнить такую последовательность действий.

1. Создать объект `BitmapData`.
2. Установить желаемые цвета пикселов в созданном объекте `BitmapData`.
3. Связать объект `BitmapData` с объектом `Bitmap`.
4. Добавить объект `Bitmap` в список отображения.

Попробуем применить эти шаги на практике!

Наша цель — отобразить квадрат синего цвета размером 10 × 10 пикселов по центру фонового квадрата зеленого цвета размером 20 × 20 пикселов. Сначала мы создадим объект `BitmapData`, используя следующий обобщенный код:

```
new BitmapData(ширина, высота, прозрачность, цветЗаливки)
```

Параметры *ширина* и *высота* обозначают размеры изображения в пикселах — максимальное значение, которое могут принимать данные параметры, равняется 2880. После создания объекта `BitmapData` изменить размеры изображения невозможно. Параметр *прозрачность* определяет, должно ли изображение поддерживать прозрачность отдельных пикселов (то есть может ли уровень канала Alpha значения цвета любого пиксела быть меньше 255). Если необходимости в поддержке прозрачности изображения нет, то параметру *прозрачность* должно быть установлено значение `false`, поскольку среда Flash отображает непрозрачные изображения быстрее, чем прозрачные. Наконец, параметр *цветЗаливки* задает значение цвета, которое изначально присваивается всем пикселам изображения.

Изображение, которое мы хотим создать, представляет собой квадрат размером 20 × 20 пикселов, для него не требуется прозрачности, и оно имеет фон зеленого цвета. Таким образом, чтобы создать наш объект `BitmapData`, мы используем следующий код:

```
// 0xFF00FF00 означает Alpha: 255, Red: 0, Green: 255, Blue: 0
var imgData:BitmapData = new BitmapData(20, 20, false, 0xFF00FF00);
```

Теперь нам необходимо установить синий цвет для пикселов квадратной области размером 10 × 10 пикселов. Класс `BitmapData` предоставляет несколько инструментов для установки цвета пикселов: `setPixel( )`, `setPixel32( )`, `setPixels( )`, `fillRect( )` и `floodFill( )`. Для наших целей отлично подходит метод `fillRect( )` — он присваивает пикселам заданной прямоугольной области указанный цвет. Задаваемый нами объект `Rectangle` имеет ширину и высоту, равную 10 пикселам, а его левый верхний угол находится в точке с координатами (5; 5). В результате все пикселы растрового изображения, формирующие прямоугольную область от точки (5; 5) до точки (14; 14) включительно, будут окрашены в синий цвет.

```
imgData.fillRect(new Rectangle(5, 5, 10, 10), 0xFF0000FF);
```

Мы завершили установку цвета пикселов в нашем объекте `BitmapData` и готовы связать его с объектом `Bitmap` для дальнейшего отображения на экране. Связать объект `BitmapData` с объектом `Bitmap` можно двумя способами: передать объект `BitmapData` в конструктор класса `Bitmap` или присвоить объект `BitmapData` переменной экземпляра `bitmapData` существующего объекта `Bitmap`. Следующий код демонстрирует обе методики:

```
// Передаем объект BitmapData в конструктор класса Bitmap
var bmp:Bitmap = new Bitmap(imgData);
// Присваиваем объект BitmapData переменной экземпляра bitmapData
var bmp:Bitmap = new Bitmap( );
bmp.bitmapData = imgData;
```

Как только объект `BitmapData` будет связан с объектом `Bitmap`, добавление данного объекта `Bitmap` в список отображения приведет к выводу изображения, описываемого объектом `BitmapData`, на экран:

```
// Выводим объект на экран
addChild(bmp);
```

Рассмотрим код, необходимый для создания и вывода на экран нового растрового изображения, содержащего квадрат синего цвета размером 10 × 10 пикселов, который размещается по центру фонового квадрата зеленого цвета размером 20 × 20 пикселов:

```
var imgData:BitmapData = new BitmapData(20, 20, false, 0xFF00FF00);
imgData.fillRect(new Rectangle(5, 5, 10, 10), 0xFF0000FF);
var bmp:Bitmap = new Bitmap(imgData);
addChild(bmp);
```

На рис. 26.3 показан результат выполнения предыдущего кода.

Рис. 26.3. Растровое изображение, созданное с нуля

Как уже отмечалось ранее, несколько различных объектов `Bitmap` могут одновременно отображать представления одного и того же объекта `BitmapData`. Например, следующий код использует наш объект `imgData` в качестве источника данных

для двух различных объектов Bitmap. Первый объект Bitmap представляет объект imgData без изменений, а второй объект Bitmap поворачивает и масштабирует исходное изображение.

```
var imgData:BitmapData = new BitmapData(20, 20, false, 0xFF00FF00);
imgData.fillRect(new Rectangle(5, 5, 10, 10), 0xFF0000FF);

var bmp1:Bitmap = new Bitmap(imgData);
addChild(bmp1);

var bmp2:Bitmap = new Bitmap(imgData);
bmp2.rotation = 45;
bmp2.x = 50;
bmp2.scaleX = 2;  // 200 %
bmp2.scaleY = 2;  // 200 %
addChild(bmp2);
```

Результаты выполнения этого кода показаны на рис. 26.4.

Рис. 26.4. Два растровых изображения с одним и тем же источником BitmapData

Обратите внимание, что преобразования, применяемые к объекту Bitmap, не оказывают никакого влияния на связанный с ним объект BitmapData. Непосредственно преобразовать (то есть повернуть, масштабировать или переместить) реальные пиксельные данные, хранящиеся в объекте BitmapData, невозможно. Тем не менее это можно сделать в процессе их копирования в новый объект BitmapData. Дополнительная информация по этому вопросу представлена далее в разд. «Копирование графики в объект BitmapData».

Загрузка внешнего растрового изображения

В предыдущем разделе мы узнали, как создавать новое растровое изображение. Теперь попробуем загрузить существующее изображение с диска. К форматам растровых изображений, которые могут быть загружены и отображены на экране, относятся JPEG, GIF и PNG.

JPEG-изображения, загружаемые извне, могут быть в прогрессивном или обычном формате. Анимированные GIF-изображения не воспроизводятся; отображается только их первый кадр.

Внешние растровые изображения могут быть загружены двумя способами: на этапе выполнения с помощью класса Loader или на этапе компиляции с помощью тега метаданных [Embed]. Для справки в листингах 26.2 и 26.3 приведены примеры кода, демонстрирующие обе методики; гораздо более глубокое рассмотрение данного вопроса представлено в гл. 28.

Код из листинга 26.2 демонстрирует, как на этапе выполнения загрузить растровое изображение с именем `photo.jpg`. Предполагается, что файл растрового изображения и SWF-файл, загружающий данный файл, находятся в одной папке.

Листинг 26.2. Загрузка растрового изображения на этапе выполнения

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.net.*;

  // Простой пример, демонстрирующий, как загружать изображение
  public class BitmapLoader extends Sprite {
    private var loader:Loader;  // загрузчик растрового изображения

    public function BitmapLoader( ) {
      // Создаем загрузчик
      loader = new Loader( );

      // Регистрируем приемник для получения уведомления об окончании
      // процесса загрузки и инициализации растрового изображения
      loader.contentLoaderInfo.addEventListener(Event.INIT,
                                                initListener);
      // Загружаем растровое изображение
      loader.load(new URLRequest("photo.jpg"));
    }

    // Вызывается, когда процесс загрузки и инициализации растрового
    // изображения будет завершен
    private function initListener (e:Event):void {
      // Добавляем загруженное растровое изображение в список отображения
      addChild(loader.content);

      // Получаем значение цвета для левого верхнего пиксела загруженного
      // растрового изображения
      trace(Bitmap(loader.content).bitmapData.getPixel(0, 0));
    }
  }
}
```

Стоит отметить, что после загрузки растрового изображения к его пиксельным данным можно обращаться через переменную экземпляра `bitmapData` класса `Bitmap`, как показано в следующем коде (обратите внимание на операцию приведения к типу данных `Bitmap`, которая необходима для компиляции кода в строгом режиме; дополнительные сведения можно найти в гл. 8):

```
Bitmap(loader.content).bitmapData
```

Код из листинга 26.3 демонстрирует, как на этапе компиляции встроить растровое изображение с именем `photo.jpg`. Предполагается, что файл класса, встраивающего это растровое изображение, и файл растрового изображения находятся в одной папке.

Тег метаданных [Embed], используемый в листинге 26.3, поддерживается приложением Flex Builder и консольным компилятором mxmlc, но не поддерживается приложением Flash CS3. В приложении Flash CS3 необходимо использовать библиотеку flex.swc, поддерживающую компилятор приложения Flex. Подробную информацию можно найти в разд. «Встраивание отображаемых элементов на этапе компиляции» гл. 28.

Листинг 26.3. Встраивание растрового изображения на этапе компиляции

```
package {
  import flash.display.*;
  import flash.events.*;
  import mx.core.BitmapAsset;

  public class BitmapEmbedder extends Sprite {
    // Встраиваем растровое изображение
    [Embed(source="photo.jpg")]
    private var Photo:Class;

    public function BitmapEmbedder ( ) {
      // Создаем экземпляр встроенного растрового изображения
      var photo:BitmapAsset = new Photo( );
      addChild(photo);
      trace(photo.bitmapData.getPixel(0, 0));
    }
  }
}
```

Как и в листинге 26.2, к пиксельным данным встроенного растрового изображения можно обращаться через переменную `bitmapData`, как показано в следующей строке кода (на этот раз операция приведения типов не требуется, поскольку тип данных объекта `photo` является потомком класса `Bitmap`):

```
photo.bitmapData
```

Анализ растрового изображения

Теперь, когда мы знаем, как создавать новые и загружать существующие растровые изображения, рассмотрим инструменты, предназначенные для анализа пикселов в существующем растровом изображении.

Чтобы получить полное 32-битное целочисленное значение цвета любого пиксела в растровом изображении, мы применяем метод экземпляра `getPixel32( )` класса `BitmapData`, который принимает следующий вид:

объектBitmapData`.getPixel32(`*x*`, `*y*`)`

Здесь *объектBitmapData* — экземпляр класса `BitmapData`, из которого будет получено значение цвета пиксела, а *x* и *y* — горизонтальное и вертикальное положения пиксела, для которого будет получено значение цвета. Например, следующий код создает растровое изображение, представляющее квадрат синего цвета, после чего отображает значение цвета левого верхнего пиксела созданного изображения (то есть пиксела с координатами (0; 0)):

```
var imgData:BitmapData = new BitmapData(20, 20, false, 0xFF0000FF);
trace(imgData.getPixel32(0, 0));  // Выводит: 4278190335
```

Значением цвета данного пиксела является большое число (4 278 190 335), поскольку значение канала Alpha равно 255, следовательно, все биты в самом старшем байте значения цвета равны 1:

```
11111111 00000000 00000000 11111111
```

В десятичном виде уровни отдельных каналов в значении цвета, возвращаемого методом `getPixel32( )`, не поддаются расшифровке, поэтому в отладочных целях для извлечения удобочитаемых значений каналов из числа, возвращаемого методом `getPixel32( )`, должен использоваться код наподобие представленного ранее в классе `Pixel`:

```
// Выводит: A:FF R:0 G:0 B:FF
trace(new Pixel(imgData.getPixel32(0, 0)));
```

Стоит отметить, что значение канала Alpha для пикселов в непрозрачных растровых изображениях всегда равно 255, даже когда цвету пиксела присваивается другое значение канала Alpha. Например, следующий код создает квадрат синего цвета, представляющий непрозрачное растровое изображение, и присваивает каналу Alpha всех его пикселов значение `0x33`. Поскольку растровое изображение является непрозрачным, операция присваивания значения каналу Alpha попросту игнорируется:

```
var imgData:BitmapData = new BitmapData(20, 20, false, 0x330000FF);
trace(imgData.getPixel32(0, 0));  // Выводит: 4278190335
                                  // (значение Alpha равно 0xFF, а не 0x33)
```

Для пикселов значение канала Alpha может быть установлено только в прозрачных растровых изображениях (то есть в растровых изображениях, при создании которых в качестве параметра `transparent` конструктора класса `BitmapData` было передано значение `true`). Например, следующий код снова создает растровое изображение, представляющее квадрат синего цвета, но на этот раз с включенной прозрачностью. Поскольку растровое изображение является прозрачным, операция присваивания значения `0x33` каналу Alpha завершается успешно.

```
var imgData:BitmapData = new BitmapData(20, 20, true, 0x330000FF);
trace(imgData.getPixel32(0, 0));  // Выводит: 855638271
                                  // (значение Alpha равно 0x33)
```

Метод getPixel32( ) в сравнении с методом getPixel( )

Язык ActionScript предоставляет удобный способ для получения значения цвета пиксела без информации о его канале Alpha — метод экземпляра `getPixel( )` класса `BitmapData`. Этот метод принимает такой же вид, как и метод `getPixel32( )`, и также возвращает 32-битное целочисленное значение цвета. Однако, в отличие от метода `getPixel32( )`, он присваивает битам канала Alpha в возвращаемом целом числе значение 0. Иными словами, вызов метода `getPixel( )` эквивалентен следующему выражению:

объектBitmapData`.getPixel32( ) & 0x00FFFFFF`

Реальное значение канала Alpha для указанного пиксела в растровом изображении остается неизменным; изменяется только возвращаемое число. Например, вспомним растровое изображение, представляющее квадрат синего цвета, из предыдущего раздела:

```
var imgData:BitmapData = new BitmapData(20, 20, false, 0xFF0000FF);
```

Если мы попытаемся получить значение цвета для левого верхнего пиксела в этом растровом изображении с помощью метода `getPixel( )`, будет возвращено значение `255`, поскольку битам канала Alpha было установлено значение 0 (сравните значение `255` со значением `4 278 190 335`, которое было возвращено ранее методом `getPixel32( )`):

```
trace(imgData.getPixel(0, 0)); // Выводит: 255
```

Метод `getPixel( )` должен использоваться только для получения комбинированного значения каналов Red, Green и Blue в виде одного числа. Если получаемое значение цвета будет использовано для дальнейшей обработки одного или нескольких каналов по отдельности, используйте метод `getPixel32( )`. Он подходит для большинства ситуаций, связанных с обработкой цвета.

Метод getPixel32( ) возвращает 32-битное целое число, представляющее полное четырехканальное значение цвета для некоторого пиксела. Метод getPixel( ) возвращает 32-битное целое число, которое содержит значения каналов Red, Green и Blue для некоторого пиксела и значение канала Alpha, равное 0.

Влияние прозрачности на получение значения цвета

Из-за особенностей внутренней архитектуры механизма визуализации среды выполнения Flash нельзя гарантировать получение корректных значений цвета пикселов в прозрачных изображениях с помощью методов `getPixel32( )`, `getPixel( )` или любых других способов. Для повышения производительности процесса визуализации, когда значение цвета пиксела сохраняется в объекте `BitmapData`, среда Flash преобразует это значение во внутренний формат, называемый *предумноженным значением цвета*. Оно объединяет значение канала Alpha со значениями каналов Red, Green и Blue этого цвета. Например, если значением канала Alpha исходного цвета является 50 % от 255, то предумноженное значение цвета будет хранить 50 % от 255 для канала Alpha, 50 % от исходного значения канала Red, 50 % от исходного значения канала Green и 50 % от исходного значения канала Blue. В результате исходные значения, присвоенные каналам Red, Green и Blue, будут потеряны.

При получении значений цвета пикселов из прозрачного изображения среда Flash осуществляет их автоматическое преобразование из предумноженного формата в стандартный *(неумноженный)* формат ARGB, который мы использовали в данной главе, приводя к потере точности. Во многих случаях преобразованное, неумноженное значение цвета не совпадает с исходным значением цвета, присвоенным пикселу. Например, следующий код создает новый объект `BitmapData`, в котором

каждый пиксел имеет чистый белый цвет и является полностью прозрачным (то есть значение канала Alpha равно 0):

```
var imgData:BitmapData = new BitmapData(20, 20, true, 0x00FFFFFF);
```

Если получить значение цвета для любого пиксела из предыдущего растрового изображения, результатом будет являться значение 0 (то есть значения всех четырех каналов равны 0):

```
trace(imgData.getPixel32(0, 0));  // Выводит: 0
```

Исходные значения для каналов Red, Green и Blue, которые были равны 255, оказались потеряны.

Таким образом, если программа желает сохранить и в дальнейшем использовать значения цвета прозрачных пикселов без потери данных, она должна сохранить эти значения в объекте `ByteArray`. Вы можете руководствоваться общим правилом: значения цвета прозрачных пикселов, сохраненные в растровом изображении, должны считаться недоступными.

В отличие от этого, значения цвета непрозрачных пикселов могут быть получены в любой момент без риска потерять данные:

```
// Получение значения цвета пиксела из непрозрачного изображения
var imgData:BitmapData = new BitmapData(20, 20, false, 0xFFFFFFFF);
trace(imgData.getPixel32(0, 0));  // Выводит: 4294967295
                                  // (исходные данные были сохранены)
```

Как показано в следующем коде, значение цвета любого пиксела, уровень канала Alpha которого имеет значение 255, остается неизменным между последовательными операциями присваивания и получения значения цвета, даже если пиксел сохраняется в прозрачном растровом изображении.

```
// Получаем значение цвета пиксела, уровень канала Alpha которого
// установлен в 255, из прозрачного изображения
var imgData:BitmapData = new BitmapData(20, 20, true, 0xFFFFFFFF);
trace(imgData.getPixel32(0, 0));  // Выводит: 4294967295
                                  // (исходные данные были сохранены)
```

Класс ColorPicker: пример использования метода getPixel32( )

Теперь, когда мы понимаем, как получить значение цвета пиксела, применим наши знания в реальной ситуации. Предположим, что мы разрабатываем интернет-приложение для создания приглашений на вечеринки. Пользователи приложения сначала выбирают фотографию, которая будет помещена на приглашение, а затем задают подходящий цвет для текста приглашения. Чтобы пользователь мог поэкспериментировать с различными цветами, приложение предоставляет специальную форму для выбора цвета. Когда пользователь перемещает указатель мыши над выбранным изображением, цвет текста приглашения автоматически изменяется в соответствии с цветом пиксела, над которым в данный момент находится указатель мыши. В листинге 26.4 продемонстрирован код для палитры выбора цвета с тестовым изображением `sunset.jpg`. Изучите комментарии, чтобы понять, как происходит получение значения цвета под указателем мыши.

Листинг 26.4. Палитра выбора цвета на основе изображения

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.text.*;
  import flash.net.*;

  // Устанавливает цвет объекта TextField в соответствии с цветом
  // выбранного в изображении пиксела.
  public class ColorPicker extends Sprite {
    private var img:Bitmap;           // Объект Bitmap
    private var imgContainer:Sprite;  // Контейнер для объекта Bitmap
    private var t:TextField;          // Раскрашиваемый объект TextField

    // Метод-конструктор
    public function ColorPicker( ) {
      // Создаем объект TextField и добавляем его в иерархию отображения
      // объекта ColorPicker
      t = new TextField( );
      t.text = "Please come to my party...";
      t.autoSize = TextFieldAutoSize.LEFT;
      addChild(t);

      // Загружаем изображение
      var loader:Loader = new Loader( );
      loader.contentLoaderInfo.addEventListener(Event.INIT,
                                                initListener);
      loader.load(new URLRequest("sunset.jpg"));
    }

    // Вызывается, когда инициализация изображения завершена
    private function initListener (e:Event):void {
      // Получаем ссылку на загруженный объект Bitmap
      img = e.target.content;
      // Помещаем загруженное растровое изображение в объект Sprite, чтобы
      // мы могли реагировать на взаимодействия с мышью
      imgContainer = new Sprite( );
      imgContainer.addChild(img);
      // Добавляем объект Sprite в иерархию отображения объекта ColorPicker
      addChild(imgContainer);
      imgContainer.y = 30;
      // Регистрируем приемник для получения уведомлений о перемещении мыши
      imgContainer.addEventListener(MouseEvent.MOUSE_MOVE,
      mouseMoveListener);
    }

    // Вызывается, когда происходит перемещение мыши над объектом Sprite,
    // содержащим изображение
    private function mouseMoveListener (e:MouseEvent):void {
      // Устанавливаем цвет текста в соответствии с цветом пиксела,
      // находящегося в данный момент под указателем мыши
```

```
            t.textColor = img.bitmapData.getPixel32(e.localX, e.localY);
        }
    }
}
```

Получение цвета области пикселов

Методы экземпляра `getPixel32( )` и `getPixel( )` класса `BitmapData` применяются для получения значения цвета отдельного пиксела. В отличие от этого, метод экземпляра `getPixels( )` класса `BitmapData` используется для получения значений цвета целой прямоугольной области пикселов. Метод `getPixels( )` может быть использован в любом из следующих сценариев:

- при передаче области растрового изображения между модулями программы;
- при использовании собственного алгоритма для обработки фрагмента растрового изображения;
- при отправке фрагмента или всего растрового изображения на сервер в необработанном бинарном формате.

Метод `getPixels( )` принимает следующий обобщенный вид:

объектBitmapData`.getPixels(`*область*`)`

Здесь *объектBitmapData* — объект `BitmapData`, из которого будут возвращаться значения цвета пикселов, а *область* — объект `flash.geom.Rectangle`, описывающий область возвращаемых пикселов. Метод `getPixels( )` возвращает объект `ByteArray`, содержащий 32-битные целочисленные значения цвета. Объект `ByteArray` — это список значений цвета для пикселов в указанной прямоугольной области, обход которых происходит слева направо и сверху вниз. Например, рассмотрим следующую диаграмму растрового изображения размером 4 × 4, пикселы которого для простоты обозначены буквами от A до P:

```
A B C D
E F G H
I J K L
M N O P
```

Не забывая, что пиксел левого верхнего угла растрового изображения находится в точке с координатой (0; 0), если мы воспользуемся методом `getPixels( )` для получения значений цвета прямоугольной области пикселов от точки (2; 1) до точки (3; 3), возвращаемый объект `ByteArray` будет содержать следующие пикселы в таком порядке:

```
G, H, K, L, O, P
```

Стоит отметить, что объект `ByteArray` представляет собой одномерный список и не содержит никакой информации о размерах и позиции прямоугольной области, из которой были получены данные пикселы. Таким образом, чтобы восстановить растровое изображение из пикселов, хранящихся в объекте `ByteArray`, в том порядке, в котором они находились ранее, мы должны иметь свободный доступ к ширине, высоте и позиции исходного прямоугольника. Информация об исходном прямоугольнике может быть присвоена переменной или даже добавлена в сам объект `ByteArray`.

Чтобы попрактиковаться в использовании метода `getPixels( )`, скопируем прямоугольную область из одного растрового изображения в другое изображение. Сначала мы создадим два объекта `BitmapData`. Первый объект представляет квадрат синего цвета размером 20 × 20 пикселов, а другой — квадрат зеленого цвета размером 30 × 30 пикселов:

```
var blueSquare:BitmapData = new BitmapData(20, 20, false, 0xFF0000FF);
var greenSquare:BitmapData = new BitmapData(30, 30, false, 0xFF00FF00);
```

Затем мы определяем прямоугольную область пикселов, которую хотим получить из квадрата зеленого цвета. Левый верхний угол прямоугольника находится в точке с координатой (5; 5), а его ширина и высота равна 10 пикселам.

```
var rectRegion:Rectangle = new Rectangle(5, 5, 10, 10);
```

Теперь мы получаем пикселы зеленого цвета:

```
var greenPixels:ByteArray = greenSquare.getPixels(rectRegion);
```

Чтобы перенести пикселы зеленого цвета на квадрат синего цвета, мы используем метод экземпляра `setPixels( )` класса `BitmapData`. Однако перед вызовом метода `setPixels( )` мы должны установить указатель файла объекта `ByteArray` в значение `0`, чтобы метод `setPixels( )` начал чтение значений цвета пикселов с начала списка:

```
greenPixels.position = 0;
```

Теперь мы можем прочитать пикселы из объекта `ByteArray greenPixels` и сохранить их в объекте `BitmapData blueSquare`:

```
blueSquare.setPixels(rectRegion, greenPixels);
```

Чтобы убедиться, что все работает так, как ожидалось, мы отображаем два растровых изображения на экране:

```
var blueBmp:Bitmap = new Bitmap(blueSquare);
var greenBmp:Bitmap = new Bitmap(greenSquare);
addChild(blueBmp);
addChild(greenBmp);
greenBmp.x = 40;
```

На рис. 26.5 показаны результаты выполнения предыдущего кода.

Рис. 26.5. Пикселы, скопированные из объекта ByteArray

Если при копировании пикселов между двумя растровыми изображениями размеры копируемого прямоугольника и целевого прямоугольника совпадают (как в предыдущем примере), мы можем использовать удобный метод экземпляра `copyPixels( )` класса `BitmapData` вместо комбинации методов `getPixels( )` и `setPixels( )`. К другим внутренним методам экземпляра класса `BitmapData`, предоставляющим удобный доступ к типичным операциям копирования, относятся: `copyChannel( )`, `clone( )`, `merge( )` и `draw( )`. Дополнительную информацию можно найти далее, в разд. «Копирование графики в объект BitmapData» этой главы.

Другие инструменты анализа

В этом разделе мы узнали, как анализировать пикселы объекта `BitmapData`, используя методы `getPixel32( )`, `getPixel( )` и `getPixels( )`. Кроме того класс `BitmapData` предоставляет несколько других, более специализированных инструментов для анализа пикселов:

- `compare( )` — проверяет, есть ли отличие между пикселами двух растровых изображений;
- `getColorBoundsRect( )` — определяет, какая область растрового изображения содержит указанный цвет;
- `hitTest( )` — определяет, перекрывают ли пикселы растрового изображения некоторую точку, прямоугольник или другое растровое изображение.

Подробную информацию о перечисленных методах можно найти в описании класса `BitmapData` в справочнике по языку ActionScript корпорации Adobe.

Внесение изменений в растровое изображение

Основные инструменты для присваивания новых цветов пикселам существующего растрового изображения являются точным отражением инструментов, предназначенных для анализа растрового изображения. К ним относятся методы `setPixel32( )`, `setPixel( )` и `setPixels( )`. Метод `setPixel32( )` присваивает новое четырехканальное значение цвета пикселу в виде 32-битного целого числа. Он принимает следующий вид:

объектBitmapData`.setPixel32(`*x*`, `*y*`, `*цвет*`)`

Здесь *объектBitmapData* — экземпляр класса `BitmapData`, содержащий пиксел, значение цвета которого будет изменяться; *x* и *y* — горизонтальная и вертикальная позиции данного пиксела; *цвет* — новое значение цвета, присваиваемое пикселу. Например, следующий код создает растровое изображение, представляющее квадрат синего цвета, и затем присваивает его левому верхнему пикселу значение белого цвета:

```
var imgData:BitmapData = new BitmapData(20, 20, false, 0xFF0000FF);
imgData.setPixel32(0, 0, 0xFFFFFFFF);
```

В отличие от этого, метод `setPixel( )`, который принимает такой же общий вид, как и метод `setPixel32( )`, устанавливает только значения каналов Red, Green и Blue цвета пиксела, не изменяя исходное значение канала Alpha. Например, следующий код создает полупрозрачное растровое изображение, представляющее квадрат синего цвета, и затем присваивает его левому верхнему пикселу значение белого цвета. Поскольку вместо метода `setPixel32( )` используется `setPixel( )`, левый верхний пиксел сохраняет свое исходное значение канала Alpha (`0x66`):

```
var imgData:BitmapData = new BitmapData(20, 20, true, 0x660000FF);
imgData.setPixel(0, 0, 0xFFFFFF);
```

После завершения операции `setPixel( )` значением цвета левого верхнего пиксела будет являться число `0x66FFFFFF`.

Любое значение канала Alpha, указываемое в числе, которое передается в метод `setPixel( )`, будет проигнорировано. Например, в следующем коде мы присваиваем значение цвета пиксела, используя число, в котором для канала Alpha указано значение `CC`. Несмотря на это, после завершения операции значением цвета левого верхнего пиксела по-прежнему будет являться число `0x66FFFFFF`:

```
imgData.setPixel(0, 0, 0xCCFFFFFF);
```

Повышение производительности с помощью метода BitmapData.lock( )

По умолчанию, всякий раз, когда над некоторым объектом `BitmapData` вызывается метод `setPixel32( )` или `setPixel( )`, экземпляры класса `Bitmap`, ссылающиеся на этот объект, получают уведомление об изменении данных. Когда методы `setPixel32( )` или `setPixel( )` вызываются друг за другом внутри одного цикла кадра (например, когда каждому пикселу в растровом изображении присваивается значение цвета), подобные уведомления могут привести к снижению производительности. Для повышения производительности мы можем использовать метод экземпляра `lock( )` класса `BitmapData`.

Вызов метода `lock( )` над объектом `BitmapData` *запрещает* среде выполнения Flash уведомлять зависимые объекты `Bitmap` при вызове методов `setPixel32( )` или `setPixel( )`. Таким образом, если вы собираетесь использовать методы `setPixel32( )` или `setPixel( )` друг за другом, всегда вызывайте метод `lock( )`. После его вызова присвойте все желаемые значения цвета пикселов; затем вызовите метод экземпляра `unlock( )` класса `BitmapData`. Метод `unlock( )` позволяет среде Flash при необходимости уведомить все зависимые объекты `Bitmap`.

Данный подход продемонстрирован в листинге 26.5. В этом коде используется цикл для присваивания случайного цвета каждому пикселу в объекте `BitmapData` размером 500 × 500 пикселов. Обратите внимание на вызов метода `lock( )` перед циклом и вызов метода `unlock( )` после цикла, выделенные полужирным шрифтом.

Листинг 26.5. Использование метода BitmapData.lock( ) для повышения производительности

```
// Создаем растровое изображение
var imgData:BitmapData = new BitmapData(500, 500, true, 0x00000000);
var bmp:Bitmap = new Bitmap(imgData);

// Вызываем метод lock( )
imgData.lock( );

// Устаналиваем значения цвета пикселов
var color:uint;
```

```
for (var i:int = 0; i < imgData.height ; i++) {
  for (var j:int = 0; j < imgData.width; j++) {
    color = Math.floor(Math.random( )*0xFFFFFFFF);
    imgData.setPixel32(j, i, color);
  }
}

// Вызываем метод unlock( )
imgData.unlock( );
```

При тестировании кода из листинга 26.5 в рабочей версии приложения Flash Player на компьютере с процессором Pentium 4 2,6 ГГц одна итерация цикла занимает приблизительно 100 мс. Без использования метода `lock( )` одна итерация занимает примерно 125 мс. Иными словами, при использовании метода `lock( )` код выполняется приблизительно на 20 % быстрее.

При измерении производительности среды Flash всегда выполняйте тесты в рабочей, а не в отладочной версии. Производительность в рабочей версии приложения зачастую оказывается в два раза выше, чем в отладочной.

Класс ScribbleAS3: пример использования метода setPixel32( )

Присваивание цвета пикселу в растровом изображении имеет множество практических применений: от создания собственных эффектов до коррекции фотографий или генерации динамического интерфейса. Рассмотрим всего одно практическое применение метода `setPixel32( )` — простую программу для рисования. В листинге 26.6 представлена адаптация на языке ActionScript 3.0 программы Scribble. Выполнение этого кода приводит к созданию пустого растрового изображения, на котором пользователь рисует линии с помощью мыши. Когда пользователь перемещает мышь, удерживая нажатой левую кнопку, на пустом растровом изображении рисуется пиксел черного цвета.

Листинг 26.6. Очень простая программа рисования ScribbleAS3

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.ui.*;
  import flash.geom.*;

  // Простое приложение для рисования. Рисует одну точку на объекте
  // BitmapData  всякий раз, когда возникает событие MouseEvent.MOUSE_MOVE
  // при нажатой левой кнопке мыши.
  public class ScribbleAS3 extends Sprite {
    // Растровое изображение, отображаемое на экране
    private var canvas:Bitmap;
    // Содержит растровое изображение, обеспечивая интерактивность
    private var canvasContainer:Sprite;
```

```
// Линия вокруг растрового изображения
private var border:Shape;
// Сообщает о том, нажата ли кнопка мыши в настоящий момент
private var isDrawing:Boolean = false;

// Конструктор
public function ScribbleAS3 ( ) {
  createCanvas( );
  registerForInputEvents( );

  // Предотвращаем изменение размеров окна приложения
  stage.scaleMode = StageScaleMode.NO_SCALE;
}

// Создает пустое растровое изображение, на котором будем рисовать
private function createCanvas (width:int = 200, height:int = 200):void {
  // Определяем объект BitmapData, который будет хранить пиксельные
  // данные для рисунка пользователя
  var canvasData:BitmapData = new BitmapData(width, height,
                                             false, 0xFFFFFFFF);

  // Создаем новый отображаемый объект Bitmap, используемый
  // для отображения объекта canvasData
  canvas = new Bitmap(canvasData);

  // Создаем объект Sprite, который будет содержать объект Bitmap. Класс
  // Bitmap не поддерживает события ввода; следовательно, помещаем его
  // в объект Sprite, чтобы пользователь мог взаимодействовать с этим
  // объектом.
  canvasContainer = new Sprite( );
  // Добавляем растровое изображение bitmap в экземпляр canvasContainer
  // класса Sprite
  canvasContainer.addChild(canvas);

  // Добавляем экземпляр canvasContainer класса Sprite (и содержащийся
  // в нем объект Bitmap) в иерархию отображения данного объекта
  addChild(canvasContainer);

  // Создаем границу вокруг области рисования.
  border = new Shape( );
  border.graphics.lineStyle(1, 0xFF000000);
  border.graphics.drawRect(0, 0, width, height);
  addChild(border);
}

// Регистрирует приемники для необходимых событий мыши и клавиатуры
private function registerForInputEvents ( ):void {
  // Регистрируем приемники для событий нажатия кнопки мыши
  // и перемещения мыши от объекта canvasContainer
  canvasContainer.addEventListener(MouseEvent.MOUSE_DOWN,
                                   mouseDownListener);
```

```
    canvasContainer.addEventListener(MouseEvent.MOUSE_MOVE,
                                     mouseMoveListener);

    // Регистрируем приемники для событий отпускания кнопки мыши и нажатия
    // клавиши от объекта Stage (то есть для глобальных событий).
    // Используем объект Stage, поскольку событие отпускания кнопки мыши
    // должно всегда завершать рисование, даже если указатель мыши
    // не находится над областью рисования. Подобным образом нажатие
    // пробела должно всегда приводить к стиранию рисунка, даже когда
    // объект canvasContainer не имеет фокуса.
    stage.addEventListener(MouseEvent.MOUSE_UP, mouseUpListener);
    stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
  }

  // Устанавливает цвет указанного пиксела
  public function drawPoint (x:int, y:int, color:uint = 0xFF000000):void {
    canvas.bitmapData.setPixel32(x, y, color);
  }

  // Отвечает на события MouseEvent.MOUSE_DOWN
  private function mouseDownListener (e:MouseEvent):void {
    // Устанавливаем флажок, указывающий на то, что основная кнопка мыши
    // в настоящий момент нажата
    isDrawing = true;
    // Рисуем точку в позиции, где произошел щелчок кнопкой мыши.
    drawPoint(e.localX, e.localY);
  }


  // Отвечает на события MouseEvent.MOUSE_MOVE
  private function mouseMoveListener (e:MouseEvent):void {
    // Рисуем точку, когда мышь перемещается над областью рисования
    // при нажатой левой кнопке мыши
    if (isDrawing) {
      // Используем переменные localX и localY, чтобы получить позицию
      // указателя относительно объекта canvasContainer.
      drawPoint(e.localX, e.localY);

      // Обновляем экран сразу после завершения выполнения
      // данной функции-приемника события
      e.updateAfterEvent( );
    }
  }

  // Отвечает на события MouseEvent.MOUSE_UP
  private function mouseUpListener (e:MouseEvent):void {
    // Устанавливаем флажок, указывающий на то, что в настоящий момент
    // основная кнопка мыши отпущена
    isDrawing = false;
  }

  // Отвечает на события KeyboardEvent.KEY_DOWN
  private function keyDownListener (e:KeyboardEvent):void {
```

```
      // Стираем рисунок, когда пользователь нажимает клавишу Пробел. Чтобы
      // очистить рисунок, мы присваиваем всем пикселам значение белого
      // цвета.
      if (e.charCode == Keyboard.SPACE) {
        canvas.bitmapData.fillRect(new Rectangle(0, 0,
                                   canvas.width,
                                   canvas.height),
                                   0xFFFFFFFF);
      }
    }
  }
}
```

Присваивание цвета области пикселов

Методы `setPixel32( )` и `setPixel( )` используются для присваивания значения цвета отдельному пикселу. В отличие от этого, метод экземпляра `setPixels( )` класса `BitmapData` применяется для присваивания значений цвета целой прямоугольной области пикселов.

Метод `setPixels( )` имеет следующий обобщенный вид:

объектBitmapData`.setPixels(`*область*`,` *пикселыByteArray*`)`

Здесь *объектBitmapData* — объект `BitmapData`, пикселам которого присваиваются значения цвета, *область* — объект `flash.geom.Rectangle`, описывающий область пикселов, которым будет присвоен цвет, а *пикселыByteArray* — объект `ByteArray`, содержащий беззнаковые 32-битные целые числа, определяющие присваиваемые значения цвета.

Метод `setPixels( )` заполняет указанную прямоугольную область в направлении слева направо и сверху вниз, начиная со значения цвета объекта *пикселыByteArray*, находящегося в текущей позиции указателя файла (то есть в позиции *пикселыByteArray*`.position`).

Например, рассмотрим следующую диаграмму растрового изображения размером 4 × 4, пикселы которого для простоты обозначены буквами от A до P:

```
A B C D
E F G H
I J K L
M N O P
```

Теперь рассмотрим следующую диаграмму массива байт, содержащего шесть 32-битных беззнаковых целочисленных значений цвета, обозначенных символами от C1 до C6:

```
C1 C2 C3 C4 C5 C6
```

Не забывайте, что пиксел левого верхнего угла растрового изображения находится в точке с координатой (0; 0). Если мы воспользуемся методом `setPixels( )` для заполнения прямоугольной области пикселов от точки (1; 0) до точки (3; 1) с помощью предыдущего массива байт, растровое изображение будет выглядеть следующим образом:

```
A C1 C2 C3
E C4 C5 C6
I  J  K  L
M  N  O  P
```

Попробуем проделать то же самое в коде. Сначала создадим квадрат красного цвета размером 4 × 4 пиксела:

```
var imgData:BitmapData = new BitmapData(4, 4, false, 0xFFFF0000);
```

Теперь мы создадим массив байт, который содержит шесть значений цвета — все они обозначают зеленый цвет. Для демонстрационных целей мы создадим массив байт вручную, однако обычно он формируется программным путем, возможно, с помощью вызова метода `getPixels( )` или в результате выполнения пользовательского алгоритма, возвращающего значения цвета. Массив будет выглядеть следующим образом:

```
var byteArray:ByteArray = new ByteArray( );
byteArray.writeUnsignedInt(0xFF00FF00);
byteArray.writeUnsignedInt(0xFF00FF00);
byteArray.writeUnsignedInt(0xFF00FF00);
byteArray.writeUnsignedInt(0xFF00FF00);
byteArray.writeUnsignedInt(0xFF00FF00);
byteArray.writeUnsignedInt(0xFF00FF00);
```

Далее мы устанавливаем позицию, с которой метод `setPixels( )` должен начать чтение значений цвета из массива байт. Мы хотим, чтобы метод `setPixels( )` начал чтение с самого начала массива байт, поэтому присваиваем переменной экземпляра `position` класса `ByteArray` значение 0:

```
byteArray.position = 0;
```

Наконец, заполняем прямоугольную область в растровом изображении цветами из массива байт:

```
imgData.setPixels(new Rectangle(1,0,3,2), byteArray);
```

Обратите внимание, что позиция и размеры объекта Rectangle, передаваемого в метод setPixels( ), определяются с помощью координаты левого верхнего угла и ширины/высоты прямоугольника, а не координат левого верхнего угла и правого нижнего угла.

Стоит отметить, что, если данные в объекте *пикселыByteArray* закончатся до того, как будет заполнена указанная прямоугольная область, среда выполнения Flash сгенерирует исключение `EOFError`. Например, если мы увеличим размер предыдущей прямоугольной области с 3 × 2 пиксела (6 пикселов) до 3 × 3 пиксела (9 пикселов) следующим образом:

```
imgData.setPixels(new Rectangle(1,0,3,3), byteArray);
```

произойдет следующая ошибка:

```
Error: Error #2030: End of file was encountered.
```

На русском языке она будет звучать так: Ошибка #2030: достигнут конец файла.

Кроме того, подобная ошибка может возникнуть, если после создания объекта `ByteArray` мы забудем установить его позицию в 0 (что является гораздо более

распространенной ошибкой в программировании, чем указание неправильных размеров прямоугольника или представление недостаточного количества значений цвета).

Перед вызовом метода setPixels( ) не забывайте устанавливать позицию указываемого входного массива байт.

Метод `setPixels( )` обычно применяется для создания растрового изображения на основе сериализованных бинарных данных, полученных из некоторого внешнего источника, например сервера или совместно используемого локального объекта.

Другие инструменты изменения изображений

В этом разделе мы узнали, как можно изменять пикселы в объекте `BitmapData` с помощью методов `setPixel32( )`, `setPixel( )` и `setPixels( )`. Класс `BitmapData` также предоставляет несколько других, более специализированных инструментов для работы с пикселами:

- `fillRect( )` — присваивает заданный цвет пикселам из прямоугольной области;
- `floodFill( )` — присваивает заданный цвет всем пикселам, окружающим некоторый пиксел `p`, цвет которых соответствует цвету данного пиксела (подобно инструменту заливки, который присутствует во многих программах для работы с графикой);
- `scroll( )` — изменяет позицию всех пикселов растрового изображения на заданную величину по горизонтали и вертикали.

Подробное описание перечисленных методов можно найти в описании класса `BitmapData` в справочнике по языку ActionScript корпорации Adobe.

Класс `BitmapData` также поддерживает различные фильтры, эффекты и операции копирования, которые могут быть использованы для управления пикселами растрового изображения. Дополнительную информацию можно получить далее, в разд. «Копирование графики в объект BitmapData» и «Применение фильтров и эффектов» этой главы.

Изменение размеров растрового изображения

Когда изменяются размеры объекта `Bitmap`, ссылающегося на объект `BitmapData`, с помощью переменных `scaleX` и `scaleY` или `width` и `height`, размеры изображения на экране меняются, однако нижележащий объект `BitmapData` остается неизменным. Чтобы изменить размер нижележащего объекта `BitmapData` на самом деле, мы должны произвести его повторную выборку с помощью метода экземпляра `draw( )` класса `BitmapData` (*повторная выборка* означает изменение числа пикселов в изображении). Общая методика выглядит следующим образом.

1. Получить ссылку на исходный объект `BitmapData`.
2. Нарисовать масштабированную версию исходного объекта `BitmapData` в новом объекте `BitmapData`.

3. Наконец, связать исходный объект `Bitmap` с новым, масштабированным объектом `BitmapData`.

Перечисленные шаги продемонстрированы в листинге 26.7.

Листинг 26.7. Повторная выборка растрового изображения

```
// Получаем временную ссылку на исходный объект BitmapData
var originalBitmapData:BitmapData = originalBitmap.bitmapData;

// Устанавливаем величину, которая будет определять коэффициент
// масштабирования растрового изображения
var scaleFactor:Number = .5;

// Вычисляем новые размеры масштабированного растрового изображения
var newWidth:int = originalBitmapData.width * scaleFactor;
var newHeight:int = originalBitmapData.height * scaleFactor;

// Создаем новый объект BitmapData, размеры которого позволят уместить
// масштабированное растровое изображение
var scaledBitmapData:BitmapData = new BitmapData(newWidth, newHeight,
                                        originalBitmapData.transparent);

// Создаем матрицу преобразований, с помощью которой будет происходить
// масштабирование растрового изображения
var scaleMatrix:Matrix = new Matrix( );
matrix.scale(scaleFactor, scaleFactor);

// Переносим масштабированное растровое изображение
// в новый объект BitmapData
scaledBitmapData.draw(originalBitmapData, matrix);

// Заменяем исходный объект BitmapData новым масштабированным объектом BitmapData
originalBitmap.bitmapData = scaledBitmapData;
```

В следующем разделе мы узнаем более подробно о методе `draw( )`.

Копирование графики в объект BitmapData

Значения цвета пикселов могут быть скопированы в объект `BitmapData` из двух источников: другого объекта `BitmapData` или любого экземпляра `DisplayObject`.

Чтобы скопировать любой экземпляр класса `DisplayObject` в объект `BitmapData`, мы применяем метод `draw( )`, который копирует значения цвета из объекта-источника в объект-получатель `BitmapData`. В процессе копирования пикселы, сохраняемые в объекте `BitmapData`, могут быть преобразованы, смешаны или сглажены.

Чтобы скопировать значения цвета из другого объекта `BitmapData`, можно использовать либо метод `draw( )`, либо любой из следующих методов класса `BitmapData`.

- `copyPixels( )` — копирует значения цвета из прямоугольной области пикселов объекта-источника `BitmapData` в объект-получатель `BitmapData`. Источ-

ник и получатель могут являться одним объектом, позволяя копировать пикселы из одной области изображения в другую область того же изображения.

- `copyChannel( )` — копирует отдельный цветовой канал из прямоугольной области пикселов объекта-источника `BitmapData` в объект-получатель `BitmapData`. Источник и получатель могут являться одним объектом, позволяя копировать пикселы из одной области изображения в другую область того же изображения.
- `clone( )` — создает новый объект `BitmapData`, дублируя существующий объект `BitmapData`.
- `merge( )` — смешивает вместе каналы двух объектов `BitmapData`, создавая новое изображение, в котором одно изображение оказывается наложенным на другое. Объект-источник и объект-получатель `BitmapData` могут быть одним объектом, позволяя смешивать два канала одного и того же изображения.

В этом разделе мы сосредоточимся на методах `draw( )` и `copyPixels( )`. Дополнительную информацию о других методах копирования можно найти в описании класса `BitmapData` в справочнике по языку ActionScript корпорации Adobe.

Метод экземпляра draw( ) класса BitmapData

Метод `draw( )` имеет следующий обобщенный вид:

целевойОбъектBitmapData.draw(*источник*, *матрицаПреобразования*, *цветовыеПреобразования*, *режимСмешения*, *областьОбрезки*, *сглаживание*)

Здесь *целевойОбъектBitmapData* — объект `BitmapData`, в который будут перенесены пикселы. Параметры метода `draw( )` описаны ниже.

- *источник* — экземпляр класса `DisplayObject` или `BitmapData`, графические данные которого будут перенесены в объект *целевойОбъектBitmapData*. Это единственный обязательный параметр метода `draw( )`. Стоит отметить, что, когда значением параметра *источник* является объект `DisplayObject`, при переносе в объект *целевойОбъектBitmapData* его преобразования не включаются. Тем не менее преобразования объекта *источник* могут быть включены вручную путем передачи значения переменной *источник*.transform.matrix в качестве параметра *матрицаПреобразований* метода `draw( )`, а значение переменной *источник*.transform.colorTransform — в качестве параметра *цветовоеПреобразование* метода `draw( )`. В качестве альтернативы объект *целевойОбъектBitmapData* может быть связан с объектом `BitmapData`, переменная экземпляра `transform` которого ссылается на переменную *источник*.transform.
- *матрицаПреобразования* — необязательный объект `Matrix`, описывающий любое перемещение (то есть изменение позиции), масштабирование, вращение и искажение, которое должно быть применено к пикселам, переносимым в объект *целевойОбъектBitmapData*. Информацию об использовании объекта `Matrix` для выполнения графических преобразований можно найти в описании класса `Matrix` в справочнике по языку ActionScript корпорации Adobe и в разделе Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Working With Geometry ▸ Using Matrix objects документации корпорации Adobe. Общий пример матричных преобразований можно найти по адресу http://windowssdk.msdn.microsoft.com/en-us/library/ms536397.aspx и http://www.senocular.com/flash/tutorials/transformmatrix.

Стоит отметить, что гарантировать достаточный размер объекта *целевойОбъектBitmapData* для хранения преобразованного объекта *источник* должен программист. В интерфейсе API приложения Flash Player 9 не предусмотрено никакой возможности для предварительного получения размера преобразованного объекта *источник*. Такая возможность может быть включена в будущие версии сред Flash, например, в виде метода `generateTransformRect( )` (разработанного после существующего метода `generateFilterRect( )`). Чтобы отдать свой голос в поддержку подобного метода, посетите страницу http://www.adobe.com/cfusion/mmform/index.cfm?name=wishform.

- *цветовыеПреобразования* — необязательный объект `ColorTransform`, описывающий любые цветовые изменения, которые должны быть применены к пикселам, переносимым в объект *целевойОбъектBitmapData*. Цветовые преобразования задаются независимо для каждого цветового канала либо с помощью множителя (числа, на которое умножается существующее значение цветового канала), либо с помощью смещения (числа, которое прибавляется к существующему значению цветового канала), либо с помощью обоих способов. Информацию по использованию объекта `ColorTransform` для выполнения графических преобразований можно найти в описании класса `ColorTransform` в справочнике по языку ActionScript корпорации Adobe.
- *режимСмешения* — необязательная константа класса `BlendMode`, обозначающая тип смешения, который должен быть применен к пикселам, переносимым в объект *целевойОбъектBitmapData*. *Смешение* означает использование формул для объединения значений цвета объекта *источник* с отображаемыми объектами, которые визуально располагаются позади него, обычно с целью создания эффекта наложения. Поддерживаемыми режимами смешения являются `BlendMode.MULTIPLY`, `BlendMode.SCREEN`, `BlendMode.HARDLIGHT` и многие другие, которые знакомы пользователям программы Adobe Photoshop. Реализация режимов смешения в языке ActionScript основывается на стандарте SVG консорциума W3C (описание этого стандарта доступно по адресу http://www.w3.org/TR/2003/WD-SVG12-20030715/#compositing) и исследовании Дженса Грашела (Jens Gruschel), опубликованного по адресу http://www.pegtop.net/delphi/articles/blendmodes. Описание каждого доступного режима смешения и изображения, иллюстрирующие результаты применения этих режимов, можно найти в описании переменной экземпляра `blendMode` класса `DisplayObject` в справочнике по языку ActionScript корпорации Adobe.
- *областьОбрезки* — необязательный объект `Rectangle`, обозначающий прямоугольную область объекта *целевойОбъектBitmapData*, в которую будут перенесены графические данные объекта *источник*.
- *сглаживание* — необязательный параметр типа `Boolean`, который обозначает, должно ли выполняться сглаживание растрового изображения во время рисования. Этот параметр оказывает влияние на результат только в том случае, когда объект *источник* является объектом `BitmapData` *и* указанный объект *матрицаПреобразования* задает параметры масштабирования или вращения. При этом, когда параметру *сглаживание* присвоено значение `true`, объект *источник* отображается в объекте *целевойОбъектBitmapData* с использованием алгоритма сглаживания растровых изображений языка ActionScript. Когда параметру

сглаживание присвоено значение `false`, объект *источник* отображается в объекте *целевойОбъектBitmapData* без сглаживания. Изображение, выводимое с использованием сглаживания, выглядит менее «зазубренным» или «пикселизированным», чем изображение, отображаемое без сглаживания. Это проиллюстрировано на рис. 26.6, где показано небольшое исходное изображение (вверху), которое увеличивается в три раза с помощью объекта `Matrix` с применением сглаживания (слева) и без применения сглаживания (справа).

Вывод на экран растрового изображения с применением сглаживания занимает больше времени, чем вывод без сглаживания. Чтобы достичь максимально возможной производительности, передавайте в качестве параметра *сглаживание* значение `false`; чтобы достичь максимально возможного качества изображения, передавайте в качестве параметра *сглаживание* значение `true`. Параметр *сглаживание* оказывает влияние только на текущую операцию `draw( )`; он никак не влияет на применение сглаживания к объекту *целевойОбъектBitmapData* в дальнейшем.

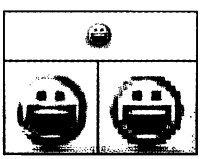

Рис. 26.6. Сглаживание растрового изображения

Метод `draw( )` обычно применяется для:

- объединения нескольких отображаемых объектов в одно растровое изображение;
- *растеризации* векторного содержимого (то есть преобразования векторов в растровое изображение) с целью применения некоторого эффекта.

Рассмотрим пример каждого случая. Сначала мы создадим две векторные фигуры — прямоугольник и эллипс — и перенесем их в один объект `BitmapData`. Этот код показан в листинге 26.8.

Векторные фигуры, переносимые в объект BitmapData, не обязаны находиться в списке отображения. Совершенно нормально создавать векторные объекты, не отображая их на экране, просто для того, чтобы скопировать эти объекты в растровое изображение (как показано в листингах 26.8 и 26.9).

Листинг 26.8. Отображаемые объекты, объединенные в растровом изображении

```
// Создаем прямоугольник
var rect:Shape = new Shape( );
rect.graphics.beginFill(0xFF0000);
rect.graphics.drawRect(0,0,25,50);

// Создаем эллипс
var ellipse:Shape = new Shape( );
ellipse.graphics.beginFill(0x0000FF);
ellipse.graphics.drawEllipse(0,0,35,25);
```

```
// Создаем объект BitmapData. Он будет выступать в роли холста
// для рисования, поэтому мы присваиваем его переменной с именем canvas.
var canvas:BitmapData = new BitmapData(100, 100, false, 0xFFFFFFFF);

// Переносим векторное изображение прямоугольника на растровое изображение
canvas.draw(rect);

// Переносим векторное изображение эллипса на растровое изображение.
// Используем матрицу преобразования, чтобы поместить эллипс в точку
// с координатой (10, 10) внутри объекта BitmapData.
var matrix:Matrix = new Matrix( );
matrix.translate(10, 10);
canvas.draw(ellipse, matrix);

// Связываем объект BitmapData с объектом Bitmap, чтобы отобразить его
// на экране
var bmp:Bitmap = new Bitmap(canvas);
addChild(bmp);
```

На рис. 26.7 показано растровое изображение, полученное в результате выполнения предыдущего кода, с увеличенным масштабом для более тщательного рассмотрения. Обратите внимание, что среда выполнения Flash сглаживает края эллипса, поэтому он смешивается с существующим прямоугольником.

Сглаживание содержимого, переносимого в объект BitmapData, происходит по отношению к существующему фону в растровом изображении.

Далее мы воспользуемся методом `draw( )` для выполнения растеризации объекта `TextField`, чтобы можно было применить к нему эффект растворения на уровне пикселов. Этот код представлен в листинге 26.9.

Рис. 26.7. Отображаемые объекты, объединенные в растровом изображении

Листинг 26.9. Растеризация и последующее растворение объекта TextField

```
package {
  import flash.display.*;
  import flash.utils.*;
  import flash.events.*;
  import flash.geom.*;
  import flash.text.*;

  public class DissolveText extends Sprite {
    // Переменные, используемые для эффекта растворения
    private var randomSeed:int = Math.floor(Math.random( ) * int.MAX_VALUE);
```

```
private var destPoint:Point = new Point(0, 0);
private var numberOfPixels:int = 10;
private var destColor:uint = 0xFF000000;

// Объект, в который будет переноситься текст
private var bitmapData:BitmapData;
// Таймер, используемый для периодического вызова метода pixelDissolve( )
private var t:Timer;

// Конструктор
public function DissolveText ( ) {
  // Создаем текст
  var txt:TextField = new TextField( );
  txt.text = "Essential ActionScript 3.0";
  txt.autoSize = TextFieldAutoSize.LEFT;
  txt.textColor = 0xFFFFFF;

  // Создаем объект BitmapData с размерами,
  // достаточными для размещения текста
  bitmapData = new BitmapData(txt.width, txt.height, false, destColor);
  // Переносим текст в объект BitmapData
  bitmapData.draw(txt);

  // Связываем объект BitmapData с объектом Bitmap, чтобы отобразить его
  // на экране
  var bitmap:Bitmap = new Bitmap(bitmapData);
  addChild(bitmap);

  // Начинаем периодически вызывать метод pixelDissolve( )
  t = new Timer(10);
  t.addEventListener(TimerEvent.TIMER, timerListener);
  t.start( );
}

// Обрабатывает события TimerEvent.TIMER
private function timerListener (e:TimerEvent):void {
  dissolve( );
}

// Выполняет растворение
public function dissolve( ):void {
  // Вызываем метод pixelDissolve( ), чтобы растворить указанное
  // количество пикселов, и запоминаем возвращенное псевдослучайное
  // число для следующего раза. Использование возвращаемого
  // псевдослучайного числа гарантирует плавное растворение.
  randomSeed = bitmapData.pixelDissolve(bitmapData,
                                        bitmapData.rect,
                                        destPoint,
                                        randomSeed,
                                        numberOfPixels,
                                        destColor);
```

```
      // Прекращаем растворение, когда все пикселы будут иметь целевой цвет
      // (то есть когда ширина и высота области, в которой отсутствует
      // целевой цвет, будут равны 0)
      var coloredRegion:Rectangle =
                bitmapData.getColorBoundsRect(0xFFFFFFFF, destColor, false);
      if (coloredRegion.width == 0 && coloredRegion.height == 0 ) {
        t.stop( );
      }
    }
  }
}
```

Метод `draw( )` также используется для растеризации векторного содержимого с целью повышения производительности. Например, вспомните простую программу рисования ScribbleAS3, представленную ранее в листинге 26.6. Она рисует точки всякий раз, когда происходит перемещение мыши, но не соединяет нарисованные точки линиями. Одним из способов соединить точки является использование метода экземпляра `lineTo( )` класса `Graphics`, который рисует векторную линию между двумя точками. Тем не менее применение векторов для рисования линий ограничивает производительность: теоретически можно нарисовать такое количество линий, которое не позволит приложению реагировать на действия пользователя (из-за ограничений в механизме отображения векторов среды выполнения Flash). Для решения этой проблемы мы можем рисовать линии с помощью метода `lineTo( )` в экземпляре класса `Shape`, не отображаемом на экране. После рисования каждой линии мы копируем ее из объекта `Shape` в объект `BitmapData`, после чего очищаем данный экземпляр класса `Shape`. Поскольку за один раз рисуется только одна векторная линия, приложение никогда не прекратит реагировать на действия пользователя.

В листинге 26.10 продемонстрирован код, часть которого вам знакома из листинга 26.6. Обратите особое внимание на метод `drawLine( )`, который выделен полужирным шрифтом.

Листинг 26.10. Приложение ScribbleAS3, версия с не отображаемыми на экране векторами

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.ui.*;
  import flash.geom.*;


  // Простое приложение для рисования. В данной версии вектор рисуется
  // за пределами экрана, после чего он копируется на поверхность растрового
  // изображения, исключая снижение производительности, которое связано
  // с рисованием слишком большого количества векторов.
  public class ScribbleAS3_VectorV2 extends Sprite {
    private var canvas:Bitmap;           // Холст растрового изображения,
                                         // отображаемого на экране
    private var virtualCanvas:Shape;     // Холст векторного изображения,
                                         // не отображаемого на экране
    private var canvasContainer:Sprite;  // Содержит растровое изображение,
                                         // обеспечивая интерактивность
```

```
private var isDrawing:Boolean = false; // Сообщает о том, нажата ли
                                       // кнопка мыши в настоящий момент
private var border:Shape; // Линия вокруг растрового изображения
private var lastX:int;    // x-координата последней точки,
                          // в которой щелкнул кнопкой мыши пользователь
private var lastY:int;    // y-координата последней точки,
                          // в которой щелкнул кнопкой мыши пользователь

// Конструктор
public function ScribbleAS3_VectorV2 ( ) {
  createCanvas( );
  registerForInputEvents( );

  // Предотвращаем изменение размеров окна приложения
  stage.scaleMode = StageScaleMode.NO_SCALE;
}

// Создает холст растрового изображения, отображаемого на экране,
// и холст векторного изображения, не отображаемого на экране
private function createCanvas (width:int = 200, height:int = 200):void {
  // Создаем новый объект, не отображаемый на экране, в котором будет
  // происходить рисование векторных линий перед их переносом
  // в объект canvasData
  virtualCanvas = new Shape( );

  // Определяем объект данных, который будет хранить реальные пиксельные
  // данные для рисунка пользователя. Линии копируются из объекта
  // virtualCanvas в данный объект.
  var canvasData:BitmapData = new BitmapData(width,
                                             height, false, 0xFFFFFFFF);

  // Создаем новый отображаемый объект Bitmap, используемый
  // для отображения объекта canvasData
  canvas = new Bitmap(canvasData);

  // Создаем объект Sprite, который будет содержать объект Bitmap. Класс
  // Bitmap не поддерживает пересылаемые события ввода, поэтому помещаем его
  // в объект Sprite, чтобы пользователь мог взаимодействовать с этим объектом.
  canvasContainer = new Sprite( );
  canvasContainer.addChild(canvas);

  // Добавляем экземпляр canvasContainer класса Sprite (и содержащийся
  // в нем объект Bitmap) в иерархию отображения данного объекта
  addChild(canvasContainer);

  // Создаем границу вокруг области рисования.
  border = new Shape( );
  border.graphics.lineStyle(1);
  border.graphics.drawRect(0, 0, width, height);
  addChild(border);
}
```

```
// Регистрирует приемники для необходимых событий мыши и клавиатуры
private function registerForInputEvents ( ):void {
  // Регистрируем приемники для событий нажатия кнопки мыши
  // и перемещения мыши от объекта canvasContainer
  canvasContainer.addEventListener(MouseEvent.MOUSE_DOWN,
                                   mouseDownListener);
  canvasContainer.addEventListener(MouseEvent.MOUSE_MOVE,
                                   mouseMoveListener);

  // Регистрируем приемники для событий отпускания кнопки мыши и нажатия
  // клавиши от объекта Stage (то есть для глобальных событий).
  // Используем объект Stage, поскольку событие отпускания кнопки мыши
  // должно всегда завершать рисование, даже если указатель мыши
  // не находится над областью рисования. Подобным образом, нажатие
  // пробела должно всегда приводить к стиранию рисунка, даже когда
  // объект canvasContainer не имеет фокуса.
  stage.addEventListener(MouseEvent.MOUSE_UP, mouseUpListener);
  stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDownListener);
}

// Устанавливает цвет указанного пиксела. Используем этот способ
// для рисования одиночного пиксела, поскольку метод Graphics.lineTo( )
// не рисует одиночные пикселы
public function drawPoint (x:int, y:int, color:uint = 0xFF000000):void {
  // Устанавливаем цвет указанного пиксела
  canvas.bitmapData.setPixel(x, y, color);
}

// Рисует векторную линию в объекте virtualCanvas, а затем копирует
// растровое представление этой линии в объект canvasData (доступ
// к растровому представлению осуществляется через переменную
// canvas.bitmapData)
public function drawLine (x1:int, y1:int, x2:int, y2:int,
                          color:uint = 0xFF000000):void {
  // Рисуем линию в объекте virtualCanvas
  virtualCanvas.graphics.clear( );
  virtualCanvas.graphics.lineStyle(1, 0x000000, 1, true,
                                   LineScaleMode.NORMAL, CapsStyle.NONE);
  virtualCanvas.graphics.moveTo(x1, y1);
  virtualCanvas.graphics.lineTo(x2, y2);
  // Копируем линию в объект canvasData
  canvas.bitmapData.draw(virtualCanvas);
}

// Отвечает на события MouseEvent.MOUSE_DOWN
private function mouseDownListener (e:MouseEvent):void {
  // Устанавливаем флажок, указывающий на то, что основная кнопка мыши
  // в настоящий момент нажата
  isDrawing = true;
  // Запоминаем точку, в которой щелкнул пользователь, чтобы в случае
  // перемещения мыши мы могли нарисовать линию из этой точки
  lastX = e.localX;
```

```
      lastY = e.localY;
      // Рисуем точку в позиции, где произошел щелчок кнопкой мыши
      drawPoint(e.localX, e.localY);
    }

    // Отвечает на события MouseEvent.MOUSE_MOVE
    private function mouseMoveListener (e:MouseEvent):void {
      // Рисуем линию, когда мышь перемещается над областью рисования
      // при нажатой основной кнопке мыши
      if (isDrawing) {
        // Используем переменные localX и localY, чтобы получить позицию
        // указателя относительно объекта canvasContainer.
        var thisX:int = e.localX;
        var thisY:int = e.localY;

        // Рисуем линию в новую позицию указателя мыши
        drawLine(lastX, lastY, thisX, thisY);

        // Запоминаем последнюю позицию указателя мыши
        // для следующего раза
        lastX = thisX;
        lastY = thisY;

        // Обновляем экран сразу после завершения выполнения
        // данной функции-приемника события
        e.updateAfterEvent( );
      }
    }

    // Отвечает на события MouseEvent.MOUSE_UP
    private function mouseUpListener (e:MouseEvent):void {
      // Устанавливаем флажок, указывающий на то, что в настоящий момент
      // основная кнопка мыши отпущена
      isDrawing = false;
    }

    // Отвечает на события KeyboardEvent.KEY_DOWN
    private function keyDownListener (e:KeyboardEvent):void {
      // Стираем рисунок, когда пользователь нажимает клавишу Пробел.
      // Чтобы очистить рисунок, мы присваиваем всем пикселам
      // значение белого цвета.
      if (e.charCode == Keyboard.SPACE) {
        canvas.bitmapData.fillRect(new Rectangle(0, 0,
                                          canvas.width, canvas.height),
                                   0xFFFFFFFF);
      }
    }
  }
}
```

Как метод draw( ) использует значения канала Alpha. Когда прозрачный объект-источник `BitmapData` копируется в прозрачный целевой объект `BitmapData`

с помощью метода draw( ), каналы Alpha двух объектов BitmapData объединяются вместе на уровне пикселов с использованием алгоритма режима смешения BlendMode.SCREEN, который записывается следующим образом:

(значениеAlphaИсходногоОбъекта * (256-*значениеAlphaЦелевогоОбъекта*) / 256) + *значениеAlphaЦелевогоОбъекта*

Когда значения канала Alpha двух объединяемых объектов находятся в диапазоне от 1 до 254, результатом будет являться значение канала Alpha с большей непрозрачностью, чем любое из исходных значений канала Alpha. Если объединяемый источник является полностью прозрачным, целевой объект BitmapData сохраняет свое исходное значение канала Alpha. Если целевой объект является полностью прозрачным, его значение канала Alpha заменяется значением канала Alpha объекта-источника.

Чтобы полностью заменить значения канала Alpha в целевом объекте BitmapData значениями объекта-источника BitmapData (вместо объединения двух значений), используйте метод copyPixels( ) вместо метода draw( ). Дополнительную информацию можно найти в подразд. «Метод экземпляра copyPixels( ) класса BitmapData» данного раздела.

Невозможность произвольного захвата изображения экрана. Стоит отметить, что в ActionScript невозможно захватить изображение экрана в произвольной прямоугольной области. Язык ActionScript позволяет лишь преобразовывать отображаемые объекты в растровый формат. В ActionScript ближайшим аналогом операции захвата изображения области отображения является использование экземпляра класса Stage в качестве параметра *источник* метода draw( ), как показано в следующем коде:

```
var canvas:BitmapData = new BitmapData(100, 100, false, 0xFFFFFFFF);
canvas.draw(некийОбъектDisplayObject.stage);
```

Здесь *некийОбъектDisplayObject* — это экземпляр класса DisplayObject, находящийся в списке отображения. Предыдущий код создает растровое изображение, содержащее все объекты, которые в настоящий момент находятся в списке отображения, со следующими оговорками:

- цвет фона SWF-файла не копируется в растровое изображение;
- если доступ к объектам, находящимся в списке отображения, запрещен из-за ограничений безопасности, они не копируются в растровое изображение и генерируется исключение SecurityError.

Метод экземпляра copyPixels( ) класса BitmapData

Как и draw( ), метод copyPixels( ) применяется для копирования значений цвета пикселов из объекта-источника в целевой объект BitmapData. Однако в отличие от метода draw( ), который копирует пиксельные данные из любого экземпляра класса DisplayObject или объекта BitmapData, метод copyPixels( ) может копировать их только из объектов BitmapData. Этот метод отличается производительностью и удобством использования. Тестирование показывает, что операции copyPixels( ) быстрее эквивалентных операций draw( ) на 25–300 %.

Чтобы достичь максимальной производительности при копировании значений цвета пикселов между двумя объектами BitmapData, используйте метод copyPixels( ) вместо метода draw( ).

Помимо того что метод `copyPixels( )` производительнее метода `draw( )`, он предоставляет разработчику простой доступ к таким операциям, как:

- размещение пикселов из объекта-источника в определенной точке объекта-получателя;
- объединение канала Alpha одного растрового изображения с каналом Alpha другого растрового изображения;
- перезапись значений канала Alpha целевого растрового изображения при копировании в него пикселов из исходного растрового изображения.

Хотя все три перечисленные операции также могут быть реализованы с помощью метода `draw( )` совместно с другими методами класса `BitmapData`, метод `copyPixels( )` обычно предпочтителен благодаря его удобству.

Метод `copyPixels( )` имеет следующий вид:

целевойОбъектBitmapData.copyPixels(*исходныйОбъектBitmapData*, *исходнаяОбласть*, *целеваяТочка*, *объектBitmapDataКаналаAlpha*, *точкаКаналаAlpha*, *объединениеКаналовAlpha*)

Здесь *целевойОбъектBitmapData* — это объект `BitmapData`, в который будут перенесены пикселы. Рассмотрим параметры метода `draw( )`.

- *исходныйОбъектBitmapData* — экземпляр класса `BitmapData`, который будет скопирован в объект *целевойОбъектBitmapData*. Объекты *исходныйОбъектBitmapData* и *целевойОбъектBitmapData* могут являться одним объектом, позволяя копировать пикселы из одной области изображения в другую область того же изображения.
- *исходнаяОбласть* — объект `Rectangle`, определяющий область объекта *исходныйОбъектBitmapData*, которая будет скопирована в объект *целевойОбъектBitmapData*. Чтобы скопировать весь объект *исходныйОбъектBitmapData*, используйте переменную *исходныйОбъектBitmapData*.rect. Если в функцию передается аргумент *объектBitmapDataКаналаАльфа*, данный параметр также определяет ширину и высоту прямоугольной области внутри объекта *объектBitmapDataКаналаAlpha*, значения канала Alpha которой будут скопированы в объект *целевойОбъектBitmapData*.
- *целеваяТочка* — объект `Point`, определяющий позицию левого верхнего угла прямоугольной области внутри объекта *целевойОбъектBitmapData*, в которую будут помещены копируемые пикселы.
- *объектBitmapDataКаналаAlpha* — необязательный объект `BitmapData`, отличный от объекта *исходныйОбъектBitmapData*, значения канала Alpha которого станут новыми значениями канала Alpha пикселов, переносимых в объект *целевойОбъектBitmapData*. Высота и ширина конкретной прямоугольной области, значения канала Alpha которой будут скопированы в объект *целевойОбъектBitmapData*, определяются параметром *исходнаяОбласть*.

С помощью этого параметра мы можем объединять RGB-каналы одного растрового изображения (*исходныйОбъектBitmapData*) с каналом Alpha другого растрового изображения (*объектBitmapDataКаналаAlpha*). Подобная методика может быть использована, например, для создания на фотографиях в электронном приложении, имитирующем альбом для наклеивания газетных вырезок, эффекта краев неправильной формы. Каждая фотография могла бы храниться в своем собственном объекте `BitmapData`, а края неправильной формы можно было бы хранить в виде значений канала Alpha в одном повторно используемом объекте `BitmapData`. С помощью параметра *объектBitmapDataКаналаAlpha* метода `copyPixels( )` канал Alpha растрового изображения с краями неправильной формы можно было бы объединять с фотографиями на этапе выполнения программы.

- *точкаКаналаAlpha* — объект `Point`, определяющий левый верхний угол прямоугольной области внутри объекта *объектBitmapDataКаналаAlpha*, из которой будут получены значения канала Alpha. Ширина и высота прямоугольной области задаются параметром *исходнаяОбласть*.
- *объединениеКаналовAlpha* — значение типа `Boolean`, которое показывает, должны ли значения каналов Alpha объектов *целевойОбъектBitmapData* и *исходныйОбъектBitmapData* в процессе копирования данных объединяться (`true`), или значения канала Alpha объекта *исходныйОбъектBitmapData* должны полностью заменять существующие значения канала Alpha объекта *целевойОбъектBitmapData* (`false`). Этот параметр оказывает влияние на результат только в том случае, когда оба объекта *целевойОбъектBitmapData* и *исходныйОбъектBitmapData* являются прозрачными растровыми изображениями. Значением по умолчанию является `false`, указывающее, что значения канала Alpha объекта *исходныйОбъектBitmapData* полностью заменяют существующие значения канала Alpha объекта *целевойОбъектBitmapData*. Алгоритм, используемый для объединения значений каналов Alpha, соответствует алгоритму, рассмотренному ранее в подразд. «Метод экземпляра draw( ) класса BitmapData» данного раздела.

Метод `copyPixels( )` предпочтительно применять для перемещения пикселов между двумя объектами `BitmapData`. Перемещение пикселов между растровыми изображениями является распространенной операцией в приложениях для работы с графикой и в видеоиграх. Рассмотрим несколько примеров.

Попрактикуемся в использовании базового синтаксиса метода `copyPixels( )`, создав квадраты синего и красного цвета и скопировав область квадрата синего цвета в квадрат красного цвета.

```
// Создаем квадраты (размером 20 × 20 пикселов каждый)
var redSquare:BitmapData = new BitmapData(20, 20, true, 0xFFFF0000);
var blueSquare:BitmapData = new BitmapData(20, 20, true, 0xFF0000FF);

// Определяем прямоугольную область, которая будет скопирована из объекта
// blueSquare в объект redSquare
var sourceRect:Rectangle = new Rectangle(5, 5, 10, 5);

// Определяем точку в объекте redSquare, в которую будет помещена
// прямоугольная область из объекта blueSquare
var destPoint:Point = new Point(0,0);
```

```
// Копируем пикселы
redSquare.copyPixels(blueSquare, sourceRect, destPoint);

// Связываем объект BitmapData redSquare с объектом Bitmap для отображения
// на экране
var b:Bitmap = new Bitmap(redSquare);
addChild(b);
```

В листинге 26.11 представлен другой пример — создание фотографии с краями неправильной формы, как было описано ранее при рассмотрении параметра *объектBitmapDataКаналаAlpha*. Понять код вам помогут комментарии. Обратите особое внимание на метод `makeScrapbookImage( )`.

Листинг 26.11. Эффект старой фотографии

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.geom.*;
  import flash.net.*;

  public class ScrapbookImage extends Sprite {
    private var numLoaded:int = 0;
    private var photoLoader:Loader;   // Загрузчик фотографии
    private var borderLoader:Loader;  // Загрузчик рамки

    // Конструктор
    public function ScrapbookImage ( ) {
      // Загружаем фотографию
      photoLoader = new Loader( );
      photoLoader.contentLoaderInfo.addEventListener(Event.INIT,
                                                     initListener);
      photoLoader.load(new URLRequest("photo.jpg"));

      // Загружаем рамку
      borderLoader = new Loader( );
      borderLoader.contentLoaderInfo.addEventListener(Event.INIT,
                                                      initListener);
      borderLoader.load(new URLRequest("border.png"));
    }

    // Обрабатывает события Event.INIT для загруженных изображений
    private function initListener (e:Event):void {
      numLoaded++;
      if (numLoaded == 2) {
        makeScrapbookImage( );
      }
    }

    // Объединяет изображение рамки с изображением фотографии, чтобы создать
    // эффект старой фотографии
    public function makeScrapbookImage ( ):void {
```

```
            // Получаем объект BitmapData для фотографии
            var photoData:BitmapData = Bitmap(photoLoader.content).bitmapData;
            // Получаем объект BitmapData для рамки
            var borderData:BitmapData = Bitmap(borderLoader.content).bitmapData;
            // Создаем объект BitmapData, который будет хранить завершенное
            // изображение фотографии
            var scrapbookImage:BitmapData = new BitmapData(borderData.width,
                                                           borderData.height,
                                                           true,
                                                           0xFFFFFFFF);

            // Копируем пикселы из фотографии,
            // применяя значения канала Alpha рамки
            scrapbookImage.copyPixels(photoData,
                                      borderData.rect,
                                      new Point(0,0),
                                      borderData,
                                      new Point(0,0),
                                      true);

            // Связываем объект BitmapData scrapbookImage с объектом Bitmap
            // для отображения на экране
            var b:Bitmap = new Bitmap(scrapbookImage);
            addChild(b);
            b.x = 100;
            b.y = 75;
        }
    }
}
```

На рис. 26.8 проиллюстрированы результаты выполнения кода из листинга 26.11. Исходное изображение показано слева, а изображение с «краями» — справа.

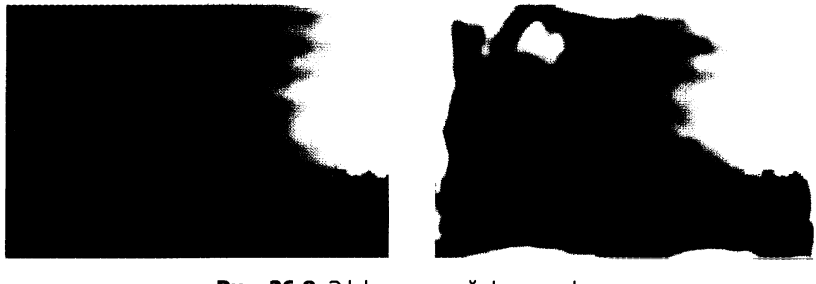

Рис. 26.8. Эффект старой фотографии

Изображения и код из листинга 26.11 доступны в Интернете по адресу http://www.moock.org/eas3/examples.

Метод `copyPixels( )` также предоставляет эффективный способ для повторного использования набора значений пикселов. Например, в двухмерных видеоиграх фоновые изображения зачастую генерируются динамически из небольшой группы готовых растровых изображений, называемых *мозаиками*. По мере перемещения игрока по виртуальному миру программа отображает соответствующий набор

фоновых мозаик. С помощью метода `copyPixels( )` каждая мозаика копируется из повторно используемого объекта `BitmapData` в фоновое изображение, отображаемое на экране. Хотя подробное рассмотрение системы мозаичных фоновых изображений выходит за рамки данной книги, полнофункциональный пример мозаики на языке ActionScript может быть загружен с сайта http://www.moock.org/eas3/examples.

Применение фильтров и эффектов

Для создания графических эффектов мы можем использовать следующий широкий диапазон инструментов, предоставляемых классом `BitmapData`.

- `colorTransform( )` — изменяет цвета с помощью объекта `ColorTransform`, который предоставляет базовый интерфейс для простейших цветовых преобразований (для более сложного управления цветовыми преобразованиями используйте класс `ColorMatrixFilter`).
- `applyFilter( )` — изменяет растровое изображение с помощью предопределенных фильтров-эффектов, например падающей тени или размытия. Дополнительную информацию можно найти далее в разделе.
- `noise( )` — заполняет растровое изображение случайными значениями цветов из настраиваемых диапазонов. На рис. 26.9 показано изображение, созданное с помощью метода `noise( )`.

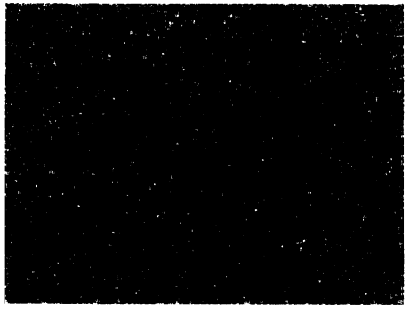

Рис. 26.9. Растровое изображение, заполненное шумом

- `perlinNoise( )` — заполняет растровое изображение случайными образцами значений цветов, выполненных в органическом стиле. На рис. 26.10 показано изображение, сгенерированное с помощью метода `perlinNoise( )`. Результаты выполнения метода `perlinNoise( )` обычно не используются напрямую; этот фильтр часто применяется вместе с другими фильтрами для создания имитации волн, пламени, облаков, воды, текстуры древесины и ландшафтов. Общее описание шума Перлина можно найти во вводной лекции Кена Перлина (Ken Perlin) по адресу http://www.noisemachine.com/talk1. Пример на языке ActionScript, демонстрирующий использование метода `perlinNoise( )` для создания текстуры древесины, можно найти по адресу http://www.connectedpixel.com/blog/texture/wood. Пример создания текстуры мрамора вы найдете по адресу http://www.connectedpixel.com/blog/texture/marble.

Рис. 26.10. Растровое изображение, заполненное с помощью шума Перлина

- `paletteMap( )` — заменяет цвета в изображении цветами из таблицы перекодировки или другого изображения.
- `pixelDissolve( )` — при циклическом использовании в анимации создает эффект перехода между двумя изображениями или пиксел за пикселом заменяет цвета изображения указанным цветом (пример замены цветов изображения указанным цветом можно найти в листинге 26.9).
- `threshold( )` — преобразует пикселы из некоторого диапазона цветов к новому указанному значению цвета.

Хотя каждый из перечисленных методов предоставляет множество возможностей для работы с графикой, в этом разделе мы сосредоточим внимание на инструменте, который, возможно, обладает наиболее универсальной применимостью, — методе `applyFilter( )`. Дополнительную информацию о других методах, предназначенных для создания эффектов, можно найти в описании класса `BitmapData` в справочнике по языку ActionScript корпорации Adobe.

Применение фильтров. Чтобы предоставить программистам удобный способ применения распространенных графических эффектов к растровому изображению, язык ActionScript предлагает набор готовых графических фильтров.

Фильтры, рассматриваемые в этом разделе, могут на постоянной основе применяться к объекту BitmapData на уровне пикселов, но они также могут применяться к любому отображаемому объекту или удаляться из него динамически. Дополнительную информацию можно получить в описании переменной экземпляра filters класса DisplayObject в справочнике по языку ActionScript корпорации Adobe.

Каждый предопределенный фильтр представляется собственным классом в пакете `flash.filters`. Например, фильтр размытия, который делает растровое изображение размытым, представляется классом `flash.filters.BlurFilter`.

Некоторые фильтры могут быть легко использованы даже теми разработчиками, которые не имеют никакого опыта в программировании графики. К этой категории относятся следующие фильтры: фаска, размытие, падающая тень и свечение, которые представляются классами `BevelFilter` (и его разновидностью с градиентом `GradientBevelFilter`), `BlurFilter`, `DropShadowFilter` и `GlowFilter` (и его разновидностью с градиентом `GradientGlowFilter`).

Другие фильтры требуют понимания фундаментальных методик программирования графики, например таких, как искривление, замещение и цветовые матрицы. К этой категории относятся следующие фильтры: матричное искривление, карта замещения и преобразования цветовых матриц, которые представляются классами `ConvolutionFilter`, `DisplacementMapFilter` и `ColorMatrixFilter`.

Независимо от типа фильтра, методика применения фильтра к растровому изображению в языке ActionScript остается одной и той же.

1. Создать экземпляр класса нужного фильтра.
2. Передать созданный экземпляр класса фильтра в метод `applyFilter( )`.

Например, чтобы применить фильтр размытия к растровому изображению, мы создаем экземпляр класса `BlurFilter`, используя следующий обобщенный формат:

```
new BlurFilter(размытиеX, размытиеY, качество)
```

Здесь *размытиеX* и *размытиеY* обозначают дистанцию размытия каждого пиксела по горизонтали и вертикали, а *качество* — качество визуализации данного эффекта, выражаемое через одну из трех констант класса `BitmapFilterQuality`.

Следующий код создает эффект размытия среднего качества, охватывающий 15 пикселов по вертикали и горизонтали:

```
var blurFilter:BlurFilter =
                        new BlurFilter(15, 15, BitmapFilterQuality.MEDIUM);
```

После создания фильтра мы применяем его к объекту `BitmapData`, используя следующий код:

```
целевойОбъектBitmapData.applyFilter(исходныйОбъектBitmapData, исходнаяОбласть,
целеваяТочка, фильтр);
```

Здесь *целевойОбъектBitmapData* — это объект `BitmapData`, в который будут перенесены пикселы с примененным фильтром. Рассмотрим параметры метода `applyFilter( )`.

- *исходныйОбъектBitmapData* — объект `BitmapData`, к которому будет применен фильтр. Объекты *исходныйОбъектBitmapData* и *целевойОбъектBitmapData* могут быть одним объектом, позволяя применять фильтр непосредственно к некоторому объекту `BitmapData`.
- *исходнаяОбласть* — объект `Rectangle`, определяющий область объекта *исходныйОбъектBitmapData*, к которой будет применен фильтр. Чтобы применить фильтр ко всему растровому изображению, используйте переменную *исходныйОбъектBitmapData*.`rect`.
- *целеваяТочка* — объект `Point`, определяющий позицию внутри объекта *целевойОбъектBitmapData*, куда будут помещаться пикселы с примененным фильтром. Обратите внимание, что параметр *целеваяТочка* соответствует левому верхнему углу указанного объекта *исходнаяОбласть*, а не левому верхнему углу области пикселов, к которой применяется операция фильтрации. Дополнительная информация приводится далее, при рассмотрении метода `generateFilterRect( )`.
- *фильтр* — это объект используемого фильтра, например экземпляр какого-либо из следующих классов: `BevelFilter`, `GradientBevelFilter`,

`BlurFilter`, `DropShadowFilter`, `GlowFilter`, `GradientGlowFilter`, `ConvolutionFilter`, `DisplacementMapFilter` или `ColorMatrixFilter`.

Следующий код применяет созданный ранее объект `BlurFilter` ко всему объекту `BitmapData`:

```
bitmapData.applyFilter(bitmapData, bitmapData.rect, new Point(0,0), blurFilter);
```

На рис. 26.11 показаны результаты применения к изображению объекта `BlurFilter` размером 15 × 15 пикселов.

Рис. 26.11. Примененный фильтр размытия

Когда фильтр применяется к растровому изображению, размеры результирующего растрового изображения зачастую оказываются больше размеров исходного. Например, результатом применения фильтра падающей тени может оказаться тень, выходящая за границы исходного растрового изображения. Чтобы полностью сохранить результаты применения фильтра, мы должны убедиться, что целевое изображение способно уместить все переносимые пикселы с примененным фильтром. Чтобы определить размеры растрового изображения, достаточные для сохранения результатов применения фильтра, мы используем метод экземпляра `generateFilterRect( )` класса `BitmapData`. Этот метод возвращает объект `Rectangle`, который обозначает область пикселов, затрагиваемую данной операцией фильтрации. Метод имеет следующий обобщенный вид:

исходныйОбъектBitmapData`.generateFilterRect(`*исходнаяОбласть*`,` *фильтр*`)`

Метод `generateFilterRect( )` возвращает объект `Rectangle`, который обозначает область пикселов, затрагиваемую указанным объектом фильтра (*фильтр*) в том случае, если этот фильтр будет применен к указанной прямоугольной области (*исходнаяОбласть*) указанного объекта `BitmapData` (*исходныйОбъектBitmapData*).

Метод generateFilterRect( ) на самом деле не применяет фильтр. Он просто определяет область пикселов, которая будет затронута данным фильтром в случае его применения.

Попробуем применить метод `generateFilterRect( )` на практике. Наша цель — создать растровое изображение с эффектом падающей тени. Сначала мы создаем исходное растровое изображение без эффекта падающей тени. Исходное изображение представляет собой квадрат серого цвета размером 20 × 20 пикселов.

```
var origBitmap:BitmapData = new BitmapData(20, 20, false, 0xFFDDDDDD);
```

Далее мы создаем объект DropShadowFilter. Конструктор его класса имеет следующий вид:

```
DropShadowFilter(расстояние:Number=4.0, угол:Number=45, цвет:uint=0,
                 прозрачность:Number=1.0, размытиеX:Number=4.0,
                 размытиеY:Number=4.0, интенсивность:Number=1.0,
                 качество:int=1, внутренняя:Boolean=false,
                 выколотка:Boolean = false, скрытьОбъект:Boolean = false)
```

Сведения о различных параметрах конструктора класса DropShadowFilter можно найти в описании этого класса в справочнике по языку ActionScript корпорации Adobe. Вот наш объект DropShadowFilter:

```
var dsFilter:DropShadowFilter = new DropShadowFilter(4, 45, 0,
                                                     1, 10, 10,
                                                     2, BitmapFilterQuality.MEDIUM);
```

Далее используем метод generateFilterRect( ), чтобы определить, насколько увеличатся размеры нашего исходного изображения после применения объекта DropShadowFilter:

```
var filterRect:Rectangle = origBitmap.generateFilterRect(origBitmap.rect,
                                                         dsFilter);
```

Теперь мы можем создать новое растровое изображение с необходимыми размерами, в которое будет перенесено исходное растровое изображение с примененным фильтром падающей тени. Обратите внимание, что мы указываем высоту и ширину нового растрового изображения с использованием результатов, возвращаемых методом generateFilterRect( ) (выделены полужирным шрифтом):

```
var finalBitmap:BitmapData = new BitmapData(filterRect.width,
                                            filterRect.height, true);
```

Целевой объект BitmapData для фильтра эффекта падающей тени должен быть прозрачным.

Теперь, когда у нас есть исходное растровое изображение, объект DropShadowFilter и целевое растровое изображение с подходящими размерами, мы можем применить наш фильтр падающей тени, как показано в следующем коде:

```
finalBitmap.applyFilter(origBitmap, origBitmap.rect,
                        new Point(-filterRect.x, -filterRect.y),
                        dsFilter);
```

В данном коде обратите внимание, что передаваемый параметр *целеваяТочка* смещает пикселы с примененным фильтром на значение, равное расстоянию, на которое результат применения фильтра выходит за верхнюю и левую границу указанного параметра *исходнаяОбласть*. В нашем примере эффект фильтра применяется ко всему растровому изображению, поэтому левый верхний угол объекта *исходнаяОбласть* находится в точке с координатой (0; 0). Размытие падающей тени расширяет исходное растровое изображение на 9 пикселов вверх и на 9 пикселов влево. Следовательно, координата по оси X сгенерированного фильтром прямоугольника, как и координата по оси Y, будет равна –9. Чтобы переместить левый верхний угол

области пикселов с примененным фильтром вниз и вправо (в левый верхний угол растрового изображения, содержащего результат применения фильтра), мы указываем объект *целеваяТочка*, который использует обратные значения координат по осям X и Y прямоугольника фильтра:

```
new Point(-filterRect.x, -filterRect.y)
```

Для обзора ниже представлен весь код, создающий эффект падающей тени:

```
var origBitmap:BitmapData = new BitmapData(20, 20, false, 0xFFDDDDDD);
var dsFilter:DropShadowFilter = new DropShadowFilter(4, 45, 0,
                                                     1, 10, 10,
                                                     2, BitmapFilterQuality.MEDIUM);
var filterRect:Rectangle = origBitmap.generateFilterRect(origBitmap.rect,
                                                         dsFilter);
var finalBitmap:BitmapData = new BitmapData(filterRect.width,
                                            filterRect.height, true);
finalBitmap.applyFilter(origBitmap, origBitmap.rect,
                        new Point(-filterRect.x, -filterRect.y),
                        dsFilter);
```

На рис. 26.12 показаны результаты выполнения предыдущего кода.

Рис. 26.12. Изображение с примененным фильтром падающей тени

Теперь, когда мы рассмотрели основы использования фильтров, вернемся к листингу 26.11, чтобы улучшить эффект старой фотографии. В листинге 26.12 показан новый код. Общая методика создания и применения фильтров в этом примере уже должна быть вам знакома. Тем не менее в примере используется специальный фильтр, который мы еще не рассматривали: `ColorMatrixFilter`. Этот фильтр с помощью матричных преобразований изменяет цвета в растровом изображении для создания таких эффектов, как, например, настройка яркости, контраста и насыщенности, изменение оттенка. Этот пример демонстрирует, как использовать фильтр `ColorMatrixFilter` в его исходном виде, но по крайней мере два разработчика предоставляют бесплатный код для выполнения распространенных матричных преобразований:

- класс `ColorMatrix` Марио Клингеманна (Mario Klingemann) — http://www.quasimondo.com/archives/000565.php;
- класс `ColorMatrix` Гранта Скиннера (Grant Skinner) — http://www.gskinner.com/blog/archives/2005/09/flash_8_source.html.

Общую информацию по цветовым матричным преобразованиям можно найти в статье «Using Matrices for Transformations, Color Adjustments, and Convolution Effects in Flash» Фила Чанга (Phil Chung), доступной по адресу http://www.adobe.com/devnet/flash/articles/matrix_transformations_04.html.

Листинг 26.12. Эффект старой фотографии, теперь с фильтрами

```
package {
  import flash.display.*;
```

```
import flash.events.*;
import flash.geom.*;
import flash.net.*;
import flash.filters.*;

public class ScrapbookImage extends Sprite {
  private var numLoaded:int = 0;
  private var photoLoader:Loader;   // Загрузчик фотографии
  private var borderLoader:Loader;  // Загрузчик рамки

  public function ScrapbookImage ( ) {
    // Загружаем фотографию
    photoLoader = new Loader( );
    photoLoader.contentLoaderInfo.addEventListener(Event.INIT,
                                                   initListener);
    photoLoader.load(new URLRequest("photo.jpg"));

    // Загружаем рамку
    borderLoader = new Loader( );
    borderLoader.contentLoaderInfo.addEventListener(Event.INIT,
                                                    initListener);
    borderLoader.load(new URLRequest("border.png"));
  }

  // Обрабатывает события Event.INIT для загруженных изображений
  private function initListener (e:Event):void {
    numLoaded++;
    // Применяем эффект, когда загружены оба изображения
    if (numLoaded == 2) {
      makeScrapbookImage( );
    }
  }

  // Объединяет изображение рамки с изображением фотографии,
  // чтобы создать эффект старой фотографии
  public function makeScrapbookImage ( ):void {
    // Получаем объект BitmapData для фотографии
    var photoData:BitmapData = Bitmap(photoLoader.content).bitmapData;
    // Получаем объект BitmapData для рамки
    var borderData:BitmapData = Bitmap(borderLoader.content).bitmapData;
    // Создаем объект BitmapData, который будет хранить завершенное
    // изображение фотографии
    var tempBitmapData:BitmapData = new BitmapData(borderData.width,
                                                   borderData.height,
                                                   true,
                                                   0x00000000);

    // Копируем пикселы из фотографии, применяя значения
    // канала Alpha рамки
    tempBitmapData.copyPixels(photoData,
                              borderData.rect,
                              new Point(0,0),
```

```
                                borderData,
                                new Point(0,0),
                                false);

    // Фильтр ColorMatrixFilter,
    // который увеличит яркость изображения
    var brightnessOffset:int = 70;
    var brightnessFilter:ColorMatrixFilter = new ColorMatrixFilter(
        new Array(1,0,0,0,brightnessOffset,
                  0,1,0,0,brightnessOffset,
                  0,0,1,0,brightnessOffset,
                  0,0,0,1,0));

    // Фильтр размытия, который делает изображение размытым
    var blurFilter:BlurFilter = new BlurFilter(1, 1);

    // Фильтр падающей тени, который создает эффект изображения,
    // наклеенного на бумагу
    var dropShadowFilter:DropShadowFilter = new DropShadowFilter(4, 35,
                    0x2E2305, .6, 5, 12, 4, BitmapFilterQuality.MEDIUM);

    // Определяем область, необходимую для отображения изображения
    // и его падающей тени
    var filteredImageRect:Rectangle = tempBitmapData.generateFilterRect(
                                  tempBitmapData.rect, dropShadowFilter);

    // Создаем объект BitmapData, который будет хранить
    // завершенное изображение
    var scrapbookImage:BitmapData =
                                  new BitmapData(filteredImageRect.width,
                                      filteredImageRect.height,
                                      true,
                                      0xFFFFFFFF);

    // Применяем фильтр ColorMatrixFilter, увеличивающий яркость
    tempBitmapData.applyFilter(tempBitmapData,
                               tempBitmapData.rect,
                               new Point(0,0),
                               brightnessFilter);

    // Применяем фильтр BlurFilter
    tempBitmapData.applyFilter(tempBitmapData,
                               tempBitmapData.rect,
                               new Point(0,0),
                               blurFilter);

    // Применяем фильтр DropShadowFilter
    scrapbookImage.applyFilter(tempBitmapData,
                               tempBitmapData.rect,
                               new Point(-filteredImageRect.x,
                                         -filteredImageRect.y),
                               dropShadowFilter);
```

```
            // Связываем объект BitmapData scrapbookImage с объектом Bitmap
            // для отображения на экране
            var b:Bitmap = new Bitmap(scrapbookImage);
            addChild(b);
            b.x = 100;
            b.y = 75;
        }
    }
}
```

На рис. 26.13 показаны результаты выполнения кода из листинга 26.12.

Рис. 26.13. Эффект старой фотографии, теперь с фильтрами

Освобождение памяти, занимаемой растровыми изображениями

Каждый пиксел в любом объекте `BitmapData` занимает небольшое количество системной памяти — 4 байта, если быть точным (один байт для каждого цветового канала). Хотя объем памяти, занимаемой каждым пикселом, сам по себе незначителен, в совокупности пикселы могут приводить к существенному расходованию памяти. Таким образом, чтобы уменьшить объем памяти, потребляемый средой Flash при работе с растровыми изображениями, каждая программа на языке ActionScript должна гарантировать, что все объекты `BitmapData`, когда в них отпадает необходимость, становятся доступными для сборки мусора.

Чтобы некоторый объект `BitmapData` стал доступен для сборки мусора, необходимо удалить все ссылки либо на него, либо на объекты, которые ссылаются на данный объект. В любом случае память, занимаемая объектом `BitmapData`, будет автоматически освобождена сборщиком мусора в следующем цикле сборки мусора.

Чтобы *немедленно* освободить память, занимаемую пикселами объекта `BitmapData` (вместо того чтобы ожидать освобождения памяти сборщиком мусора), используйте метод экземпляра `dispose( )` класса `BitmapData`. Например, следующий

код создает объект BitmapData, который занимает 1600 байт (ширина 20 пикселов × высота 20 пикселов × 4 байта на пиксел):

```
var imgData:BitmapData = new BitmapData(20, 20, false, 0xFF00FF00);
```

Чтобы немедленно освободить 1600 байт памяти, мы используем метод dispose( ), как показано в следующем коде:

```
imgData.dispose( );  // Освобождаем память, занимаемую объектом imgData
```

Метод dispose( ) освобождает память, в которой хранится информация о пикселах объекта imgData, но не освобождает память, занимаемую самим объектом imgData. Память, занимаемая объектом imgData, будет освобождена в соответствии с обычным циклом сборки мусора.

Как только над объектом BitmapData будет вызван метод dispose( ), данный объект окажется недоступным. Обращение к его методам и переменным приведет к генерации исключения ArgumentError.

Метод dispose( ) может быть полезен для управления памятью, потребляемой функциями или циклами, в которых используются временные растровые изображения, как, например, в случае создания изображения с применением фильтров, образующегося путем объединения нескольких временных изображений.

Слова, слова, слова

В двух предыдущих главах мы рассматривали методики создания изображений для отображения на экране. В следующей главе мы узнаем, как создавать текст для отображения на экране.

Отображение и ввод текста

Приложение Flash Player предоставляет расширенный, усовершенствованный интерфейс API для работы с текстом. В этой главе мы познакомимся с некоторыми из его ключевых возможностей — созданием и отображением текста, форматированием текста и обработкой текстового ввода.

Информация, представленная в этой главе, относится конкретно к приложению Flash Player (к автономной версии и к версии, реализованной в виде модуля расширения браузера). Однако в целом эта информация также применима к любой среде выполнения Flash, поддерживающей полнофункциональное отображение и ввод текста, например к приложению Adobe AIR. Стоит отметить, что, в отличие от Flash Player, приложение Adobe AIR обеспечивает полнофункциональную поддержку языка HTML и стилей CSS, аналогичную той, которая присутствует в таких браузерах, как Internet Explorer и Mozilla Firefox. При работе с другими средами выполнения Flash обязательно обращайтесь к соответствующей документации за информацией о поддержке текста.

Основным элементом API для работы с текстом приложения Flash Player является класс `TextField`, который обеспечивает управление текстом, отображаемым на экране.

В этой книге (и во многих источниках по языку ActionScript) термин «текстовое поле» в общем смысле обозначает некоторое текстовое поле на экране и его соответствующий экземпляр класса TextField языка ActionScript. Тем не менее фраза «объект TextField», в частности, обозначает объект языка ActionScript, который управляет текстовым полем.

Прежде чем приступить к созданию текста и работе с ним, бегло ознакомимся с основными классами API для работы с текстом приложения Flash Player, перечисленными в табл. 27.1. Набор возможностей API для работы с текстом можно разбить на следующие общие категории:

- управление текстом, отображаемым на экране;
- форматирование текста;
- установка параметров отображения текста для модуля FlashType;
- управление шрифтами (например, определение доступных шрифтов);
- получение метрик текста (характеристик);
- предоставление константных значений.

В табл. 27.1 приводится краткое описание каждого класса API для работы с текстом. Они разбиты на категории в соответствии с предыдущим списком общих категорий. Все классы API для работы с текстом находятся в пакете `flash.text`. Стоит отметить, что эта глава является хорошим введением в API для работы с текстом языка

ActionScript, но она не дает исчерпывающего описания всех возможностей данного интерфейса. Интерфейс API для работы с текстом просто огромен. Для дальнейшего изучения познакомьтесь с описанием пакета `flash.text` в справочнике по языку ActionScript корпорации Adobe. Кроме того, чтобы узнать о дополнительных и более специализированных возможностях управления текстом, рассмотрите применение компонентов `Label`, `Text`, `TextArea` и `TextInput`, предоставляемых платформой разработки Flex, или компонентов `Label`, `TextArea` и `TextInput`, входящих в состав среды разработки Flash.

Таблица 27.1. Обзор API для работы с текстом

Назначение	Класс	Описание
Управление текстом, отображаемым на экране	TextField	Представляет следующие типы текстовых полей. 1. Текстовые поля, создаваемые из кода на языке ActionScript. 2. Текстовые поля типа «динамический текст» или «вводимый текст», создаваемые в среде разработки Flash
	StaticText	Представляет текстовые поля типа static, создаваемые в среде разработки Flash
	TextSnapshot	Строка, содержащая текст из всех статических текстовых полей некоторого экземпляра класса DisplayObjectContainer
Форматирование текста	TextFormat	Простой класс данных, представляющий информацию о форматировании символов
	StyleSheet	Представляет таблицу стилей, содержащую информацию о форматировании символов. Данный класс реализован на основании спецификации Cascading style sheet Level 1 (CSS1) консорциума W3C
Установка параметров отображения текста для модуля FlashType	CSMSettings	Простой класс данных, используемый для предоставления модулю отображения текста FlashType приложения Flash Player заданных параметров сглаживания для отображения определенного шрифта определенного размера. Применяется вместе со статическим методом set AdvancedAntiAliasingTable( ) класса TextRenderer
	TextRenderer	Управляет настройками отображения для модуля FlashType приложения Flash Player
Управление шрифтами	Font	Предоставляет доступ к списку шрифтов, установленных в системе или внедренных в SWF-файлы, и регистрирует шрифты, загружаемые на этапе выполнения
Получение характеристик текста	TextLineMetrics	Описывает характеристики для одной строки текста в поле
Предоставление константных значений	AntiAliasType, FontStyle, FontType, GridFitType, TextColorType, TextDisplayMode, TextFieldAutoSize, TextFieldType, TextFormatAlign	Определяют константы, применяемые для указания различных значений переменных и параметров в API для работы с текстом. Дополнительные сведения можно получить в табл. 27.2

В табл. 27.2 представлен краткий обзор классов API для работы с текстом, которые просто обеспечивают доступ к специальным значениям через константы класса.

Таблица 27.2. Классы API для работы с текстом, содержащие константные значения

Назначение	Класс	Описание
Константы, используемые при выборе модуля отображения текста	AntiAliasType	Определяет константы, которые описывают типы сглаживания. Применяется вместе с переменной экземпляра antiAliasType класса TextField
Константы, применяемые при установке значений модуля отображения FlashType	FontStyle	Определяет константы, которые описывают вариации шрифта (например, полужирный, курсив). Применяется вместе со статическим методом setAdvancedAntiAliasingTable( ) класса TextRenderer и переменной экземпляра fontStyle класса Font
	GridFitType	Определяет константы, которые описывают типы подбора по сетке элементов растра. Используется с переменной экземпляра gridFitType класса TextField
	TextColorType	Определяет константы, которые описывают типы цвета текста (темный или светлый). Применяется вместе со статическим методом setAdvancedAntiAliasingTable( ) класса TextRenderer
	TextDisplayMode	Определяет константы, которые описывают типы межпикселного сглаживания. Применяется вместе со статической переменной displayMode класса TextRenderer
Константы, используемые при установке параметров текстовых полей	TextFieldAutoSize	Определяет константы, которые описывают параметры автоматического изменения размеров. Применяется вместе с переменной экземпляра autoSize класса TextField
	TextFieldType	Определяет константы, которые описывают типы текстовых полей (dynamic или input). Используется вместе с переменной экземпляра type класса TextField
Константы, используемые при получении списков шрифтов	FontType	Определяет константы, которые описывают типы местоположений шрифтов (системные или внедренные). Применяется вместе со статическим методом enumerateFonts( ) класса Font
Константы, используемые при установке выравнивания текста	TextFormatAlign	Определяет константы, которые описывают типы выравнивания текста (то есть по центру, по левому краю, по правому краю или по ширине). Применяется вместе с переменной экземпляра align класса TextFormat

Теперь, когда мы получили общее представление об инструментах, доступных в API для работы с текстом, создадим какой-нибудь текст!

Создание и отображение текста

Чтобы отобразить текст с помощью ActionScript, мы сначала создаем объект `TextField`. Этот объект представляет прямоугольный текстовый контейнер, который может отображаться на экране и заполняться форматированным текстом

через код или пользовательский ввод. Например, следующий код создает объект `TextField` и присваивает его переменной `t`:

```
var t:TextField = new TextField( );
```

После создания объекта `TextField` мы используем переменную экземпляра `text` класса `TextField`, чтобы указать текст для отображения на экране. Например, после выполнения следующего кода объект `t` будет выводить текст `Hello world`:

```
t.text = "Hello world";
```

Наконец, для того, чтобы показать текстовое поле на экране, мы передаем объект `TextField` в метод `addChild( )` или `addChildAt( )` любого объекта `DisplayObjectContainer`, который на настоящий момент находится в списке отображения. Например, если предположить, что объект *некийКонтейнер* находится в списке отображения, следующий код приведет к отображению текстового поля `t` на экране:

```
некийКонтейнер.addChild(t);
```

В листинге 27.1 предыдущий код представлен в контексте демонстрационного класса `HelloWorld`. Обратите внимание, что в этом коде мы импортируем класс `TextField` (вместе с остальными классами в пакете `flash.text`) перед его использованием.

Листинг 27.1. Отображение текста

```
package {
  import flash.display.*;
  import flash.text.*;  // Импортируем класс TextField и другие классы,
                        // размещенные в пакете flash.text

  public class HelloWorld extends Sprite {
    public function HelloWorld ( ) {
      // Создаем объект TextField
      var t:TextField = new TextField( );

      // Указываем текст для отображения
      t.text = "Hello world";

      // Добавляем объект TextField в список отображения
      addChild(t);
    }
  }
}
```

Результат выполнения кода из листинга 27.1 проиллюстрирован на рис. 27.1. Текст показан в том виде, в котором он по умолчанию отображается в операционной системе Windows XP (шрифты и форматирование будут рассмотрены далее в этой главе).

Hello world

Рис. 27.1. Текстовое поле

Хотя на предыдущем рисунке показан текст в текстовом поле, на нем не видна прямоугольная область отображения поля. По умолчанию она имеет размеры 100 пик-

селов по ширине и высоте. Рисунок 27.2 представляет измененную версию рис. 27.1, в которой изображена штриховая линия, представляющая прямоугольную область отображения текстового поля.

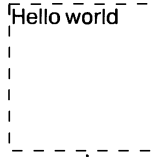

Рис. 27.2. Текстовое поле и его прямоугольная область отображения

Ширина и высота прямоугольной области отображения текстового поля могут быть указаны явно с помощью переменных экземпляра `width` и `height` класса `TextField`. Например, следующий код устанавливает для прямоугольной области отображения объекта `t` ширину, равную 200 пикселам, и высоту, равную 50 пикселам:

```
t.width = 200;
t.height = 50;
```

Рисунок 27.3 демонстрирует результат предыдущих изменений, снова используя штриховую линию, которая представляет прямоугольную область отображения текстового поля.

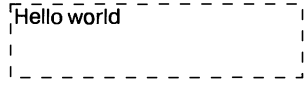

Рис. 27.3. Заданный размер для прямоугольной области отображения текстового поля

По умолчанию прямоугольная область отображения поля не показывается на экране. Тем не менее мы можем сделать ее видимой, присвоив одной или обеим переменным экземпляра `background` и `border` класса `TextField` значение `true`. Переменная `background` определяет, требуется ли заливка прямоугольной области отображения текстового поля сплошным цветом, а переменная `border` указывает, нужно ли показывать линию толщиной 1 пиксел вокруг прямоугольной области отображения текстового поля. Цвет фона и границы задаются путем присваивания 24-битных значений RGB-цвета переменным `backgroundColor` и `borderColor`.

В листинге 27.2 представлен наш обновленный класс `HelloWorld`, который делает видимой прямоугольную область отображения объекта `t` с помощью границы темно-серого цвета и фона светло-серого цвета. Новый код выделен полужирным шрифтом.

Листинг 27.2. Отображение текста с границей и фоном

```
package {
  import flash.display.*;
  import flash.text.*;  // Импортируем класс TextField и другие классы,
                        // размещенные в пакете flash.text

  public class HelloWorld extends Sprite {
    public function HelloWorld ( ) {
```

```
        var t:TextField = new TextField( );  // Создаем объект TextField
        t.text = "Hello world";              // Указываем текст
                                             // для отображения
        t.background = true;              // Включаем отображение фона
        t.backgroundColor = 0xCCCCCC;     // В качестве цвета фона указываем
                                          // светло-серый цвет
        t.border = true;                  // Включаем отображение границы
        t.borderColor = 0x333333;         // В качестве цвета границы указываем
                                          // темно-серый цвет

        addChild(t);  // Добавляем объект TextField в список отображения
    }
  }
}
```

Результат выполнения кода из листинга 27.2 показан на рис. 27.4.

Рис. 27.4. Текстовое поле с границей и фоном

Перенос слов

По умолчанию, когда ширина текста в текстовом поле оказывается больше ширины прямоугольной области отображения этого текстового поля, текст, выходящий за пределы данной области, теряется из виду. Например, следующий код в качестве текста объекта `t` присваивает строку, ширина которой превышает 100 пикселов при использовании шрифта по умолчанию в операционной системе Windows XP:

```
t.text = "Hello world, how are you?";
```

Результат продемонстрирован на рис. 27.5.

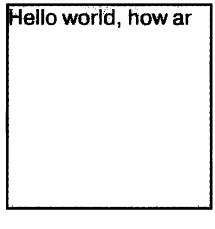

Рис. 27.5. Текст, выходящий за границы, не отображается на экране

Чтобы предотвратить скрытие текста, ширина которого оказывается больше ширины прямоугольной области отображения текстового поля, мы можем включить возможность автоматического добавления разрывов строк в текстовом поле, присвоив переменной экземпляра `wordWrap` класса `TextField` значение `true`. Когда

возможность автоматического добавления разрывов строк включена, происходит *мягкий перенос* длинных строк текста, то есть любая строка, ширина которой оказывается больше ширины прямоугольной области отображения данного текстового поля, будет автоматически перенесена на следующую строку. Например, следующий код включает возможность автоматического добавления разрывов строк для объекта `TextField t`:

```
t.wordWrap = true;
```

Результат продемонстрирован на рис. 27.6.

Рис. 27.6. Автоматическое добавление разрывов строк

Перенос слов является всего лишь свойством отображения. В тех местах, где переносится исходный текст, никакие символы возврата каретки или новой строки не добавляются. Если переменной `wordWrap` присвоено значение `true`, изменение значения переменной `width` текстового поля приведет к тому, что мягкий перенос будет происходить в другом месте (то есть изменится размещение текста). Жесткие переносы могут быть добавлены в текстовое поле с помощью служебной последовательности `"\n"` или тега `<BR>` в тексте HTML (использование разметки HTML в текстовых полях мы рассмотрим далее, в разд. «HTML-форматирование текста»).

Не путайте переменную wordWrap с переменной multiline (рассматривается далее, в разд. «HTML-форматирование текста» и «Ввод через текстовые поля»). Переменная multiline влияет на возможность разметки HTML и пользовательского ввода вызывать разрывы строк, а переменная wordWrap определяет, должна ли среда выполнения Flash осуществлять автоматический перенос строк.

Автоматическое изменение размеров

Чтобы прямоугольная область отображения текстового поля автоматически изменяла свои размеры в соответствии с размером текста в текстовом поле, мы используем переменную экземпляра `autoSize` класса `TextField`. Если переменной `autoSize` присвоить любое значение, отличное от значения `TextFieldAutoSize.NONE` (установлено по умолчанию), размер текстового поля всегда будет достаточным для отображения текста, присваиваемого этому полю.

Переменная autoSize перекрывает любые абсолютные размеры, задаваемые через переменные height и width объекта TextField.

Переменная `autoSize` может принимать одно из следующих четырех возможных значений: `TextFieldAutoSize.NONE`, `TextFieldAutoSize.LEFT`, `TextFieldAutoSize.RIGHT` и `TextFieldAutoSize.CENTER`. Они определяют направление, в котором текстовое поле должно расширяться или сужаться, чтобы соответствовать размерам присвоенного ему текста.

Если переменной `autoSize` присвоено значение `NONE`, то размеры текстового поля остаются фиксированными. Если переменной присвоено значение `LEFT`, то левая граница текстового поля фиксируется, а правая — перемещается. Если переменной присвоено значение `RIGHT`, то правая граница текстового поля фиксируется, а левая — перемещается. Если же переменной присвоено значение `CENTER`, то происходит равномерное перемещение левой и правой границ текстового поля. В последних трех случаях, когда происходит перенос строк или встречаются твердые разрывы строк, нижняя граница текстового поля также перемещается для размещения нескольких строк текста.

В листинге 27.3 создается текстовое поле, прямоугольная область отображения которого изменяет свои размеры справа и, когда встречается разрыв строки, снизу.

Листинг 27.3. Перемещаемые правая и нижняя границы

```
package {
  import flash.display.*;
  import flash.text.*;

  public class HelloWorld extends Sprite {
    public function HelloWorld ( ) {
      var t:TextField = new TextField( );
      t.text = "Hello world, how are you?";
      t.background = true;
      t.backgroundColor = 0xCCCCCC;
      t.border = true;
      t.borderColor = 0x333333;

      // Прямоугольная область отображения объекта t
      // будет автоматически изменять свои размеры,
      // чтобы вместить значение переменной t.text.
      t.autoSize = TextFieldAutoSize.LEFT;

      addChild(t);
    }
  }
}
```

Результат выполнения кода из листинга 27.3 показан на рис. 27.7. Сравните его с рис. 27.5, на котором было изображено текстовое поле, не использующее ни возможность автоматического изменения размеров, ни возможность переноса слов.

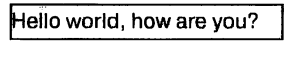

Рис. 27.7. Перемещаемые правая и нижняя границы

Теперь предположим, что мы добавили разрыв строки в текст объекта `t`, как показано в следующем коде (разрыв строки вставляется с помощью последовательности символов `"\n"`):

```
t.text = "Hello world." + "\n" + "How are you?";
```

На рис. 27.8 показано результирующее текстовое поле. Обратите внимание, что среда выполнения Flash автоматически изменяет размеры прямоугольной области отображения поля, чтобы полностью вместить текст, путем перемещения ее левой и нижней границ.

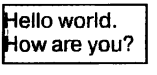

Рис. 27.8. Перемещаемые правая и нижняя границы, с разрывом строки

Если переменной `wordWrap` присвоить значение `true`, а переменной `autoSize` — любое значение, отличное от значения `TextFieldAutoSize.NONE`, нижняя граница текстового поля будет автоматически перемещаться вниз или вверх, но при этом его левая, правая и верхняя границы будут оставаться в исходном положении. Этот код продемонстрирован в листинге 27.4 — в нем создается текстовое поле с перемещаемой нижней границей.

Листинг 27.4. Перемещаемая нижняя граница

```
package {
  import flash.display.*;
  import flash.text.*;

  public class HelloWorld extends Sprite {
    public function HelloWorld ( ) {
      var t:TextField = new TextField( );
      t.text = "Hello world, how are you?";
      t.background = true;
      t.backgroundColor = 0xCCCCCC;
      t.border = true;
      t.borderColor = 0x333333;

      // Вместе две следующие строки кода делают нижнюю границу объекта t
      // автоматически перемещаемой для размещения значения переменной
      // t.text.
      t.autoSize = TextFieldAutoSize.LEFT;
      t.wordWrap = true;

      addChild(t);
    }
  }
}
```

Результат выполнения кода из листинга 27.4 показан на рис. 27.9. Обратите внимание, что ширина прямоугольного отображаемого контейнера зафиксирована и составляет 100 пикселов (по умолчанию), а для размещения текста с переносами перемещается нижняя граница.

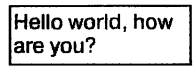

Рис. 27.9. Перемещаемая нижняя граница

Чтобы создать текстовое поле с перемещаемой нижней границей и фиксированной шириной, присвойте переменной autoSize любое значение, отличное от значения TextFieldAutoSize.NONE, а переменной wordWrap присвойте значение true.

Встраиваемые шрифты для отображения повернутого, искаженного и прозрачного текста

По умолчанию приложение Flash Player не отображает повернутые или искаженные текстовые поля на экране. Например, если мы хотим добавить следующее текстовое поле в список отображения, текст «Hello world» не появится на экране, поскольку поле повернуто:

```
var t:TextField = new TextField( );
t.text = "Hello world";
t.rotation = 30;  // Поворачиваем текст
```

Подобным образом, если мы захотим добавить следующее текстовое поле в список отображения, текст «Hello world» не появится на экране, поскольку поле искажено:

```
var t:TextField = new TextField( );
t.text = "Hello world";
t.transform.matrix = new Matrix(1, 1, 0, 1);  // Искажаем текст
```

Кроме того, по умолчанию Flash Player для отображения всех текстовых полей использует полную непрозрачность, даже если для них устанавливается уровень прозрачности через переменную экземпляра `alpha` класса `TextField`. Например, если бы мы хотели добавить следующее текстовое поле в список отображения, на экране текст «Hello world» оказался бы полностью непрозрачным, хотя уровню канала Alpha данного поля установлено значение 20 %:

```
var t:TextField = new TextField( );
t.text = "Hello world";
t.alpha = .2;
```

Приложение Flash Player правильно отображает только те повернутые, искаженные и прозрачные текстовые поля, которые используют встраиваемые шрифты. Информацию о визуализации текста с использованием встраиваемых шрифтов можно найти далее, в разд. «Шрифты и отображение текста».

Изменение содержимого текстового поля

После того как текстовому полю будет присвоено текстовое содержимое, это содержимое можно изменить через переменную `text`. Например, следующий код создает объект `TextField` и присваивает его текстовому содержимому строку `"Hello"`:

```
var t:TextField = new TextField( );
t.text = "Hello";
```

Следующий код полностью заменяет текст объекта `t` строкой `"Goodbye"`:

```
t.text = "Goodbye";
```

Чтобы добавить новый текст к существующему содержимому текстового поля (вместо того чтобы полностью заменять текст в текстовом поле), мы используем либо метод экземпляра `appendText( )` класса `TextField`, либо оператор +=. Например, следующий код добавляет строку `"...hope to see you again!"` к тексту `"Goodbye"`:

```
t.appendText("...hope to see you again!");
```

После выполнения предыдущей строки кода переменная `t.text` будет иметь такое значение:

```
"Goodbye...hope to see you again!"
```

Для добавленного текста используется форматирование последнего символа в текстовом поле, а любой существующий текст сохраняет свое первоначальное форматирование. Если в момент вызова функции `appendText( )` текстовое поле не содержит никакого текста, добавляемый текст будет отформатирован в соответствии с текстовым форматом, используемым по умолчанию для данного текстового поля.

Информацию по форматированию текста и текстовому формату, используемому по умолчанию, можно найти в следующем разд. «Форматирование текстовых полей».

Следующий код, как и предыдущий, добавляет новый текст в конец существующего текста в текстовом поле, но делает это не с помощью метода `appendText( )`, а с помощью оператора +=:

```
t.text += " Come again soon.";
```

В отличие от метода `appendText( )`, оператор += сбрасывает форматирование для *всего* текста в поле, устанавливая формат, используемый по умолчанию. Кроме того, оператор гораздо медленнее метода, поэтому следует избегать его использования.

Чтобы заменить некоторую последовательность символов в текстовом поле новой последовательностью, мы используем метод экземпляра `replaceText( )` класса `TextField`, который имеет следующий обобщенный вид:

объектTextField`.replaceText(`*индексНачала*`, `*индексКонца*`, `*новыйТекст*`)`

Метод `replaceText( )` удаляет символы в объекте *объектTextField*, начиная с индекса *индексНачала* и заканчивая индексом *индексКонца*-1, и заменяет их текстом *новыйТекст*. Новое объединенное значение сохраняется в переменной *объектTextField*`.text`.

Например, следующий код заменяет символы `"bcd"` в тексте `"abcde"` новым текстом `"x"`:

```
var t:TextField = new TextField( );

t.text = "abcde";
t.replaceText(1, 4, "x");
trace(t.text);  // Выводит: axe
```

Если значения *индексНачала* и *индексКонца* равны, то строка *новыйТекст* вставляется непосредственно перед указанным индексом *индексНачала*. Например, следующий код вставляет символ "s" непосредственно перед символом "t":

```
var t:TextField = new TextField( );
t.text = "mat";
t.replaceText(2, 2, "s");
trace(t.text);  // Выводит: mast
```

В оставшейся части этого раздела рассматриваются вопросы форматирования, относящиеся к методу replaceText( ), и для чтения последующего материала необходимо понимание методик форматирования текста, рассматриваемых в следующем разделе.

Форматирование текста, вставляемого через метод replaceText( ), зависит от указываемых значений для параметров *индексНачала* и *индексКонца*. Если значения отличаются, то вставляемый текст использует форматирование символа, следующего за вставляемым текстом (то есть символа в позиции *индексКонца*). Любой существующий текст сохраняет свое первоначальное форматирование. Например, рассмотрим следующий код, который создает текстовое поле, отображающее слово "lunchtime", при этом символы "time" отформатированы с использованием полужирного шрифта:

```
var boldFormat:TextFormat = new TextFormat( );
boldFormat.bold = true;
var t:TextField = new TextField( );
t.text = "lunchtime";
t.setTextFormat(boldFormat, 5, 9);  // Выделяем слово "time"
                                    // полужирным шрифтом
```

Результатом выполнения этого кода является строка:

lunch**time**

Теперь мы воспользуемся методом replaceText( ), чтобы заменить слово "lunch" словом "dinner", как показано ниже:

```
t.replaceText(0, 5, "dinner");    ` // Заменяем слово "lunch" словом "dinner"
```

В итоге слово "dinner" будет отформатировано с использованием полужирного шрифта, что соответствует форматированию символа в позиции *индексКонца* ("t"). Результат выглядит следующим образом:

dinnertime

Чтобы вместо существующего формата текстового поля использовать новый формат для вставляемого текста, мы присваиваем новый текст с помощью метода replaceText( ), а затем сразу же присваиваем желаемый формат этому тексту. Например, следующий код снова заменяет слово "lunch" словом "dinner", но на этот раз к новому добавленному тексту также применяется форматирование:

```
t.replaceText(0, 5, "dinner");      // Заменяем слово "lunch" словом "dinner"
var regularFormat:TextFormat = new TextFormat( );
regularFormat.bold = false;
t.setTextFormat(regularFormat, 0, 6);  // Отменяем форматирование полужирным
                                       // шрифтом слова dinner
```

Результат выполнения предыдущего кода выглядит следующим образом:

```
dinnertime
```

Когда оба аргумента *индексНачала* и *индексКонца* метода `replaceText( )` равны `0`, текст вставляется в начало текстового поля, а для форматирования вставляемого текста применяется используемый по умолчанию формат данного поля. Первоначальное форматирование любого существующего текста сохраняется.

Когда аргументы *индексНачала* и *индексКонца* равны, и при этом их значения больше `0`, для форматирования вставляемого текста используется формат символа, непосредственно предшествующего вставляемому тексту (то есть символа в позиции *индексКонца*-1). Первоначальное форматирование любого существующего текста сохраняется. Например, следующий код снова создает текстовое поле, которое отображает слово `"lunchtime"`, применяя полужирный шрифт для слова `"time"`:

```
var boldFormat:TextFormat = new TextFormat( );
boldFormat.bold = true;
var t:TextField = new TextField( );
t.text = "lunchtime";
t.setTextFormat(boldFormat, 5, 9);  // Выделяем слово "time"
                                    // полужирным шрифтом
```

На этот раз мы вставляем текст `"break"` сразу после символа `"t"`:

```
t.replaceText(5, 5, "break");   // Вставляем слово "break" после символа "t"
```

Поскольку символ в позиции *индексКонца*-1 (`"h"`) не отформатирован с использованием полужирного шрифта, для слова `"break"` полужирный шрифт применяться не будет, а результат выполнения предыдущего кода будет выглядеть следующим образом:

```
lunchbreaktime
```

Теперь ближе познакомимся с методиками форматирования текста.

Форматирование текстовых полей

Язык ActionScript предоставляет три различных инструмента для форматирования текста: класс `flash.text.TextFormat`, разметку HTML и класс `flash.text.StyleSheet`. Все три инструмента позволяют управлять следующими вариантами форматирования абзацев и символов, но используют различный синтаксис.

Форматирование на уровне абзацев — выравнивание, отступы, маркеры, высота строки (интервал между строками), шаг табуляции.

Форматирование на уровне символов — гарнитура, размер, вес шрифта (полужирный или обычный), цвет, стиль шрифта (курсив или обычный), кернинг, расстояние между буквами (трекинг), подчеркивание текста, гипертекстовые ссылки.

Форматирование на уровне абзацев применяется ко всему абзацу, при этом абзац определяется как часть текста, ограниченная разрывами строки (`\n`, `<BR>` или `<P>`). В отличие от этого, форматирование на уровне символов применяется

к произвольным последовательностям отдельных символов, ограниченным индексами в тексте или тегами HTML или XML.

Класс `TextFormat` предоставляет точное программное управление над форматированием текста и обычно используется для динамической генерации текстового вывода. Класс `StyleSheet` помогает отделить инструкции форматирования от содержимого, к которому применяется данное форматирование, и обычно применяется при форматировании больших блоков содержимого HTML или XML. Инструкции форматирования языка HTML предлагают простой, интуитивно понятный способ форматирования текста, но загрязняют текстовое содержимое разметкой. Разметка HTML обычно используется в тех случаях, когда удобство важнее гибкости, как, возможно, при форматировании текста в прототипе приложения или форматировании небольших фрагментов текста, которые гарантированно не будут изменяться в течение проекта.

Класс TextFormat является полностью совместимым и взаимозаменяемым с инструкциями форматирования языка HTML. Однако класс StyleSheet не совместим ни с классом TextFormat, ни с инструкциями форматирования языка HTML. Текстовые поля, использующие таблицы стилей, могут быть отформатированы только с помощью экземпляров класса StyleSheet.

В следующих разделах рассматриваются вопросы общего использования объектов `TextFormat` и `StyleSheet`, а также разметки HTML. Каждый раздел содержит примеры распространенных операций форматирования. Подробное описание каждого отдельного параметра форматирования можно найти в следующих разделах справочника по языку ActionScript корпорации Adobe:

- класс `TextFormat`;
- переменная экземпляра `htmlText` класса `TextField`;
- класс `StyleSheet`.

Присваивание значения переменной text объекта TextField приводит к удалению любого пользовательского форматирования, связанного с этим полем. Чтобы добавить текст в поле, сохранив его текущее форматирование, используйте метод экземпляра replaceText( ) класса TextField.

Форматирование текста с помощью класса TextFormat

Общий процесс форматирования текста с использованием класса `TextFormat` заключается в следующем.

1. Создать объект `TextFormat`.
2. Установить переменные объекта `TextFormat`, отражающие желаемое форматирование.
3. Применить форматирование к одному символу или более, используя метод экземпляра `setTextFormat( )` класса `TextField`.

Применим перечисленные шаги на примере. Наша цель — отформатировать весь текст в поле, используя шрифт Arial размером 20 пунктов с полужирным начертанием.

В языке ActionScript все размеры шрифтов указываются в пикселах. Если шрифт имеет размер 20 пунктов, это интерпретируется приложением Flash Player как 20 пикселов.

Мы начнем работу над нашим кодом форматирования с создания текстового поля, размеры которого будут изменяться автоматически, чтобы соответствовать размерам форматируемого текста:

```
var t:TextField = new TextField( );
t.text = "ActionScript is fun!";
t.autoSize = TextFieldAutoSize.LEFT;
```

Далее создадим объект `TextFormat`:

```
var format:TextFormat = new TextFormat( );
```

Затем присвоим переменным `font`, `size` и `bold` созданного объекта `TextFormat` желаемые значения: `"Arial"`, `20` и `true`, как показано в следующем коде:

```
format.font = "Arial";
format.size = 20;
format.bold = true;
```

Вместе переменные объекта `TextFormat` описывают стиль форматирования, который может быть применен к некоторому символу или последовательности символов. Доступные переменные перечислены в следующем подразд. «Доступные переменные класса TextFormat».

После создания объекта `TextFormat` и присваивания значений его переменным мы можем применять его для форматирования некоторого символа или последовательности символов с помощью метода `setTextFormat( )`, который имеет следующий вид:

объектTextField.setTextFormat(*объектTextFormat*, *индексНачала*, *индексКонца*)

В приведенном обобщенном коде *объектTextField* — это текстовое поле, текст которого будет отформатирован, а *объектTextFormat* — объект `TextFormat`, содержащий инструкции форматирования. Параметр *индексНачала* — это необязательное целое число, обозначающее индекс первого символа, который будет отформатирован с помощью объекта *объектTextFormat*. Параметр *индексКонца* — это необязательное целое число, обозначающее индекс символа, следующего за последним символом, который будет отформатирован с помощью объекта *объектTextFormat*.

Когда указаны оба аргумента *индексНачала* и *индексКонца*, метод `setTextFormat( )` форматирует последовательность символов, начиная с индекса *индексНачала* и заканчивая индексом *индексКонца*-1, в соответствии со значениями переменных объекта *объектTextFormat*. Когда аргумент *индексНачала* указан, а аргумент *индексКонца* — нет, метод `setTextFormat( )` форматирует один символ с индексом *индексНачала* в соответствии со значениями переменных объекта *объектTextFormat*. Когда не указан ни аргумент *индексНачала*, ни аргумент *индексКонца*, метод `setTextFormat( )` форматирует все символы в объекте *объектTextField* в соответствии со значениями переменных объекта *объектTextFormat*. Любая переменная объекта *объектTextFormat*, которой присвоено значение `null`, не влияет на форматирование целевой последовательности

символов (существующее форматирование, определяемое данной переменной, сохраняется).

Попробуем использовать метод `setTextFormat( )`, чтобы отформатировать символы в нашем тестовом текстовом поле `t`. Вот этот код:

```
t.setTextFormat(format);
```

Для обзора в листинге 27.5 приведен весь код, необходимый для форматирования всех символов в текстовом поле с использованием шрифта Arial размером 20 пунктов полужирного начертания.

Листинг 27.5. Форматирование текстового поля

```
// Создаем текстовое поле
var t:TextField = new TextField( );
t.text = "ActionScript is fun!";
t.autoSize = TextFieldAutoSize.LEFT;

// Создаем объект TextFormat и устанавливаем значения его переменных
var format:TextFormat = new TextFormat( );
format.font = "Arial";
format.size = 20;
format.bold = true;

// Применяем форматирование
t.setTextFormat(format);
```

Результат выполнения кода из листинга 27.5 проиллюстрирован на рис. 27.10.

ActionScript is fun!

Рис. 27.10. Текст, отформатированный с помощью объекта TextFormat

Теперь предположим, что мы хотим отформатировать весь текст в поле с помощью шрифта Arial размером 20 пунктов, но при этом хотим, чтобы полужирным шрифтом было выделено только слово `fun`. Нам потребуется два объекта `TextFormat`: один для общих настроек шрифта и один для шрифта полужирного начертания. Этот код показан в листинге 27.6 (обратите внимание на использование аргументов *индексНачала* и *индексКонца* во втором вызове метода `setTextFormat( )`):

Листинг 27.6. Два формата

```
// Создаем текстовое поле
var t:TextField = new TextField( );
t.text = "ActionScript is fun!";
t.autoSize = TextFieldAutoSize.LEFT;

// Создаем объект TextFormat для общих настроек шрифта
var fontFormat:TextFormat = new TextFormat( );
fontFormat.font = "Arial";
fontFormat.size = 20;

// Создаем объект TextFormat для настроек шрифта полужирного начертания
var boldFormat:TextFormat = new TextFormat( );
```

```
boldFormat.bold = true;

// Применяем общие настройки шрифта ко всему текстовому полю
t.setTextFormat(fontFormat);

// Применяем настройки шрифта полужирного начертания только к слову fun
t.setTextFormat(boldFormat, 16, 19);
```

Стоит отметить, что последовательное форматирование не является деструктивным; изменения в форматировании касаются только установленных переменных, поэтому во втором вызове метода `setTextFormat( )` слово `fun` сохраняет свою гарнитуру шрифта (Arial) и размер (20) и приобретает полужирное начертание.

Теперь, когда известно, как применять форматирование, бегло рассмотрим полный список параметров форматирования, доступных через класс `TextFormat`.

Доступные переменные класса TextFormat

В табл. 27.3 и 27.4 перечислены переменные класса `TextFormat`, применяемые для форматирования на уровне символов и абзацев. Подробные сведения о каждой переменной можно найти в описании класса `TextFormat` в справочнике по языку ActionScript корпорации Adobe.

В табл. 27.3 перечислены переменные класса `TextFormat`, используемые для установки параметров форматирования на уровне символов.

Таблица 27.3. Переменные класса TextFormat для форматирования на уровне символов

Переменная	Описание
bold	Тип Boolean. Задает отображение символа с использованием полужирного начертания
color	Указывает цвет символа в виде 24-битного целого числа (например, 0xFF0000)
font	Задает шрифт
italic	Тип Boolean. Определяет отображение символа с использованием курсива
kerning	Тип Boolean. Задает, нужно ли использовать автоматический кернинг для пар символов
letterSpacing	Указывает расстояние между буквами (трекинг), в пикселах
size	Задает размер шрифта, в пунктах (1/72 дюйма)
target	Определяет окно или фрейм для гипертекстовой ссылки
underline	Тип Boolean. Задает отображение символа с использованием подчеркивания
url	Определяет гипертекстовую ссылку

В табл. 27.4 перечислены переменные класса `TextFormat`, используемые для установки параметров форматирования на уровне абзацев.

Таблица 27.4. Переменные класса TextFormat для форматирования на уровне абзацев

Переменная	Описание
align	Задает выравнивание абзаца по горизонтали (по левому или правому краю, по центру или по ширине) с помощью одной из констант класса TextFormatAlign
blockIndent	Определяет расстояние в пикселах, представляющее смещение абзаца от левой границы текстового поля

Продолжение

Таблица 27.4 (продолжение)

Переменная	Описание
bullet	Указывает, нужно ли добавлять маркеры к абзацам
indent	Задает расстояние в пикселах, представляющее отступ первой строки абзаца от левой границы текстового поля
leading	Указывает величину вертикального промежутка в пикселах между строками текста
leftMargin	Определяет расстояние по горизонтали в пикселах между левой границей текстового поля и левым краем абзаца
rightMargin	Задает расстояние по горизонтали в пикселах между правой границей текстового поля и правым краем абзаца
tabStops	Задает шаги табуляции по горизонтали в пикселах

Из всех переменных, перечисленных в двух предыдущих таблицах, особого внимания заслуживает переменная экземпляра `font` класса `TextFormat`. Ее значение определяет название шрифта в виде строки. Как мы узнаем далее, в разд. «Шрифты и отображение текста», у разработчиков есть возможность заставить приложение Flash Player отображать текст с помощью шрифтов, установленных в локальной системе конечного пользователя (называемых *шрифтами устройства*), или шрифтов, включенных в SWF-файл (называемых *встраиваемыми*). Таким образом, переменная `font` должна определять либо название шрифта в локальной системе, либо название шрифта, встраиваемого в SWF-файл.

Специально для шрифтов устройств язык ActionScript также предлагает три особых названия — `"_sans"`, `"_serif"` и `"_typewriter"`, которые могут применяться в тех случаях, когда текст должен отображаться шрифтом без засечек, с засечками или моноширинным, используемым по умолчанию в локальной системе. Например, если при использовании шрифтов устройства в качестве названия шрифта указать значение `"_sans"`, в США на компьютере с установленной операционной системой Windows XP текст будет отображен шрифтом Arial, а на компьютере с установленной операционной системой Mac OS X — шрифтом Helvetica.

Мы узнаем больше об особенностях выбора названий шрифтов далее в этой главе. В дальнейшем будем полагать, что текст отображается с помощью шрифтов, установленных в локальной системе конечного пользователя (шрифтов устройства), и будем использовать названия шрифтов, поставляемых с операционной системой Windows XP.

Предупреждение насчет встраиваемых шрифтов

Стоит отметить, что, когда текст, для форматирования которого применяется полужирное начертание или курсив, отображается с помощью встраиваемых шрифтов, вариации соответствующего шрифта (-ов) полужирного начертания и курсива должны быть доступны для Flash Player в виде встраиваемых шрифтов. Например, рассмотрим следующий код, который создает текстовое поле, содержащее слова «hello» и «world», при этом слово «hello» отформатировано с использованием шрифта Courier New, а слово «world» — с использованием шрифта Courier New полужирного начертания:

```
// Создаем текстовое поле
var t:TextField = new TextField( );
t.text = "hello world";
```

```
// Создаем объекты TextFormat
var fontFormat:TextFormat = new TextFormat( );
fontFormat.font = "Courier New";
var boldFormat:TextFormat = new TextFormat( );
boldFormat.bold = true;

// Применяем форматирование
t.setTextFormat(fontFormat, 0, 11);
t.setTextFormat(boldFormat, 6, 11);
```

Чтобы отобразить предыдущее текстовое поле с помощью встраиваемых шрифтов, приложение Flash Player должно иметь доступ к встраиваемым версиям шрифтов Courier New и Courier New Bold. Дополнительную информацию можно найти далее, в подразд. «Форматирование текста с помощью встраиваемого шрифта» разд. «Шрифты и отображение текста».

Вызов метода setTextFormat( ) не оказывает влияния на текст, присваиваемый в дальнейшем

Метод `setTextFormat( )` может быть использован для форматирования текста в текстовом поле только *после* того, как этот текст будет добавлен в текстовое поле. Например, в следующем коде мы по ошибке вызываем метод `setTextFormat( )` до того, как будет присвоен текст, который мы хотим отформатировать:

```
// Создаем текстовое поле
var t:TextField = new TextField( );
t.autoSize = TextFieldAutoSize.LEFT;

// Создаем объект TextFormat и устанавливаем значения его переменных
var format:TextFormat = new TextFormat( );
format.font = "Arial";
format.size = 20;
format.bold = true;


// Применяем форматирование
t.setTextFormat(format);

// Присваиваем текст
t.text = "ActionScript is fun!";
```

Когда в предыдущем коде происходит вызов метода `setTextFormat( )`, объект `t` еще не содержит никакого текста, поэтому попытка применить форматирование оказывается безрезультатной. Вот правильный код:

```
// Создаем текстовое поле
var t:TextField = new TextField( );
t.autoSize = TextFieldAutoSize.LEFT;

// Создаем объект TextFormat и устанавливаем
// значения его переменных
var format:TextFormat = new TextFormat( );
format.font = "Arial";
```

```
format.size = 20;
format.bold = true;

// Присваиваем текст
t.text = "ActionScript is fun!";

// Применяем форматирование
t.setTextFormat(format);
```

Выполняя форматирование текста с помощью метода setTextFormat( ), всегда присваивайте текст перед вызовом этого метода.

Информацию о применении форматирования к текстовому полю до того, как будет присвоен текст, можно найти далее, в подразд. «Форматирование по умолчанию для текстовых полей».

Применение форматирования на уровне абзацев

Чтобы применить любой из вариантов форматирования на уровне абзацев, перечисленных ранее в табл. 27.4, мы должны применить желаемый формат к первому символу в абзаце (напомним, что в языке ActionScript абзац определяется как часть текста, ограниченная разрывами строки).

Например, рассмотрим следующий код, который сначала создает текстовое поле с двумя абзацами, а затем — объект `TextFormat`, задающий вариант форматирования на уровне абзацев — выравнивание по центру:

```
// Создаем текстовое поле
var t:TextField = new TextField( );
t.width = 300;
t.border = true;
// Абзацы разделяются одним разрывом строки (представляемым служебной
// последовательностью "\n")
t.text = "This is paragraph one.\nThis is paragraph two.";

// Создаем объект TextFormat
var alignFormat:TextFormat = new TextFormat( );
alignFormat.align = TextFormatAlign.CENTER;
```

Чтобы установить выравнивание только для первого абзаца в текстовом поле, мы применяем форматирование к первому символу первого абзаца, который находится в позиции с индексом 0:

```
t.setTextFormat(alignFormat, 0);
```

Для установки выравнивания только второму абзацу мы применяем форматирование к первому символу второго абзаца, который находится в позиции с индексом 23:

```
t.setTextFormat(alignFormat, 23);
```

Чтобы установить выравнивание сразу для нескольких абзацев, мы применяем форматирование с помощью аргументов *индексНачала* и *индексКонца*, диапазон значений которых включает позиции символов желаемых абзацев:

```
t.setTextFormat(alignFormat, 0, 24);
```

Если включена возможность переноса строк и какой-нибудь абзац переносится на следующую строку, указанный диапазон форматирования должен включать позицию первого символа следующей строки. В противном случае форматирование не будет применено к перенесенной строке. Таким образом, чтобы добиться наилучших результатов, применяя форматирование на уровне абзацев к абзацу в текстовом поле с включенной возможностью переноса, всегда применяйте форматирование ко всей последовательности символов в абзаце.

Для динамического определения начального и конечного индексов символов в абзаце используйте методы экземпляра `getFirstCharInParagraph( )` и `getParagraphLength( )` класса `TextField`. Например, следующий код применяет метод `getParagraphLength( )`, чтобы динамически определить начальные и конечные индексы двух абзацев в текстовом поле из предыдущего примера. После этого данный код использует полученные индексы для применения форматирования — выравнивания текста — ко всей последовательности символов во втором абзаце.

```
// Создаем текстовое поле
var t:TextField = new TextField( );
t.width = 100;
t.border = true;
t.wordWrap = true;
t.text = "This is paragraph one.\nThis is paragraph two.";

// Создаем объект TextFormat
var alignFormat:TextFormat = new TextFormat( );
alignFormat.align = TextFormatAlign.CENTER;

// Определяем начальные и конечные индексы абзацев
var firstParagraphStart:int = 0;
var firstParagraphEnd:int = t.getParagraphLength(firstParagraphStart)-1;
var secondParagraphStart:int = firstParagraphEnd+1;
var secondParagraphEnd:int = secondParagraphStart
                           + t.getParagraphLength(secondParagraphStart)-1;

// Применяем форматирование
t.setTextFormat(alignFormat, secondParagraphStart, secondParagraphEnd);
```

Получение информации о форматировании для последовательности символов

Чтобы получить информацию о существующем форматировании для одного или более символов, находящихся в текстовом поле, мы используем метод экземпляра `getTextFormat( )` класса `TextField`. Этот метод возвращает объект `TextFormat`, переменные которого описывают форматирование указанных символов. В общем виде метод `getTextFormat( )` записывается следующим образом:

объектTextField.getTextFormat(*индексНачала*, *индексКонца*)

Если при вызове этого метода указывается один целочисленный аргумент или значение аргумента *индексКонца* равно значению выражения *индексНачала*+1, то возвращаемый объект `TextFormat` будет отражать форматирование для одного символа,

находящегося в позиции *индексНачала*. Например, в следующем коде мы применяем форматирование к первым четырем символам текстового поля, а затем проверяем название шрифта для первого символа:

```
// Создаем текстовое поле
var t:TextField = new TextField( );
t.width = 100;
t.border = true;
t.wordWrap = true;
t.text = "What time is it?";

// Создаем объект TextFormat, переменной font которого присваивается
// значение "Arial"
var arialFormat:TextFormat = new TextFormat( );
arialFormat.font = "Arial";

// Применяем форматирование к слову 'What', индексы символов –
// от 0 до 3 (включительно)
t.setTextFormat(arialFormat, 0, 4);

// Получаем объект TextFormat для первого символа
var firstCharFormat:TextFormat = t.getTextFormat(0);

// Проверяем значение переменной font
trace(firstCharFormat.font); // Выводит: Arial
```

Если при вызове метода `getTextFormat( )` указываются два целочисленных аргумента, то возвращаемый объект `TextFormat` представляет форматирование для последовательности символов, начиная с позиции *индексНачала* и заканчивая позицией *индексКонца*-1. Если же метод `getTextFormat( )` вызывается без аргументов, то возвращаемый объект `TextFormat` представляет форматирование для всех символов в этом поле.

Если определенный параметр форматирования (например, шрифт, полужирное начертание или курсив) не является одинаковым для всех символов в указанной последовательности, то значение соответствующей переменной объекта `TextFormat` для этой последовательности будет равно `null`. Возвращаясь к нашему примеру, если получить объект `TextFormat` для всего текста объекта `t`, то мы обнаружим, что переменные, которые обозначают варианты форматирования, используемые всеми символами, содержат ненулевые значения:

```
// Получаем объект TextFormat для всех символов в объекте t
var allCharsFormat:TextFormat = t.getTextFormat( );

// Теперь проверим, имеют ли все символы полужирное начертание
trace(allCharsFormat.bold); // Выводит: false
```

Однако переменные, обозначающие варианты форматирования, которые отличаются от символа к символу, содержат значение `null`:

```
// Проверяем шрифт для всех символов
trace(allCharsFormat.font); // Выводит: null (шрифт не единообразен)
```

Шрифт первых четырех символов в объекте `t` отличается от шрифта остальных символов, стало быть, не существует такого значения переменной `font`, которое

могло бы точно описать всю последовательность целиком. В связи с этим переменной `font` присваивается значение `null`.

Отметим, что любые изменения в объекте `TextFormat`, возвращаемом методом `getTextFormat( )`, не оказывают никакого влияния на текст в объекте *объектTextField*, кроме тех случаев, когда измененный объект `TextFormat` повторно применяется к тексту через вызов метода `setTextFormat( )`. Например, само по себе следующее выражение присваивания значения переменной `font` не оказывает никакого влияния на объект `t`:

```
allCharsFormat.font = "Courier New";
```

Однако, если мы добавим вызов метода `setTextFormat( )`, изменение будет применено:

```
// Применяет шрифт "Courier New" ко всему текстовому полю
t.setTextFormat(allCharsFormat);
```

Форматирование по умолчанию для текстовых полей

Всякий раз, когда новый текст добавляется в текстовое поле (либо программным путем, либо через пользовательский ввод), среда выполнения Flash форматирует этот текст с использованием *формата текста по умолчанию* для данного поля. Формат текста по умолчанию для поля представляется внутренним объектом `TextFormat`, который определяет, как должен быть отформатирован новый текст, если форматирование не указано явно.

Тем не менее формат текста по умолчанию для текстового поля не является постоянным; он динамически изменяет свои настройки, чтобы соответствовать форматированию текста в точке вставки (также называемой *позицией курсора*). Таким образом, когда пользователь вводит новый текст после некоторого символа в текстовом поле, формат нового текста будет обусловлен форматом данного символа.

Как мы уже знаем, когда для добавления нового текста в текстовое поле используется метод `replaceText( )`, новый текст принимает форматирование либо символа, следующего за новым текстом (если существующий текст был удален), либо символа, предшествующего новому тексту (если существующий текст не был удален).

Вообще говоря, формат текста по умолчанию следует рассматривать не как инструмент разработчика для форматирования нового текста, добавляемого в текстовое поле, а как внутреннее средство среды Flash для определения форматирования нового текста, добавляемого в текстовое поле. Чтобы отформатировать новый текст с помощью некоторого формата, используйте метод `setTextFormat( )`. Например, рассмотрим следующий код, создающий текстовое поле, весь текст которого отформатирован с использованием полужирного начертания:

```
var t:TextField = new TextField( );
t.width = 400;
t.text = "This is bold text.";
var boldFormat:TextFormat = new TextFormat( );
boldFormat.bold = true;
t.setTextFormat(boldFormat);
```

В обычной ситуации любой новый текст, добавляемый в объект `t`, также будет автоматически отформатирован с использованием полужирного начертания:

```
t.appendText(" This is bold too.");
```

Чтобы добавить новый текст без полужирного начертания в объект `t`, мы должны применить нужное форматирование после добавления этого текста в поле, как показано в следующем коде:

```
// Добавляем текст
t.appendText(" This isn't bold.");

// Сразу же форматируем новый текст без использования полужирного
// начертания. Обратите внимание, что индексы первого и последнего символов
// нового текста определяются динамически с помощью метода String.indexOf( )
// и переменной TextField.length.
var regularFormat:TextFormat = new TextFormat( );
regularFormat.bold = false;
t.setTextFormat(regularFormat,
                t.text.indexOf("This isn't bold."),
                t.length);
```

Подобный подход, применяемый для форматирования добавляемого пользователем текста, продемонстрирован далее, в разд. «Ввод через текстовые поля».

Хотя формат текста по умолчанию в основном является внутренним инструментом среды Flash, разработчики могут использовать его для решения одной важной задачи: устанавливать форматирование *пустого* текстового поля. Формат для пустого объекта `TextField` указывается путем присваивания объекта `TextFormat` переменной `defaultTextFormat` объекта `TextField`, как показано в следующем коде:

```
// Создаем текстовое поле
var t:TextField = new TextField( );
t.width = 300;

// Создаем объект TextFormat
var defaultFormat:TextFormat = new TextFormat( );
defaultFormat.size = 20;
defaultFormat.color = 0xFF0000;
defaultFormat.font = "Verdana";

// Присваиваем объект TextFormat переменной defaultTextFormat объекта t
t.defaultTextFormat = defaultFormat;
```

Как только переменной `defaultTextFormat` пустого объекта `TextField` будет присвоено значение, весь текст, добавляемый в данное поле (либо программным путем, либо через пользовательский ввод), будет отформатирован в соответствии с указанным значением переменной `defaultTextFormat` до тех пор, пока к символам в текстовом поле не будет применено новое пользовательское форматирование. Например, следующий код добавляет новый текст в объект `t`; текст автоматически форматируется с использованием шрифта Verdana размером 20 пунктов красного цвета (в соответствии со значением переменной `t.defaultTextFormat`):

```
t.text = "This is 20 pt red Verdana";
```

Когда к символам в текстовом поле будет применено пользовательское форматирование, новый текст, добавляемый в поле, будет отформатирован в соответствии с форматом текста в месте вставки.

Теперь, когда мы познакомились с основами форматирования текста с помощью класса `TextFormat`, перейдем к форматированию текста с помощью разметки HTML.

HTML-форматирование текста

Чтобы применить форматирование к текстовому полю с помощью разметки HTML, используется следующая базовая последовательность действий.

1. Создать объект `TextField`.
2. Создать строку текста, представляющую разметку HTML, с использованием ограниченного набора HTML-тегов форматирования, поддерживаемых языком ActionScript.
3. Присвоить текст, отформатированный с помощью разметки HTML, переменной `htmlText` объекта `TextField`. Любой текст, отформатированный с помощью разметки HTML и присвоенный переменной `htmlText`, выводится на экран в виде отформатированного текста.

Применим перечисленные шаги на примере. Наша цель — отформатировать весь текст в поле, используя шрифт Arial размером 20 пунктов с полужирным начертанием (как мы делали это ранее с помощью объекта `TextFormat`).

Начнем с создания текстового поля, которое будет автоматически изменять свои размеры, чтобы соответствовать нашему отформатированному тексту:

```
var t:TextField = new TextField( );
t.autoSize = TextFieldAutoSize.LEFT;
```

Затем мы создадим нашу отформатированную текстовую строку, используя теги <B> и <FONT>:

```
var message:String = "<FONT FACE='Arial' SIZE='20'>"
                   + "<B>ActionScript is fun!</B></FONT>";
```

Наконец, присваиваем строку, содержащую разметку HTML, переменной `htmlText` объекта `t`:

```
t.htmlText = message;
```

Результат показан на рис. 27.11.

ActionScript is fun!

Рис. 27.11. Текст, отформатированный с помощью разметки HTML

Зачастую текст, содержащий разметку HTML, присваивается непосредственно переменной `htmlText`, как показано в следующем коде:

```
t.htmlText = "<FONT FACE='Arial' SIZE='20'>"
           + "<B>ActionScript is fun!</B></FONT>";
```

С помощью HTML-разметки мы можем применять любые параметры форматирования, доступные в классе `TextFormat`. В табл. 27.5 перечислены поддерживаемые

языком ActionScript теги и атрибуты HTML, дополненные перекрестными ссылками на эквивалентные переменные класса `TextFormat`. Дополнительную информацию по поддержке HTML в языке ActionScript можно найти в разделе, посвященном описанию переменной экземпляра `htmlText` класса `TextField`, справочника по языку ActionScript корпорации Adobe.

В отличие от автономной версии приложения Flash Player и версии, реализованной в виде модуля расширения браузера, приложение Adobe AIR включает полнофункциональные синтаксический анализатор и подсистему визуализации разметки HTML, которые способны обрабатывать весь диапазон инструкций языка HTML, таблиц стилей CSS и языка JavaScript, обычно применяемых в браузерах.

Обратите внимание, что, когда таблицы стилей не используются, приложение Flash Player автоматически добавляет разметку HTML к строковому значению переменной `htmlText`, если HTML-код, присвоенный переменной `htmlText`, не полностью описывает форматирование текстового поля. Например, следующий код присваивает переменной `htmlText` текст, не содержащий теги <P> или <FONT>:

```
var t:TextField = new TextField( );
t.htmlText = "This field contains <B>HTML!</B>";
```

Прочитав значение переменной `t.htmlText`, мы обнаружим, что были добавлены теги <P> и <FONT>:

```
trace(t.htmlText);
// Выводит:
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"
LETTERSPACING="0" KERNING="0">This field contains <B>HTML!</B></FONT></P>
```

Теперь рассмотрим набор тегов и атрибутов языка HTML, поддерживаемых языком ActionScript, которые представлены в табл. 27.5.

Таблица 27.5. Теги языка HTML, поддерживаемые языком ActionScript

Тег	Описание	Атрибуты	Описание	Эквивалентная переменная экземпляра класса TextFormat
1	2	3	4	5
<A>	Определяет гипертекстовую ссылку	HREF	Задает цель гипертекстовой ссылки	url
		TARGET	Определяет окно или фрейм гипертекстовой ссылки	target
<B>	Задает отображение символа с использованием полужирного начертания	Отсутствует		bold
 	Вызывает разрыв строки в теле текста; функционально эквивалентен служебной последовательности \n	Отсутствует		Отсутствует

1	2	3	4	5
<FONT>	Определяет информацию о шрифте	FACE	Определяет название шрифта	font
		SIZE	Задает размер шрифта, в пунктах	size
		COLOR	Определяет цвет шрифта в виде 24-битного целого шестнадцатеричного числа, которому предшествует знак фунта (#). Например, красный цвет записывается как #FF0000	color
		KERNING	Указывает, нужно ли использовать кернинг для пар символов (1 означает «использовать», 0 — «не использовать»)	kerning
		LETTERSPACING	Определяет расстояние между буквами (то есть трекинг), в пикселах	letterSpacing
<I>	Задает отображение символа с использованием курсива	Отсутствует		italic
<IMAGE>	Определяет отображаемый элемент, вставляемый в текстовое поле	SRC	Местоположение элемента (изображение, SWF-файл или символ клипа), вставляемого в текстовое поле	Отсутствует
		WIDTH	Необязательная ширина вставляемого элемента	Отсутствует
		HEIGHT	Необязательная высота вставляемого элемента	Отсутствует
		ALIGN	Необязательное выравнивание вставляемого элемента по горизонтали	Отсутствует
		HSPACE	Необязательное пространство по горизонтали, окружающее вставляемый элемент	Отсутствует
		VSPACE	Необязательное пространство по вертикали, окружающее вставляемый элемент	Отсутствует

Продолжение

Таблица 27.5 (продолжение)

1	2	3	4	5
		ID	Определяет необязательный идентификатор, по которому можно обращаться к вставляемому элементу через метод экземпляра getImageReference( ) класса TextField	Отсутствует
		CHECKPOLICYFILE	Указывает, требуется ли проверка файла политики безопасности перед обращением к элементу в виде данных (см. гл. 19)	Отсутствует
<LI>	Определяет абзац, перед которым отображается маркер. Стоит отметить, что изменить форму маркера невозможно, а теги <UL> или <OL> не требуются	Отсутствует		bullet
<P>	Определяет абзац	ALIGN	Определяет выравнивание абзаца по горизонтали (по левому, правому краю, по центру или по ширине)	align
		CLASS	Определяет класс CSS, применяется вместе с таблицами стилей	Отсутствует
<SPAN>	Помечает произвольный фрагмент текста, который может быть отформатирован с помощью таблицы стилей	CLASS	Определяет класс CSS, применяется вместе с таблицами стилей	Отсутствует
<TEXTFORMAT>	Определяет форматирующую информацию для фрагмента текста	LEFTMARGIN	Определяет расстояние по горизонтали в пикселах между левой границей текстового поля и левым краем абзаца	leftMargin
		RIGHTMARGIN	Задает расстояние по горизонтали в пикселах между правой границей текстового поля и правым краем абзаца	rightMargin

1	2	3	4	5
		BLOCKINDENT	Определяет расстояние в пикселах, на которое смещается абзац относительно левой границы текстового поля	blockIndent
		INDENT	Определяет расстояние в пикселах, на которое смещается первая строка относительно левой границы текстового поля	indent
		LEADING	Определяет величину промежутка по вертикали в пикселах между строками текста	leading
		TABSTOPS	Определяет шаги табуляции по горизонтали, в пикселах	tabStops
<U>	Задает отображение символа с использованием подчеркивания	Отсутствует		underline

Вообще говоря, использование тегов языка HTML, перечисленных в табл. 27.5, в ActionScript аналогично их использованию в распространенных браузерах. С другой стороны, между использованием HTML в ActionScript и использованием HTML в браузерах существует несколько существенных отличий.

- Тег <TABLE> не поддерживается; используйте шаги табуляции для имитации таблиц языка HTML.
- В языке ActionScript разметка HTML в основном используется для форматирования, и содержимое HTML не организуется в виде метафоры документа браузера. Следовательно, теги <HTML> и <BODY> не являются обязательными (однако тег <BODY> по желанию может использоваться для форматирования содержимого HTML с помощью таблиц стилей).
- Неподдерживаемые теги игнорируются, хотя их текстовое содержимое сохраняется.
- В языке ActionScript значения, присваиваемые атрибутам тегов, должны заключаться в кавычки. Дополнительная информация представлена в подразд. «Заключение в кавычки значений атрибутов».
- В среде выполнения Flash гипертекстовые ссылки автоматически не подчеркиваются — подчеркивание должно выполняться вручную либо с помощью тега <U>, либо с помощью переменной text-decoration таблицы стилей CSS.
- Тег <LI> не поддерживает вложенные маркеры и тег <OL> (нумерованный список).
- В приложении Flash Player незакрытые теги <P> не вызывают разрывы строк, как это происходит в обычном языке HTML. Для добавления разрывов строк Flash Player требует, чтобы были указаны закрывающие теги </P>.

- В приложении Flash Player тег <P> вызывает один разрыв строки, как и тег
, хотя в браузерах тег <P> обычно вызывает двойной разрыв строки.
- Теги <P> и
 не вызывают разрывы строк в текстовых полях, переменной multiline которых присвоено значение false. Более того, переменной multiline по умолчанию присваивается значение false. Таким образом, при использовании тегов <P> и
 присваивайте переменной multiline значение true.
- Гипертекстовые ссылки могут быть использованы для выполнения кода на языке ActionScript. Дополнительную информацию можно найти далее, в подразд. «Гипертекстовые ссылки» разд. «Ввод через текстовые поля».
- Атрибут NAME тега <A> не поддерживается приложением Flash Player, поэтому внутренние ссылки внутри тела текста невозможны.
- В среде выполнения Flash якорные теги не добавляются в последовательность перехода, и поэтому на них нельзя перейти с помощью клавиатуры.

Поддержка сущностей

Сущности специальных символов, поддерживаемые языком ActionScript, перечислены в табл. 27.6. Если сущность встречается в значении переменной htmlText текстового поля, то приложение Flash Player отображает на экране соответствующий символ. Поддерживаются также и числовые сущности, например ™ (символ торгового знака).

Таблица 27.6. Поддерживаемые сущности

Сущность	Представляемый символ
<	<
>	>
&	&
"	"
'	'
	Неразрывный пробел

Заключение в кавычки значений атрибутов

За пределами приложения Flash Player значения атрибутов языка HTML могут быть заключены в одинарные или двойные кавычки либо вообще быть без кавычек. Следующие теги допустимо использовать в большинстве браузеров:

```
<P ALIGN=RIGHT>
<P ALIGN='RIGHT'>
<P ALIGN="RIGHT">
```

Однако в приложении Flash Player значения атрибутов, не заключенные в кавычки, недопустимы. Например, в языке ActionScript синтаксис <P ALIGN=RIGHT> недопустим. Тем не менее для отделения значений атрибутов можно использовать как одинарные, так и двойные кавычки. При создании значений текстовых полей, содержащих атрибуты языка HTML, используйте один тип кавычек для обозначения самой строки, а другой тип кавычек — для отделения значений атрибутов. Например, оба следующих примера допустимы:

```
t.htmlText = "<P ALIGN='RIGHT'>hi there</P>";
t.htmlText = '<P ALIGN="RIGHT">hi there</P>';
```

Однако следующий пример приведет к ошибке, поскольку двойные кавычки используются и для обозначения строки, и для отделения значения атрибута:

```
// НЕПРАВИЛЬНО! Не делайте так!
t.htmlText = "<P ALIGN="RIGHT">hi there</P>";
```

Взаимосвязь переменных text и htmlText

Поскольку присваивать текстовое содержимое текстовому полю можно и с помощью переменной `text`, и с помощью переменной `htmlText` класса `TextField`, проявляйте осторожность при одновременном их использовании.

Когда разметка с тегами HTML присваивается переменной `htmlText`, значением переменной `text` будет являться значение переменной `htmlText`, но с опущенными HTML-тегами. Например, в следующей строке кода мы присваиваем фрагмент HTML переменной `htmlText` текстового поля:

```
var t:TextField = new TextField( );
t.htmlText = '<P ALIGN="LEFT">' +
    + '<FONT FACE="Times New Roman" SIZE="12" COLOR="#000000" '
    + 'LETTERSPACING="0" KERNING="0">This field contains <B>HTML!</B>'
    + '</FONT></P>';
```

После присваивания переменная `htmlText` будет иметь следующее значение:

```
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"
LETTERSPACING="0" KERNING="0">This field contains <B>HTML!</B></FONT></P>
```

Однако переменная `text` будет иметь следующее значение:

```
This field contains HTML!
```

Обратите внимание, что последовательные присваивания значений переменным `htmlText` и `text` перезаписывают предыдущие результаты. Иначе говоря, присваивание нового значения переменной `text` перезаписывает существующее значение переменной `htmlText`, и наоборот. В отличие от этого, последовательные операции конкатенации (не повторного присваивания) не перезаписывают существующие значения. Например, следующий код присваивает некоторое содержимое HTML переменной `htmlText`, после чего выполняет конкатенацию данного содержимого со строкой через переменную `text`:

```
var t:TextField = new TextField( );
t.htmlText = "<B>hello</B>";
t.text += " world";
```

После конкатенации значение переменной `htmlText` будет выглядеть следующим образом:

```
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"
LETTERSPACING="0" KERNING="0">hello world</FONT></P>
```

Как показано в предыдущем коде, конкатенация значения переменной `text` со значением переменной `htmlText` приводит к сбрасыванию форматирования текстового поля. Когда мы присваиваем строку `"world"` переменной `text`, среда выполнения

Flash удаляет тег <B>, который изначально был присвоен переменной htmlText! Таким образом, одновременное использование операций присваивания значений переменным text и htmlText в большинстве случаев не рекомендуется.

Теги HTML, присваиваемые непосредственно переменной экземпляра text класса TextField, никогда не интерпретируются в виде разметки HTML; они всегда отображаются в первоначальном виде. Например, следующий код присваивает строку, содержащую теги HTML, переменной text, а затем выполняет конкатенацию данного содержимого с обычной строкой через переменную htmlText:

```
var t:TextField = new TextField( );
t.text = "<B>world</B>";
t.htmlText += "hello";
```

После конкатенации значение переменной htmlText будет выглядеть следующим образом:

```
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"
LETTERSPACING="0" KERNING="0">&lt;B&gt;world&lt;/B&gt;hello</FONT></P>
```

Обратите внимание, что символы < и > в теге <B> были преобразованы в сущности языка HTML < и >.

Нераспознанные теги и атрибуты

Как и браузеры, приложение Flash Player игнорирует теги и атрибуты, которые не может распознать. Например, мы присвоим следующее значение переменной htmlText:

```
<P>Please fill in and print this form</P>
<FORM><INPUT TYPE="TEXT"></FORM>
<P>Thank you!</P>
```

В результате на экране появится следующий текст:

```
Please fill in and print this form
Thank you!
```

Элементы <FORM> и <INPUT> не поддерживаются приложением Flash Player, поэтому оба тега игнорируются (фактически неизвестные теги удаляются из значения переменной htmlText!).

Подобным образом, если мы используем контейнерные элементы, например <TD>, содержимое сохраняется, однако разметка игнорируется. Например, выполнение следующего кода:

```
объектTextField.htmlText = "<TABLE><TR><TD>table cell text</TD></TR></TABLE>";
```

приведет к выводу на экран такой строки без табличного форматирования:

```
table cell text
```

Однако если не закрыть тег, весь последующий текст будет считаться частью этого тега и не отобразится на экране. Например, в результате следующей операции присваивания:

```
объектTextField.htmlText = "We all know that 5 < 20. That's obvious.";
```

приложение Flash Player выведет:

```
We all know that 5
```

Чтобы включить символ < в разметку HTML текстового поля, используйте сущность `<`, как показано в следующей строке кода:

```
объектTextField.htmlText = "We all know that 5 &lt; 20. That's obvious.";
```

Приложение Flash Player выведет текст:

```
We all know that 5 < 20. That's obvious.
```

Дополнительную информацию по включению исходного HTML-кода в текстовое поле можно найти по адресу http://moock.org/asdg/technotes/sourceInHtmlField/.

Мы узнали, каким образом можно отформатировать текстовое поле с помощью класса `TextFormat` и разметки HTML. Теперь рассмотрим последний инструмент, предназначенный для форматирования текста, — класс `StyleSheet`.

Форматирование текста с помощью класса StyleSheet

Класс `StyleSheet` языка ActionScript применяется для форматирования текстовых полей с помощью таблиц стилей. Функциональность этого класса базируется на очень ограниченном подмножестве каскадных таблиц, описанных в рекомендации уровня 1 (CSS1) консорциума W3C.

Для чтения этого раздела требуется понимание основных концепций языка CSS. Если вы незнакомы с этим языком, перед продолжением прочитайте следующий вводный материал в язык CSS: http://www.w3.org/TR/CSS21/intro.html и http://www.w3.org/MarkUp/Guide/Style.

Однако имейте в виду, что язык ActionScript не поддерживает полный набор возможностей, рекомендованных консорциумом W3C.

Как описано в рекомендации по языку CSS консорциума W3C, *таблица стилей* — это набор правил, которые определяют представление документа. Каждое правило описывает стиль для определенного элемента в документе HTML или XML. Следующий код демонстрирует пример правила, которое задает шрифт красного цвета для элементов `<H1>`:

```
h1 { color: #FF0000 }
```

Внутри данного правила селектор (`h1`) обозначает элемент, к которому применяется стиль. Блок объявления (`{ color: #FF0000 }`) содержит одно или более объявлений, описывающих стиль, который должен применяться к выбранному элементу. Каждое объявление (`color: #FF0000`) включает свойство стиля (`color`) и значение (`#FF0000`).

Названия селекторов не зависят от регистра символов, в отличие от имен свойств стиля.

Вот простая таблица стилей, которая содержит два правила — одно для элементов `<p>` и одно для элементов `<li>`:

```
p {
  font-family: sans-serif
  font-size: 12px;
}

li {
  font-family: serif
  font-size: 10px;
  margin-left: 10px;
}
```

В языке ActionScript 3.0 таблица стилей, подобная той, которая продемонстрирована в предыдущем коде, представляется экземпляром класса `StyleSheet`. С помощью методов класса `StyleSheet` мы можем создать новую таблицу стилей программным путем или провести синтаксический разбор существующей внешней таблицы стилей. Чтобы связать объект `StyleSheet` с определенным объектом `TextField`, мы присваиваем объект `StyleSheet` переменной `styleSheet` (которая будет рассмотрена далее) данного объекта `TextField`. С каждым текстовым полем может быть связан всего лишь один объект `StyleSheet`, что резко контрастирует с языком CSS1, позволяющим связывать с одним документом HTML или XML несколько таблиц стилей.

Конкретные свойства, доступные для использования в таблицах стилей ActionScript, перечислены в табл. 27.7. Язык ActionScript поддерживает только те свойства стилей из рекомендации консорциума W3C, которые соответствуют параметрам форматирования класса `TextFormat`. В сравнении с полным набором свойств, определяемых консорциумом W3C, набор свойств, поддерживаемых ActionScript, чрезвычайно ограничен.

Таблица 27.7. Поддерживаемые свойства стиля языка CSS

Название свойства стиля	Описание
color	Задает цвет шрифта в виде 24-битного шестнадцатеричного числа, перед которым указывается знак фунта (#). Например, красный цвет записывается как #FF0000
display	Определяет, должен ли элемент быть скрыт (none), должен ли после него добавляться автоматический разрыв строки (block) или же автоматический разрыв строки добавляться не должен (inline)
font-family	Задает название шрифта устройства или встроенного шрифта
font-size	Указывает размер шрифта в пикселах
font-style	Задает отображение символа с использованием курсива (italic) или обычное отображение символа (normal по умолчанию)
font-weight	Задает отображение символа с использованием полужирного начертания (bold) или обычное отображение символа (normal по умолчанию)
kerning	Определяет, должен ли быть использован автоматический кернинг для пар символов (true) или нет (false). Это свойство является неофициальным расширением набора поддерживаемых свойств стилей, описанных в рекомендации консорциума W3C
leading	Задает расстояние по вертикали между строками текста, в пикселах. Это свойство является неофициальным расширением набора поддерживаемых свойств стилей, описанных в рекомендации консорциума W3C. Сравните его со свойством line-height, описанным в рекомендации консорциума W3C

Название свойства стиля	Описание
letter-spacing	Задает расстояние между буквами (то есть трекинг) в пикселах
margin-left	Определяет расстояние по горизонтали между левой границей текстового поля и левым краем абзаца в пикселах
margin-right	Задает расстояние по горизонтали между правой границей текстового поля и правым краем абзаца в пикселах
text-align	Задает выравнивание абзаца по горизонтали (left (по умолчанию), right, center или justify)
text-decoration	Определяет графические элементы украшения, добавляемые к тексту. В ActionScript поддерживаемыми значениями являются underline и none (по умолчанию)
text-indent	Задает расстояние, которое обозначает смещение первой строки абзаца относительно левой границы текстового поля (аналогично переменной экземпляра indent класса TextFormat), в пикселах

Таблицы стилей могут применяться для форматирования как XML-, так и HTML-элементов. Однако среди HTML-элементов, используемых приложением Flash Player для форматирования (см. табл. 27.7), с помощью таблиц стилей могут быть отформатированы только теги <P>, <LI> и <A>. Другие встроенные теги (например, <B> и <I>) всегда выполняют свою предопределенную задачу форматирования HTML, и его нельзя изменить с помощью таблиц стилей. Более того, теги <P> и <LI> всегда отображаются в виде блочных элементов, даже если в соответствии с таблицей стилей они должны отображаться в виде встроенных элементов.

Чтобы добавить форматирование для различных интерактивных состояний гипертекстовой ссылки, используйте следующие селекторы псевдоклассов: `a:link`, `a:hover` и `a:active`. Например, следующее правило определяет, что гипертекстовые ссылки при наведении на них указателя мыши должны подчеркиваться:

```
a:hover {
  text-decoration: underline;
}
```

Благодаря таблицам стилей у разработчиков появляется чрезвычайно важная возможность отделять информацию о стилях от содержимого и применять одинаковое определение стиля к нескольким блокам содержимого. Тем не менее в языке ActionScript таблицы стилей имеют много ограничений, снижающих их потенциальную полезность. Перед тем как рассмотреть вопросы форматирования текста с помощью таблиц стилей, внимательно познакомимся с этими ограничениями.

Существенные ограничения таблиц стилей в языке ActionScript

Приложение Flash Player намеренно предоставляет только минимальную реализацию таблиц стилей, играющую роль интерфейса для установки параметров форматирования текстового поля. В результате поддерживаемым приложением Flash Player таблицам стилей не хватает нескольких важных возможностей, описанных в рекомендации по языку CSS консорциума W3C. Читатели, привыкшие работать

с CSS и HTML, должны помнить о следующих ограничениях приложения Flash Player.

- Все длины выражаются в пикселах. Относительная единица `em` не поддерживается, а пункты трактуются как пикселы.
- В любой момент времени с каждым текстовым полем можно связать всего одну таблицу стилей. Таблицы стилей приложения Flash Player не каскадируются.
- Повторное присваивание таблицы стилей текстовому полю не приводит к отображению этого поля с использованием новой таблицы стилей (в качестве обходного решения можно применять подход, описание которого приводится сразу за данным списком).
- Свойства `margin-top` и `margin-bottom` не поддерживаются.
- Элементы не могут произвольным образом отображаться в виде элементов списка. Значение `list-item` свойства `display` не поддерживается. Более того, форму маркеров элемента списка (то есть метки) нельзя изменять, даже для собственного элемента `<LI>` языка HTML.
- Приложение Flash Player поддерживает только базовые селекторы типов и селекторы классов. Все остальные множества селекторов не поддерживаются. Более того, селекторы типов и селекторы классов нельзя объединять (например, следующий селектор в приложении Flash Player недопустим: `p.someCustomClass`). Если таблица стилей содержит селектор потомка, вся таблица стилей будет проигнорирована и никакое форматирование применено не будет.
- Если таблица стилей присваивается текстовому полю, то текстовое содержимое этого поля не может быть изменено с помощью методов `replaceText( )`, `appendText( )`, `replaceSelText( )` или через пользовательский ввод.

Чтобы изменить таблицу стилей текстового поля, сначала присвойте желаемый объект `StyleSheet` переменной `styleSheet`, а затем присвойте переменную `htmlText` самой себе, как показано в следующем коде:

```
t.styleSheet = некийНовыйОбъектStyleSheet;
t.htmlText = t.htmlText;
```

Стоит отметить, что новая таблица стилей должна устанавливать все свойства стиля, которые были установлены прежней таблицей стилей. В противном случае любые неустановленные прежние значения свойств стиля будут сохранены.

Теперь, когда мы познакомились с основными возможностями и ограничениями таблиц стилей, поддерживаемых приложением Flash Player, посмотрим на них в действии. Следующие два раздела описывают, как применять таблицу стилей к текстовому полю, сначала с использованием таблицы стилей, создаваемой программным путем, а затем с помощью таблицы стилей, загружаемой из внешнего CSS-файла.

Форматирование текста с помощью таблицы стилей, создаваемой программным путем

Чтобы отформатировать текст с помощью таблицы стилей, создаваемой программным путем, используйте следующую обобщенную последовательность действий.

1. Создайте один или несколько базовых блоков, представляющих блоки объявления правил.
2. Создайте объект `StyleSheet`.
3. Используйте метод экземпляра `setStyle( )` класса `StyleSheet`, чтобы создать одно или несколько правил на основании блоков объявления, созданных на шаге 1.
4. Используйте переменную экземпляра `styleSheet` класса `TextField`, чтобы зарегистрировать объект `StyleSheet` в желаемом объекте `TextField`.
5. Присвойте желаемое содержимое HTML или XML переменной `htmlText` объекта `TextField`.

Всегда регистрируйте объект StyleSheet (шаг 4) до того, как будет присвоено содержимое HTML или XML (шаг 5). В противном случае таблица стилей не будет применена к этому содержимому.

Применим перечисленные шаги на практике. Наша цель — отформатировать весь текст в текстовом поле, используя шрифт Arial размером 20 пунктов полужирного начертания (как мы делали это ранее с помощью класса `TextFormat` и разметки HTML). Как и раньше, форматируемым текстом является следующий простой фрагмент на языке HTML:

```
<p>ActionScript is fun!</p>
```

В нашей таблице стилей мы определяем правило, которое говорит приложению Flash Player отображать содержимое всех тегов `<P>` с помощью шрифта Arial размером 20 пунктов полужирного начертания. Блок объявления для нашего правила является простым базовым объектом с динамическими переменными экземпляра, имена которых соответствуют свойствам стиля CSS, а значения переменных задаются соответствующими значениями стиля CSS. Вот этот код:

```
// Создаем объект, который будет служить блоком объявления
var pDeclarationBlock:Object = new Object( );

// Присваиваем значения свойствам стиля
pDeclarationBlock.fontFamily = "Arial"
pDeclarationBlock.fontSize   = "20";
pDeclarationBlock.fontWeight = "bold";
```

В приведенном коде обратите внимание на то, что при создании таблиц стилей программным путем формат имен свойств CSS слегка изменяется: дефисы удаляются, а символы, которые следуют за дефисами, записываются в верхнем регистре. Например, свойство `font-family` превращается в свойство `fontFamily`.

Теперь, когда наш блок объявления готов, мы создаем объект `StyleSheet`. Вот этот код:

```
var styleSheet:StyleSheet = new StyleSheet( );
```

Чтобы создать правило для тега `<P>`, мы используем метод экземпляра `setStyle( )` класса `StyleSheet`. Метод `setStyle( )` создает новое правило стиля на основе

двух параметров: имени селектора (тип `String`) и блока объявления (тип `Object`), как показано в следующем обобщенном коде:

```
объектStyleSheet.setStyle("селектор", блокОбъявления);
```

Соответственно, вот код, создающий правило для тега <P>:

```
styleSheet.setStyle("p", pDeclarationBlock);
```

Наша таблица стилей готова. Теперь мы создадим текстовое поле, к которому будет применено форматирование. Следующий код создает текстовое поле и присваивает наш объект `StyleSheet` переменной `styleSheet` этого поля:

```
var t:TextField = new TextField( );
t.width = 200
t.styleSheet = styleSheet;
```

Наконец, мы присваиваем текст, который должен быть отформатирован, переменной `htmlText`:

```
t.htmlText = "<p>ActionScript is fun!</p>";
```

В листинге 27.7 представлен весь код целиком для нашего текстового поля, к которому применяется форматирование.

Листинг 27.7. Форматирование текста с помощью таблицы стилей, создаваемой программным путем

```
// Создаем блок объявления
var pDeclarationBlock:Object = new Object( );
pDeclarationBlock.fontFamily = "Arial"
pDeclarationBlock.fontSize = "20";
pDeclarationBlock.fontWeight = "bold";

// Создаем таблицу стилей
var styleSheet:StyleSheet = new StyleSheet( );

// Создаем правило
styleSheet.setStyle("p", pDeclarationBlock);

// Создаем текстовое поле
var t:TextField = new TextField( );
t.width = 200

// Присваиваем таблицу стилей
t.styleSheet = styleSheet;

// Присваиваем HTML-код, который должен быть отформатирован
t.htmlText = "<p>ActionScript is fun!</p>";
```

Результат выполнения кода из листинга 27.7 идентичен результату, который был проиллюстрирован ранее на рис. 27.10 и 27.11.

Селекторы класса

Чтобы применить стиль не ко всем абзацам сразу, а к одному определенному подмножеству абзацев, мы используем селектор класса CSS. Предположим, что мы

хотим акцентировать внимание пользователя на важных примечаниях в документе. Мы помещаем примечания в теги <P> и присваиваем атрибуту `class` этих тегов собственное значение `specialnote`, как показано в следующем коде:

```
<p class='specialnote'>Set styleSheet before htmlText!</p>
```

Затем мы создаем правило для этого класса `specialnote`, используя селектор класса, как показано в следующем коде:

```
styleSheet.setStyle(".specialnote", specialnoteDeclarationBlock);
```

Как и в языке CSS, селектор класса в языке ActionScript состоит из точки, за которой указывается желаемое значение атрибута `class` (в нашем случае `specialnote`).

В листинге 27.8 представлен переработанный пример нашей предыдущей таблицы стилей, чтобы продемонстрировать использование селекторов класса CSS в языке ActionScript.

Листинг 27.8. Форматирование, применяемое к определенному классу абзаца

```
var specialnoteDeclarationBlock:Object = new Object( );
specialnoteDeclarationBlock.fontFamily = "Arial"
specialnoteDeclarationBlock.fontSize   = "20";
specialnoteDeclarationBlock.fontWeight = "bold";

var styleSheet:StyleSheet = new StyleSheet( );
styleSheet.setStyle(".specialnote", specialnoteDeclarationBlock);

// Создаем текстовое поле
var t:TextField = new TextField( );
t.width     = 300;
t.wordWrap  = true;
t.multiline = true;
t.styleSheet = styleSheet;
t.htmlText = "<p>Always remember...</p>"
  + "<p class='specialnote'>Set styleSheet before htmlText!</p>"
  + "<p>Otherwise, the stylesheet will not be applied.</p>";
```

Результат выполнения кода из листинга 27.8 продемонстрирован на рис. 27.12.

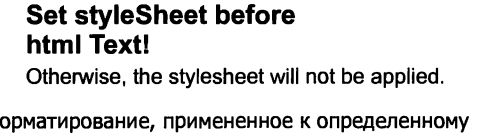

Рис. 27.12. Форматирование, примененное к определенному классу абзаца

Форматирование тегов XML с помощью стилей CSS

Чтобы применить стиль к определенному фрагменту содержимого, мы можем создать для этого фрагмента собственный XML-тег. Например, вместо того, чтобы описывать особое примечание в виде класса абзаца (как это было сделано в предыдущем разделе), мы могли бы создать совершенно новый XML-тег, как показано в следующем коде:

```
<specialnote>Set styleSheet before htmlText!</specialnote>
```

Чтобы применить правило стиля к тегу <specialnote>, мы используем обычный селектор типа (без точки в начале), как показано в следующем коде:

```
styleSheet.setStyle("specialnote", specialnoteDeclarationBlock);
```

Кроме того, чтобы указать, что поведение нашего тега <specialnote> должно быть аналогичным поведению абзаца, мы присваиваем переменной display в правиле <specialnote> значение block. В листинге 27.9 представлен весь код целиком, предназначенный для форматирования пользовательского XML-тега, при этом заслуживающие внимания отличия от нашего предыдущего кода с применением селектора класса выделены полужирным шрифтом.

Листинг 27.9. Форматирование содержимого XML с помощью таблицы стилей

```
var specialnoteDeclarationBlock:Object = new Object( );
specialnoteDeclarationBlock.fontFamily = "Arial"
specialnoteDeclarationBlock.fontSize   = "20";
specialnoteDeclarationBlock.fontWeight = "bold";
specialnoteDeclarationBlock.display    = "block";

var styleSheet:StyleSheet = new StyleSheet( );
styleSheet.setStyle("specialnote", specialnoteDeclarationBlock);

var t:TextField = new TextField( );
t.width     = 300;
t.wordWrap  = true;
t.multiline = true;
t.styleSheet = styleSheet;
t.htmlText = "<p>Always remember...</p>"
  + "<specialnote>Set styleSheet before htmlText!</specialnote>"
  + "<p>Otherwise, the stylesheet will not be applied.</p>";
```

Результат выполнения кода из листинга 27.9 идентичен результату, который был проиллюстрирован на рис. 27.12.

Форматирование текста с помощью загружаемой извне таблицы стилей

Чтобы отформатировать текст с помощью загружаемой извне таблицы стилей, используйте следующую обобщенную последовательность действий.

1. Создайте таблицу стилей во внешнем CSS-файле.
2. Используйте класс URLLoader, чтобы загрузить этот CSS-файл.
3. После завершения загрузки CSS-файла создайте объект StyleSheet.
4. Используйте метод экземпляра parseCSS( ) класса StyleSheet, чтобы импортировать правила из CSS-файла в объект StyleSheet.
5. Используйте переменную экземпляра styleSheet класса TextField, чтобы зарегистрировать объект StyleSheet в желаемом объекте TextField.
6. Присвойте требуемое содержимое HTML или XML переменной htmlText объекта TextField.

Применим перечисленные шаги на примере. Наша цель, как и раньше, — создать приложение, которое форматирует весь текст в поле, используя шрифт Arial разме-

ром 20 пунктов полужирного начертания. Как и раньше, форматируемым текстом является следующий простой фрагмент на языке HTML:

```
<p>ActionScript is fun!</p>
```

Сначала мы добавляем следующее CSS-правило в текстовый файл с именем `styles.css`:

```
p {
  font-family: Arial;
  font-size: 20px;
  font-weight: bold;
}
```

Далее мы создаем основной класс нашего приложения `StyleSheetLoadingDemo`. Он использует объект `URLLoader` для загрузки файла `styles.css`, как показано в следующем коде:

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;
  import flash.net.*;

  public class StyleSheetLoadingDemo extends Sprite {
    public function StyleSheetLoadingDemo ( ) {
      // Загружаем файл styles.css
      var urlLoader:URLLoader = new URLLoader( );
      urlLoader.addEventListener(Event.COMPLETE, completeListener);
      urlLoader.load(new URLRequest("styles.css"));
    }

    private function completeListener (e:Event):void {
      // Код этого метода выполняется после завершения загрузки
      // файла styles.css
    }
  }
}
```

Когда загрузка файла `styles.css` будет завершена, произойдет вызов метода `completeListener( )`. Внутри этого метода мы создаем новый объект `StyleSheet` и импортируем в него правила из файла `styles.css`, как показано в следующем коде:

```
private function completeListener (e:Event):void {
  var styleSheet:StyleSheet = new StyleSheet( );
  styleSheet.parseCSS(e.target.data);
}
```

Как только правила будут импортированы в объект `StyleSheet`, мы создаем наш объект `TextField`, регистрируем таблицу стилей и затем присваиваем форматируемый текст, как показано в следующем коде:

```
var t:TextField = new TextField( );
t.width = 200;
```

```
t.styleSheet = styleSheet;
t.htmlText = "<p>ActionScript is fun!</p>";
```

В листинге 27.10 показан весь код целиком для класса `StyleSheetLoadingDemo`.

Листинг 27.10. Форматирование текста с помощью внешней таблицы стилей

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;
  import flash.net.*;

  public class StyleSheetLoadingDemo extends Sprite {
    public function StyleSheetLoadingDemo ( ) {
      var urlLoader:URLLoader = new URLLoader( );
      urlLoader.addEventListener(Event.COMPLETE, completeListener);
      urlLoader.load(new URLRequest("styles.css"));
    }

    private function completeListener (e:Event):void {
      var styleSheet:StyleSheet = new StyleSheet( );
      styleSheet.parseCSS(e.target.data);

      var t:TextField = new TextField( );
      t.width = 200;
      t.styleSheet = styleSheet;
      t.htmlText = "<p>ActionScript is fun!</p>";

      addChild(t);
    }
  }
}
```

Результат выполнения кода из листинга 27.10 идентичен результату, который был проиллюстрирован ранее на рис. 27.10 и 27.11.

Мы рассмотрели методики форматирования текста в языке ActionScript. В следующем разделе будут освещены несколько вопросов, касающихся визуализации и использования шрифтов в SWF-файле.

Шрифты и отображение текста

По умолчанию приложение Flash Player отображает текст с помощью *шрифтов устройства*. Это шрифты, установленные в системе конечного пользователя. Когда приложение Flash Player отображает текст с помощью шрифта устройства, оно полностью делегирует процесс визуализации текста локальной среде (то есть операционной системе). Например, рассмотрим следующее простое приложение `HelloWorld`, создающее текстовое поле, для форматирования которого применяется шрифт Arial:

```
package {
  import flash.display.*;
  import flash.text.*;
```

```
  public class HelloWorld extends Sprite {
    public function HelloWorld ( ) {
      var fontFormat:TextFormat = new TextFormat( );
      fontFormat.font = "Arial";

      var t:TextField = new TextField( );
      t.text = "Hello world";
      t.setTextFormat(fontFormat);

      addChild(t);
    }
  }
}
```

При выполнении этого кода приложение Flash Player добавляет объект `t` в список отображения и готовится к обновлению экрана. Чтобы отобразить символы «Hello world», приложение Flash Player передает строку `"Hello world"` визуализатору текста операционной системы и просит его отобразить эти символы с помощью системного шрифта Arial. Операционная система визуализирует эти символы непосредственно в буфере кадра приложения Flash Player. Например, в операционной системе Windows XP строка `"Hello world"` будет отображена с использованием визуализатора ClearType корпорации Microsoft.

Если бы предыдущее приложение `HelloWorld` выполнялось на двух различных компьютерах под управлением двух различных операционных систем, эти компьютеры могли бы иметь два различных встроенных визуализатора текста и, возможно, даже две различные версии шрифта Arial. Таким образом, даже если требуемый шрифт будет доступен, символ, отображаемый на экране, на разных компьютерах может выглядеть по-разному. Кроме того, незначительное влияние на поведение визуализатора текста могут оказать переменная `cacheAsBitmap` или фильтры. Например, если объект `TextField` помещается в объект `Sprite`, переменной `cacheAsBitmap` которого присвоено значение `true`, операционная система Windows XP будет использовать обычное сглаживание вместо визуализатора ClearType.

Если шрифт, указанный для некоторого символа, не установлен в операционной системе конечного пользователя, то приложение Flash Player автоматически попросит операционную систему отобразить данный символ с помощью подходящего замещающего шрифта. Например, если для некоторого символа указан шрифт Verdana и этот символ отображается в стандартной версии операционной системы Mac OS X (не включающей шрифт Verdana), он будет визуализирован с помощью рубленого шрифта Helvetica, используемого по умолчанию. Таким образом, в зависимости от доступности шрифтов в операционной системе конечного пользователя, текст, визуализируемый с помощью шрифтов устройства на двух различных компьютерах под управлением двух различных операционных систем, может выглядеть совершенно по-разному.

При использовании шрифтов устройства отображение текста зависит от операционной системы и — для версии приложения Flash Player, реализованной в виде модуля расширения, — от браузера.

Если для некоторого символа шрифт не указан вообще, то локальный визуализатор отображает этот символ с использованием произвольного шрифта по умолчанию, выбираемого приложением Flash Player. Например, в операционной системе Windows XP шрифтом по умолчанию является Times New Roman.

Чтобы исключить различия в отображении текста на разных компьютерах и устройствах, приложение Flash Player позволяет разработчикам встраивать начертания шрифтов в SWF-файл. Текст, отображаемый с помощью встроенных начертаний шрифтов, гарантированно будет иметь очень похожий внешний вид на различных компьютерах, операционных системах и устройствах. Однако за такую согласованность необходимо платить; встраивание начертания для шрифта Roman целиком обычно увеличивает размер SWF-файла на 20–30 Кбайт (шрифты для азиатских языков могут занимать гораздо больше места). В отличие от этого, шрифты устройств совершенно не увеличивают размер SWF-файл. Таким образом, шрифты устройств обычно используются в тех случаях, когда небольшой размер файла гораздо важнее визуальной целостности, а встраиваемые начертания шрифтов обычно применяются, когда визуальная целостность гораздо важнее небольшого размера файла.

Чтобы использовать встроенный шрифт, мы должны сначала встроить начертания этого шрифта, а затем на этапе выполнения связать встроенные шрифты с желаемым текстовым полем (-ями). Когда встроенные шрифты связываются с текстовым полем, оно отображается либо с помощью стандартного визуализатора векторной графики приложения Flash Player, либо с помощью специализированного визуализатора FlashType, но не с помощью визуализатора текста локальной среды. Обратите внимание, что каждая вариация стиля шрифта должна быть встроена по отдельности. Если текстовое поле использует встраиваемые версии шрифта Courier New полужирного начертания, курсива и полужирного курсива, мы должны встроить все три вариации шрифта, иначе текст будет отображаться неправильно. Подчеркивание не считается ни вариацией шрифта, ни его размером или цветом.

Методика встраивания начертаний шрифтов на этапе компиляции зависит от различных инструментов разработки. В двух следующих разделах рассматривается, как встраивать шрифты в среде разработки Flash и в приложении Flex Builder или консольном компиляторе mxmlc. Для примера в каждом разделе описывается процедура встраивания шрифта Verdana. Как только начертания шрифта будут встроены в SWF-файл, они могут быть использованы для форматирования текста, как описывается в подразд. «Форматирование текста с помощью встраиваемого шрифта» этого раздела.

Встраивание начертаний шрифта в среде разработки Flash

Чтобы встроить начертания шрифта Verdana в среде разработки Flash, выполняйте следующие шаги.

1. Выберите команду меню Window ▸ Library (Окно ▸ Библиотека).
2. В меню Options (Параметры), значок которого расположен в правом верхнем углу панели Library (Библиотека), выберите команду New Font (Новый шрифт). На экране появится окно Font Symbol Properties (Свойства символа шрифта).
3. В раскрывающемся списке Font (Шрифт) выберите значение Verdana.

4. В поле Name (Имя) введите значение `Verdana` (это имя выполняет косметическую функцию и используется только на палитре Library (Библиотека)).
5. Нажмите кнопку OK.
6. На палитре Library (Библиотека) выберите символ шрифта Verdana.
7. В меню Options (Параметры) выберите команду Linkage (Связывание).
8. В области Linkage (Связывание) окна Linkage Properties (Свойства связывания) установите флажок Export For ActionScript (Экспорт для ActionScript).
9. В поле Class (Класс) должно автоматически появиться значение `Verdana`. Если этого не произошло, введите значение `Verdana` в поле ввода Class (Класс). Указанное имя класса используется при загрузке шрифтов на этапе выполнения (этот процесс рассматривается далее, в подразд. «Загрузка шрифтов на этапе выполнения»).
10. Нажмите кнопку OK.

Среда разработки Flash позволяет встраивать начертания для любого шрифта, который отображается в раскрывающемся списке Font (Шрифт) окна Font Symbol Properties (Свойства символа шрифта).

Чтобы экспортировать шрифт без сглаживания, добавьте следующий шаг между шагами 2 и 3 описанной процедуры: в окне Font Symbol Properties (Свойства символа шрифта) (шаг 2) установите флажок Bitmap text (Растровый текст) и затем укажите размер шрифта.

Когда установлен флажок Bitmap text (Растровый текст), при вычислении начертаний символов компилятор связывает фигуры с целыми пикселами. В результате для каждого символа указанного размера образуется четкая векторная фигура без сглаживания. Чтобы добиться наилучших результатов при использовании параметра Bitmap text (Растровый текст), размер шрифта, устанавливаемый для текста, который форматируется с помощью встраиваемого шрифта, должен всегда равняться размеру шрифта, указываемому в окне Font Symbols Properties (Свойства символа шрифта). Кроме того, избегайте масштабирования текста, к которому применяется форматирование с использованием встраиваемого шрифта.

В приложении Flex Builder 2 и компиляторе mxmlc параметр Bitmap text (Растровый текст) отсутствует.

Встраивание начертаний шрифта в приложении Flex Builder 2 и компиляторе mxmlc

Для встраивания начертаний шрифта в проект ActionScript приложения Flex Builder 2 или с помощью консольного компилятора mxmlc мы используем тег метаданных `[Embed]`. Чтобы использовать его, мы должны предоставить компилятору доступ к библиотеке `flex.swc`, поддерживающей компилятор Flex. По умолчанию все проекты приложения Flex Builder 2 автоматически включают местоположение файла `flex.swc` в список путей к библиотекам языка ActionScript, поэтому

в приложении Flex Builder 2 все методики, описываемые в данном разделе, будут работать без специальной настройки компилятора.

Элементы, встраиваемые с помощью тега метаданных `[Embed]`, включая шрифты, могут быть встроены как на уровне переменной, так и на уровне класса. Тем не менее встраивание шрифта на уровне переменной более удобно, чем встраивание шрифта на уровне класса, поэтому шрифты редко встраиваются на уровне класса.

Дополнительную информацию о теге метаданных [Embed] можно найти в гл. 28.

Обобщенный код, необходимый для встраивания шрифта на уровне переменной в проекте ActionScript приложения Flex Builder 2 или с помощью компилятора mxmlc, выглядит следующим образом:

```
[Embed(source="путьКШрифту",
       fontFamily="имяШрифта")]
private var имяКласса:Class;
```

В приведенном коде, который должен размещаться в теле класса, *путьКШрифту* обозначает путь к файлу шрифта в локальной файловой системе, *имяШрифта* — произвольное имя, по которому можно будет обращаться к шрифту из приложения, а *имяКласса* — имя переменной, которая будет ссылаться на класс, представляющий встроенный шрифт. Этот класс применяется только при загрузке шрифтов на этапе выполнения, как станет известно далее из подразд. «Загрузка шрифтов на этапе выполнения».

Например, следующий код демонстрирует, как встроить начертания шрифта Verdana в операционную систему Windows XP. Обратите внимание, что в значении *путьКШрифту* должны использоваться прямые слэши, но при этом оно не чувствительно к регистру символов.

```
[Embed(source="c:/windows/fonts/verdana.ttf",
       fontFamily="Verdana")]
private var verdana:Class;
```

При выполнении предыдущего кода среда Flash автоматически генерирует класс, представляющий элемент встроенного шрифта, и присваивает этот класс переменной `verdana`.

Тег метаданных [Embed] может применяться только для шрифтов TrueType.

В простых случаях код, который встраивает шрифт, находится в том же классе, который использует этот шрифт для форматирования текста. Это демонстрирует листинг 27.11, в котором представлен простой класс `HelloWorldVerdana`, отображающий текст «Hello world», отформатированный с помощью встроенного шрифта (более подробно о форматировании текста с помощью встраиваемых шрифтов мы узнаем в следующем разделе).

Листинг 27.11. Отображение Hello World с использованием шрифта Verdana

```
package {
  import flash.display.*;
  import flash.text.*;

  public class HelloWorldVerdana extends Sprite {
    // Встраиваем шрифт Verdana
    [Embed(source="c:/windows/fonts/verdana.ttf",
           fontFamily="Verdana")]
    private var verdana:Class;

    public function HelloWorldVerdana ( ) {
      var t:TextField = new TextField( );
      t.embedFonts = true;
      // Форматируем текст с помощью шрифта Verdana
      t.htmlText = "<FONT FACE='Verdana'>Hello world</FONT>";
      addChild(t);
    }
  }
}
```

В более сложных приложениях, использующих множество встраиваемых шрифтов, за встраивание всех шрифтов обычно отвечает один центральный класс — благодаря этому происходит отделение кода, отвечающего за встраивание шрифта, от кода, осуществляющего форматирование текста. Это демонстрирует листинг 27.12, в котором представлено два класса: `FontEmbedder`, отвечающий за встраивание шрифта, и `HelloWord` — основной класс, форматирующий текст с помощью шрифта, встроенного через класс `FontEmbedder`. Обратите внимание, что класс `HelloWorld` по необходимости создает ссылку на класс `FontEmbedder`, заставляя класс `FontEmbedder` и все его шрифты скомпилироваться в SWF-файл.

Листинг 27.12. Централизованное встраивание шрифтов

```
// Класс FontEmbedder
package {
  // Встраивает шрифты для данного приложения
  public class FontEmbedder {
    [Embed(source="c:/windows/fonts/verdana.ttf",
           fontFamily="Verdana")]
    private var verdana:Class;
  }
}

// Класс HelloWorld
package {
  import flash.display.*;
  import flash.text.*;

  public class HelloWorld extends Sprite {
    // Создаем ссылку на класс, который встраивает шрифты для данного
    // приложения. Эта ссылка заставляет скомпилировать класс
    // и, соответственно, его шрифты в SWF-файл.
    FontEmbedder;
```

```
    public function HelloWorld ( ) {
      var t:TextField = new TextField( );
      t.embedFonts = true;
      t.htmlText = "<FONT FACE='Verdana'>Hello world</FONT>";
      addChild(t);
    }
  }
}
```

Для сравнения в листинге 27.13 представлен код, демонстрирующий методику встраивания шрифта на уровне класса. Обратите внимание, что класс, использующий встроенный шрифт, должен ссылаться на класс, который встраивает этот шрифт.

Листинг 27.13. Встраивание шрифта на уровне класса

```
// Класс, встраивающий шрифт
package {
  import flash.display.*;
  import mx.core.FontAsset;

  [Embed(source="c:/windows/fonts/verdana.ttf", fontFamily="Verdana")]
  public class Verdana extends FontAsset {
  }
}

// Класс, использующий встроенные шрифты
package {
  import flash.display.*;
  import flash.text.*;

  public class HelloWorld extends Sprite {
    // Создаем ссылку на класс, который встраивает шрифты для данного
    // приложения. Эта ссылка заставляет скомпилировать класс
    // и, соответственно, его шрифты в SWF-файл.
    Verdana;

    // Конструктор
    public function HelloWorld ( ) {
      var t:TextField = new TextField( );
      t.embedFonts = true;
      t.htmlText = "<FONT FACE='Verdana'>Hello world</FONT>";
      addChild(t);
    }
  }
}
```

Стоит отметить, что из-за ошибки в приложении Flex Builder 2 и компиляторе mxmlc для шрифтов, встраиваемых с помощью синтаксиса тега [Embed], нельзя использовать кернинг. Тем не менее он может быть использован для шрифтов, встраиваемых с помощью среды разработки Flash. Если кернинг необходимо применять в приложении, скомпилированном с помощью приложения Flex Builder 2 или компилятора mxmlc, встройте желаемый шрифт в SWF-файл с помощью среды разработки Flash, а затем загрузите этот шрифт динамически (дополнительные сведения можно получить в подразд. «Загрузка шрифтов на этапе выполнения»).

Теперь, когда мы узнали, как встраивать шрифты с помощью среды разработки Flash и тега метаданных [Embed], рассмотрим вопросы форматирования текста с использованием встраиваемых шрифтов.

Форматирование текста с помощью встраиваемого шрифта

Чтобы отформатировать некоторый объект TextField с помощью встраиваемых шрифтов, мы должны сначала присвоить переменной embedFonts данного объекта значение true. Когда это сделано, приложение Flash Player знает, что при отображении содержимого данного текстового поля необходимо использовать встраиваемые шрифты. Это демонстрирует следующий код:

```
// Создаем объект TextField
var t:TextField = new TextField( );

// Говорим приложению Flash Player использовать встраиваемые шрифты
// при отображении содержимого объекта t
t.embedFonts = true;
```

Само по себе присваивание переменной embedFonts значения true не вызывает добавления никаких шрифтов в SWF-файл. Эта операция просто означает, что текстовое поле должно отображаться с помощью встраиваемых шрифтов, если они доступны.

Переменная embedFonts должна устанавливаться по отдельности для каждого текстового поля, использующего конкретный шрифт, даже если один и тот же шрифт применяется для нескольких полей. Тем не менее, когда несколько полей используют один и тот же встраиваемый шрифт, размер файла не увеличивается — вместе с SWF-файлом загружается всего одна копия шрифта.

Присвоив переменной embedFonts объекта TextField значение true, мы устанавливаем шрифт для текстового поля, используя переменную экземпляра font класса TextFormat, атрибут face тега <font> или свойство fontFamily языка CSS, как было рассмотрено ранее в разд. «Форматирование текстовых полей». Например:

```
// Устанавливаем шрифт с помощью объекта TextFormat
var format:TextFormat = new TextFormat( );
format.font = "названиеШрифта";
var t:TextField = new TextField( );
t.embedFonts = true;
t.defaultTextFormat = format;
t.text = "hello world";

// Устанавливаем шрифт с помощью разметки HTML
var t:TextField = new TextField( );
t.embedFonts = true;
t.htmlText = "<FONT FACE='названиеШрифта'>Hello world</FONT>";

// Устанавливаем шрифт с помощью языка CSS
var styleSheet:StyleSheet = new StyleSheet( );
var pStyle:Object = new Object( );
pStyle.fontFamily = "названиеШрифта";
```

```
styleSheet.setStyle("p", pStyle);
var t:TextField = new TextField( );
t.embedFonts = true;
t.styleSheet = styleSheet; // Присваиваем значение переменной styleSheet
                           // до того, как будет присвоено значение
                           // переменной htmlText!
t.htmlText = "<p>hello world</p>";
```

В приведенном коде *названиеШрифта* обозначает название встраиваемого шрифта, которое определяется инструментом, использованным для компиляции SWF-файла, содержащего этот шрифт.

Для шрифтов, встраиваемых с помощью тега метаданных `[Embed]`, *названиеШрифта* должно соответствовать строковому значению, указываемому для параметра `fontFamily` тега `[Embed]`, который применяется для встраивания шрифта.

Для шрифтов, встраиваемых с помощью среды разработки Flash, *названиеШрифта* должно соответствовать названию, выбранному в списке Font (Шрифт) окна Font Symbol Properties (Свойства символа шрифта), которое используется для встраивания шрифта (см. описание шага 3 в подразд. «Встраивание начертаний шрифта в среде разработки Flash»). Для шрифтов, которые встраиваются при установленном флажке Bitmap text (Растровый текст), *названиеШрифта* должно соответствовать следующему шаблону:

названиеВСпискеШрифт_размерШрифтаpt_кодВариации

В этом шаблоне *названиеВСпискеШрифт* — название, выбранное в списке Font (Шрифт) окна Font Symbol Properties (Свойства символа шрифта), *размерШрифта* — размер шрифта, установленный в окне Font Symbol Properties (Свойства символа шрифта), а *кодВариации* — одно из значений `st` (обычный), `b` (полужирный), `i` (курсив) или `bi` (полужирный курсив), соответствующее выбранной вариации шрифта в окне Font Symbol Properties (Свойства символа шрифта).

Описанный шаблон применяется в приложениях Flash CS3 и Flash Player 9, но в будущем он может измениться. С другой стороны, чтобы обеспечить обратную совместимость, гипотетические будущие версии приложения Flash Player будут продолжать поддерживать этот шаблон.

В табл. 27.8 приведено несколько примеров предыдущего шаблона *названиеШрифта*, который должен применяться при установленном флажке Bitmap text (Растровый текст).

Таблица 27.8. Примеры шаблона, применяемого при установленном флажке Bitmap text (Растровый текст)

Название в списке Font (Шрифт)	Вариация шрифта	Размер шрифта	Пример значения названиеШрифта
Verdana	Обычный	12	Verdana_12pt_st
Verdana	Полужирный	12	Verdana_12pt_b
Verdana	Курсив	12	Verdana_12pt_i
Verdana	Полужирный курсив	12	Verdana_12pt_bi

Стоит отметить, что общая методика выбора шрифта символа не зависит от того, какой шрифт используется для отображения текста — шрифт устройства или встроенный шрифт. В случае со шрифтами устройства указываемое название должно соответствовать названию шрифта, установленного в системе конечного пользователя. В случае со встраиваемыми шрифтами указываемое название должно соответствовать названию встроенного шрифта.

Использование полужирного начертания и курсива со встраиваемыми шрифтами. Чтобы использовать вариации шрифта полужирного начертания, курсива или полужирного курсива в объекте `TextField`, переменной `embedFonts` которого присвоено значение `true`, мы должны встроить эти вариации по отдельности. Например, если мы используем вариации шрифта Arial полужирного начертания, курсива и полужирного курсива в объекте `TextField`, переменной `embedFonts` которого присвоено значение `true`, мы должны встроить все три вариации шрифта Arial.

Каждой вариации шрифта, встраиваемой с помощью среды разработки Flash, должно быть присвоено уникальное имя класса в окне Font Symbol Properties (Свойства символа шрифта). Подобным образом, название каждой вариации шрифта, встраиваемой с помощью тега метаданных `[Embed]`, должно соответствовать имени ее собственной переменной (для шрифтов, встраиваемых на уровне переменной) или класса (для шрифтов, встраиваемых на уровне класса). Более того, каждая вариация некоторого шрифта должна указывать одно и то же значение для параметра `fontFamily` тега `[Embed]` и использовать соответствующий параметр вариации шрифта (либо `fontWeight`, либо `fontStyle`), чтобы определить встраиваемую вариацию.

Например, следующий код встраивает вариации шрифта Verdana полужирного начертания и курсива. Для вариации шрифта полужирного начертания переменной `fontFamily` присваивается значение `"Verdana"`, а переменной `fontStyle` — значение `"italic"`. Обратите внимание, что параметр *источник* для каждой инструкции встраивания обозначает местоположение файла шрифта, содержащего подходящую вариацию шрифта (`verdanab.ttf` и `verdanai.ttf` соответственно).

```
[Embed(source="c:/windows/fonts/verdanab.ttf",
       fontFamily="Verdana",
       fontWeight="bold")]
private var verdanaBold:Class;

[Embed(source="c:/windows/fonts/verdanai.ttf",
       fontFamily="Verdana",
       fontStyle="italic")]
private var verdanaItalic:Class;
```

Для справки в листинге 27.14 представлен код, необходимый для встраивания и использования вариаций шрифта Verdana обычного и полужирного начертания.

Листинг 27.14. Встраивание нескольких вариаций шрифта

```
// Класс, встраивающий шрифт
package {
  public class FontEmbedder {
```

```
    // Встраиваем вариацию обычного начертания
    [Embed(source="c:/windows/fonts/verdana.ttf",
           fontFamily="Verdana")]
    private var verdana:Class;

    // Встраиваем вариацию полужирного начертания
    [Embed(source="c:/windows/fonts/verdanab.ttf",
           fontFamily="Verdana",
           fontWeight="bold")]
    private var verdanabold:Class;
  }
}

// Класс, использующий встроенные шрифты
package {
  import flash.display.*;
  import flash.text.*;

  public class HelloWorld extends Sprite {
    // Заставляет скомпилировать класс FontEmbedder и, соответственно,
    // его шрифты в SWF-файл.
    FontEmbedder;

    public function HelloWorld ( ) {
      var t:TextField = new TextField( );
      t.embedFonts = true;
      // Используем две вариации шрифта Verdana (обычного и полужирного
      // начертания)
      t.htmlText = "<FONT FACE='Verdana'>Hello <b>world</b></FONT>";

      addChild(t);
    }
  }
}
```

Загрузка шрифтов на этапе выполнения

Представим, что мы создаем приложение для приобретения туристических услуг, которое позволяет пользователю приобрести авиабилет, оплатить комнату в отеле и услуги по трансферу из аэропорта в отель и обратно. Каждый раздел приложения имеет собственный дизайн, использующий свои шрифты. В некоторых случаях пользователи приобретают только авиабилет и полностью пропускают разделы приложения, относящиеся к оплате комнаты в отеле и услуг по доставке.

Чтобы ускорить процесс начальной загрузки приложения для приобретения туристических услуг, мы можем отложить загрузку шрифтов до тех пор, пока они на самом деле не потребуются приложению. Непосредственно перед обращением пользователя к конкретному разделу приложения мы загружаем шрифты, необходимые для данного раздела. Таким образом, пользователи, которые обращаются только к одному разделу, загружают шрифты, необходимые только для него,

и перед началом использования приложения им не придется ждать, пока будут загружены шрифты для остальных разделов.

Чтобы загрузить шрифты на этапе выполнения, повторите следующие обобщенные шаги.

1. Встройте шрифт (-ы) в SWF-файл (с помощью методик, описанных ранее в подразд. «Встраивание начертаний шрифта в среде разработки Flash» и «Встраивание начертаний шрифта в приложении Flex Builder 2 и компиляторе mxmlc»).
2. В SWF-файле, содержащем встроенный шрифт, используйте статический метод `registerFont( )` класса `Font`, чтобы добавить шрифт в глобальный список шрифтов.
3. Загрузите SWF-файл со встроенным шрифтом.

Применим перечисленные шаги на примере. Мы начнем с создания SWF-файла `Fonts.swf`, который встраивает шрифт Verdana (обычного и полужирного начертания) с помощью тега метаданных `[Embed]`. Рассмотрим код для основного класса приложения `Fonts.swf`:

```
package {
  import flash.display.*;
  import flash.text.*;

  // Встраивает шрифты для использования в любом SWF-файле,
  // загрузившем этот файл
  public class Fonts extends Sprite {
    [Embed(source="c:/windows/fonts/verdana.ttf",
           fontFamily="Verdana")]
    private var verdana:Class;

    [Embed(source="c:/windows/fonts/verdanab.ttf",
           fontFamily="Verdana",
           fontWeight="bold")]
    private var verdanaBold:Class;
  }
}
```

Теперь мы должны добавить наши встраиваемые шрифты в глобальный список шрифтов. Для этого мы используем статический метод `registerFont( )` класса `Font`, который принимает единственный параметр `font`. В качестве параметра `font` передается ссылка на класс `Font`, который представляет шрифт, добавляемый в глобальный список шрифтов. Как только шрифт будет добавлен, он может быть использован любым SWF-файлом, выполняющимся в приложении Flash Player.

В предыдущем коде классы, представляющие две вариации шрифта Verdana, присваиваются переменным `verdana` и `verdanaBold`. Таким образом, чтобы добавить эти шрифты в глобальный список шрифтов, мы передаем значения этих переменных в метод `registerFont( )`, как показано в следующем коде:

```
Font.registerFont(verdana);
Font.registerFont(verdanaBold);
```

Чтобы гарантировать, что шрифты будут добавлены в глобальный список шрифтов сразу после их загрузки, мы вызываем метод `registerFont( )` внутри конструктора класса `Fonts`, как показано в следующем коде:

```
package {
  import flash.display.*;
  import flash.text.*;

  // Встраивает шрифты для дальнейшего использования любым SWF-файлом,
  // загрузившим этот файл
  public class Fonts extends Sprite {
    [Embed(source="c:/windows/fonts/verdana.ttf",
           fontFamily="Verdana")]
    private var verdana:Class;

    [Embed(source="c:/windows/fonts/verdanab.ttf",
           fontFamily="Verdana",
           fontWeight="bold")]
    private var verdanaBold:Class;

    // Конструктор
    public function Fonts ( ) {
      // Регистрируем встроенные шрифты этого класса
      // в глобальном списке шрифтов
      Font.registerFont(verdana);
      Font.registerFont(verdanaBold);
    }
  }
}
```

Если бы мы встраивали наши шрифты с помощью символов `Font` среды разработки Flash, нам бы пришлось добавить предыдущие вызовы метода `registerFont( )` в первый кадр основной временной шкалы, а в метод `registerFont( )` мы бы передавали имя класса шрифта, указываемое в поле **Class** (Класс) окна **Linkage Properties** (Свойства связывания) для каждого встраиваемого символа `Font` (обратитесь к описанию шага 8 в подразд. «Встраивание начертаний шрифта в среде разработки Flash»).

После этого мы компилируем файл `Fonts.swf` и загружаем его на этапе выполнения с помощью класса `Loader`. Сразу после завершения загрузки файла `Fonts.swf` его шрифты могут быть использованы любым другим SWF-файлом, выполняющимся в приложении Flash Player. В листинге 27.15 представлен пример класса, который загружает и затем использует шрифты, встроенные в файл `Fonts.swf`.

Исчерпывающую информацию о загрузке SWF-файлов можно найти в гл. 28.

Листинг 27.15. Использование загруженных шрифтов

```
package {
  import flash.display.*;
  import flash.text.*;
```

```
import flash.events.*;
import flash.net.*;

// Этот класс демонстрирует, как форматировать текст с помощью загруженных
// шрифтов. Сами шрифты встроены в файл Fonts.swf, представленный ранее.
public class HelloWorld extends Sprite {
  public function HelloWorld ( ) {
    // Загружаем SWF-файл, содержащий встроенные шрифты
    var loader:Loader = new Loader( );
    loader.contentLoaderInfo.addEventListener(Event.INIT, initListener);
    loader.load(new URLRequest("Fonts.swf"));
  }

  // Выполняется после завершения инициализации файла Fonts.swf, когда
  // его шрифты будут доступны для использования
  private function initListener (e:Event):void {
    // Для отладочных целей отображаем все доступные встроенные шрифты
    showEmbeddedFonts( );

    // Шрифт был загружен, поэтому теперь отображаем
    // отформатированный текст
    outputMsg( );
  }

  // Отображаем текст, отформатированный с использованием
  // встроенных шрифтов
  private function outputMsg ( ):void {
    // Создаем текстовое поле
    var t:TextField = new TextField( );
    t.embedFonts = true;  // Говорим среде выполнения Flash отображать это
                          // текстовое поле с помощью встроенных шрифтов
    // Используем две вариации шрифта Verdana (обычного и полужирного
    // начертания)
    t.htmlText = "<FONT FACE='Verdana'>Hello <b>world</b></FONT>";

    // Добавляем текстовое поле в список отображения
    addChild(t);
  }

  // Выводит список доступных на настоящий момент встроенных шрифтов
  public function showEmbeddedFonts ( ):void {
    trace("========Embedded Fonts========");

    var fonts:Array = Font.enumerateFonts( );
    fonts.sortOn("fontName", Array.CASEINSENSITIVE);
    for (var i:int = 0; i < fonts.length; i++) {
      trace(fonts[i].fontName + ", " + fonts[i].fontStyle);
    }
  }
 }
}
```

Многие браузеры кэшируют SWF-файлы, поэтому для приложений, состоящих из нескольких SWF-файлов, можно добиться снижения общего времени загрузки, загружая шрифты из одного SWF-файла на этапе выполнения.

Отсутствующие шрифты и глифы

Как мы узнали ранее, если при отображении текстового поля с помощью шрифтов устройства шрифт для некоторого символа не установлен в операционной системе конечного пользователя, приложение Flash Player автоматически попросит ее отобразить символ с помощью подходящего замещающего шрифта.

В отличие от этого, если текстовое поле отображается с помощью встраиваемых шрифтов и шрифт для некоторого символа недоступен в списке встроенных шрифтов, приложение Flash Player сначала попытается отобразить этот символ с использованием любой доступной вариации указанного шрифта. Например, рассмотрим следующий код, который использует две вариации шрифта Verdana:

```
var t:TextField = new TextField( );
t.embedFonts = true;
t.htmlText = "<FONT FACE='Verdana'>Hello <b>world</b></FONT>";
```

Обратите внимание, что для слова «Hello» указана вариация шрифта Verdana обычного начертания, а для слова «world» указана вариация шрифта Verdana полужирного начертания. Если на этапе выполнения вариация встроенного шрифта Verdana полужирного начертания окажется недоступной, но при этом *доступной* окажется вариация встроенного шрифта Verdana обычного начертания, текст «Hello world» будет полностью отображен с использованием вариации шрифта Verdana обычного начертания. Однако если не будут доступны *ни* вариация обычного начертания, *ни* вариация полужирного начертания шрифта Verdana, символы не будут отображены вообще и никакой текст на экране не появится!

Если при использовании встраиваемых шрифтов текст в вашем приложении таинственным образом исчезает или отображается с помощью неправильной вариации шрифта, то, скорее всего, оказались недоступны необходимые шрифты. Чтобы определить, какие шрифты доступны на этапе выполнения, используйте статический метод enumerateFonts( ) класса Font, рассматриваемый в разд. «Определение доступности шрифта».

Если при использовании встраиваемых шрифтов текстовое поле содержит символ, глиф которого отсутствует в указанном шрифте, этот символ отображен не будет. В отличие от этого, если при использовании шрифтов устройства текстовое поле содержит символ, глиф которого отсутствует в указанном шрифте, приложение Flash Player автоматически выполнит поиск замещающего шрифта, содержащего отсутствующий глиф. Если такой шрифт будет найден, то символ будет отображен с помощью замещающего шрифта. Если замещающий шрифт найден не будет, то символ не будет отображен на экране.

Если переменной `embedFonts` объекта `TextField` присвоено значение `true`, но программа не предоставляет никакой информации по форматированию этого объ-

екта, Flash Player попытается отобразить его содержимое с помощью встроенного шрифта, имя которого совпадает с именем шрифта, используемого по умолчанию в текущей среде (в операционной системе Microsoft Windows таким шрифтом является Times New Roman). Если такого встроенного шрифта не существует, то текст не будет выведен на экран.

Определение доступности шрифта

Чтобы на этапе выполнения определить список доступных шрифтов устройства и встроенных шрифтов, используйте статический метод `enumerateFonts( )` класса `Font`. Метод `enumerateFonts( )` возвращает массив объектов `Font`, каждый из которых представляет доступный шрифт устройства или встроенный шрифт. Метод `enumerateFonts( )` определяет единственный параметр типа `Boolean enumerateDeviceFonts`, который указывает, должен ли возвращаемый массив включать шрифты устройства. По умолчанию параметру `enumerateDeviceFonts` присвоено значение `false`, поэтому массив, возвращаемый методом `enumerateFonts( )`, не включает шрифты устройства. Каждый объект `Font` в возвращаемом массиве определяет следующие переменные, описывающие представляемый шрифт.

- `fontName` — имя шрифта. Для шрифтов устройства переменная `fontName` содержит имя, которое отображается в списке системных шрифтов. Для шрифтов, встраиваемых через среду разработки Flash, переменная `fontName` содержит имя, которое отображается в списке **Font** (Шрифт) окна **Font Symbol Properties** (Свойства символа шрифта), применяемого для встраивания шрифта. Для шрифтов, встраиваемых с помощью тега метаданных `[Embed]`, переменная `fontName` содержит строковое значение, которое указывается для параметра `fontFamily` тега `[Embed]`, применяемого для встраивания шрифта.
- `fontStyle` — вариация шрифта (обычное начертание, полужирное начертание, курсив или полужирный курсив) в виде одной из четырех следующих констант языка ActionScript: `FontStyle.REGULAR`, `FontStyle.BOLD`, `FontStyle.ITALIC`, `FontStyle.BOLD_ITALIC`.
- `fontType` — тип шрифта: встроенный или шрифт устройства. Эта переменная ссылается на одну из двух следующих констант языка ActionScript: `FontType.EMBEDDED` или `FontType.DEVICE`.

В листинге 27.16 представлен код, генерирующий список всех доступных встроенных шрифтов, отсортированных в алфавитном порядке.

Листинг 27.16. Перечисление всех встроенных шрифтов

```
var fonts:Array = Font.enumerateFonts( );
fonts.sortOn("fontName", Array.CASEINSENSITIVE);
for (var i:int = 0; i < fonts.length; i++) {
  trace(fonts[i].fontName + ", " + fonts[i].fontStyle);
}
```

В листинге 27.17 продемонстрирован код, генерирующий список всех доступных шрифтов устройства, которые отсортированы в алфавитном порядке.

Листинг 27.17. Перечисление всех шрифтов устройства

```
var fonts:Array = Font.enumerateFonts(true);
fonts.sortOn("fontName", Array.CASEINSENSITIVE);
for (var i:int = 0; i < fonts.length; i++) {
  if (fonts[i].fontType == FontType.DEVICE) {
    trace(fonts[i].fontName + ", " + fonts[i].fontStyle);
  }
}
```

В листинге 27.18 представлен код, генерирующий список всех доступных встроенных шрифтов и шрифтов устройства, отсортированных в алфавитном порядке.

Листинг 27.18. Перечисление всех встроенных шрифтов и шрифтов устройства

```
var fonts:Array = Font.enumerateFonts(true);
fonts.sortOn("fontName", Array.CASEINSENSITIVE);
for (var i:int = 0; i < fonts.length; i++) {
    trace(fonts[i].fontType + ": "
          + fonts[i].fontName + ", " + fonts[i].fontStyle);
}
```

Функция `enumerateFonts( )` может быть использована, чтобы дать возможность пользователю выбирать шрифты приложения или автоматически выбирать резервный шрифт, как показано в листинге 27.19.

Листинг 27.19. Автоматический выбор резервного шрифта

```
package {
  import flash.display.*;
  import flash.text.*;

  public class FontFallbackDemo extends Sprite {
    public function FontFallbackDemo ( ) {
      var format:TextFormat = new TextFormat( );

      // Присваивает первый доступный шрифт
      format.font = getFont(["ZapfChancery", "Verdana", "Arial", "_sans"]);

      var t:TextField = new TextField( );
      t.text = "ActionScript is fun!";
      t.autoSize = TextFieldAutoSize.LEFT;
      t.setTextFormat(format)

      addChild(t);
    }

    // Из передаваемого списка шрифтов возвращает имя первого шрита
    // в списке, который доступен либо в качестве встроенного шрифта, либо
    // в качестве шрифта устройства
    public function getFont (fontList: Array):String {
      var availableFonts:Array = Font.enumerateFonts(true);
      for (var i:int = 0; i < fontList.length; i++) {
        for (var j:int = 0; j < availableFonts.length; j++) {
```

```
            if (fontList[i] == Font(availableFonts[j]).fontName) {
              return fontList[i];
            }
          }
        }
        return null;
      }
    }
}
```

Определение доступности глифа

Чтобы определить, имеет ли некий встроенный шрифт глиф для указанного символа или набора символов, мы используем метод экземпляра `hasGlyphs( )` класса `Font`. Принимая строковый аргумент, этот метод возвращает значение типа `Boolean`, которое указывает, имеет ли шрифт все глифы, необходимые для отображения данной строки.

Метод экземпляра hasGlyphs( ) класса Font работает только со встраиваемыми шрифтами. Не существует способа, позволяющего определить, имеет ли некий шрифт устройства глиф для указанного символа.

Чтобы воспользоваться методом `hasGlyphs( )`, мы должны сначала получить ссылку на объект `Font` для интересующего нас шрифта. Для этого мы применяем цикл `for`, чтобы осуществить поиск в массиве, возвращаемом методом `enumerateFonts( )`. Например, следующий код получает ссылку на объект `Font` для шрифта Verdana и присваивает ее переменной `font`:

```
var fontName:String = "Verdana";
var font:Font;
var fonts:Array = Font.enumerateFonts(true);
for (var i:int = 0; i < fonts.length; i++) {
  if (fonts[i].fontName == fontName) {
    font = fonts[i];
    break;
  }
}
```

Как только будет получена ссылка на желаемый объект `Font`, мы можем использовать метод `hasGlyphs( )`, чтобы проверить, имеет ли соответствующий шрифт все глифы, необходимые для отображения некоторой строки. Например, следующий код проверяет, можно ли с помощью шрифта Verdana отобразить строку «Hello world» на английском языке:

```
trace(font.hasGlyphs("Hello world"));  // Выводит: true
```

Следующий код проверяет, можно ли с помощью шрифта Verdana отобразить строку «みんなさん、こんにちは» на японском языке:

```
trace(font.hasGlyphs(みんなさん、こんにちは));  // Выводит: false
```

Отображение текста с помощью встраиваемых шрифтов

Как бы удивительно это ни звучало, для объектов `TextField`, переменной `embedFonts` которых присвоено значение `true`, приложение Flash Player предоставляет два различных режима отображения текста. Названия этих режимов звучат достаточно размыто: *обычный* и *расширенный*.

В обычном режиме приложение Flash Player отображает текст с помощью стандартного визуализатора векторной графики, который применяется для отображения всех векторных фигур в SWF-файле. Стандартный визуализатор векторной графики рисует текст с использованием алгоритма сглаживания, который характеризуется высокой скоростью и создает гладкие линии. Текст, отображаемый с помощью стандартного визуализатора векторной графики, обычно получается четким и разборчивым при средних и больших размерах шрифта (приблизительно 16 пунктов и выше), однако при малых размерах шрифта (12 пунктов и меньше) текст оказывается размытым и неразборчивым.

В расширенном режиме приложение Flash Player выводит текст с помощью специализированного визуализатора текста, называемого визуализатором FlashType. Визуализатор FlashType — это лицензированная реализация системы Saffron Type System, созданной лабораторией Mitsubishi Electric Research Laboratories (MERL). Визуализатор FlashType специально разработан для того, чтобы обеспечивать четкую визуализацию типов фигур, которые часто встречаются в шрифтах небольших размеров.

В настоящее время наилучшие результаты при использовании визуализатора FlashType достигаются для западных шрифтов, нежели для азиатских. Тем не менее азиатский текст, отображаемый с помощью визуализатора FlashType, все равно оказывается более разборчивым, чем текст, отображаемый с помощью стандартного визуализатора векторной графики приложения Flash Player.

Текст, выводимый с помощью визуализатора FlashType, обычно получается более разборчивым, чем текст, выводимый с помощью стандартного визуализатора векторной графики приложения Flash Player. Кроме того, при небольших размерах шрифта визуализатор FlashType отображает текст быстрее, чем стандартный визуализатор векторной графики приложения Flash Player. Тем не менее при больших размерах шрифта отображение текста с помощью визуализатора FlashType занимает значительно больше времени, чем отображение текста с помощью стандартного визуализатора векторной графики приложения Flash Player.

Дополнительную информацию о системе Saffron Type System можно получить в официальном обзоре проекта Saffron компании Mitsubishi, доступном по адресу http://www.merl.com/projects/ADF-Saffron, а также в технической презентации проекта Saffron Рональда Перри (Ronald Perry), доступной по адресу http://www.merl.com/people/perry/SaffronOverview.ppt.

Разработчики могут выбирать любой из двух режимов визуализации текста приложения Flash Player динамически на этапе выполнения для каждого текстового

поля по отдельности. Чтобы приложение Flash Player отображало некоторый объект `TextField` с помощью стандартного визуализатора векторной графики, присвойте переменной `antiAliasType` данного объекта значение `AntiAliasType.NORMAL`. Например, следующий код создает объект `TextField`, а затем говорит приложению Flash Player отобразить этот объект с помощью встроенных шрифтов, используя стандартный визуализатор векторной графики. Обратите внимание, что помимо присваивания значения переменной `antiAliasType` этот код присваивает переменной `embedFonts` объекта `TextField` значение `true`. Режимы визуализации текста приложения Flash Player применяются только к тексту, отображаемому с помощью встраиваемых шрифтов.

```
// Создаем объект TextField
var t:TextField = new TextField( );

// Говорим приложению Flash Player отобразить это текстовое поле
// с помощью встроенных шрифтов
t.embedFonts = true;

// Говорим приложению Flash Player использовать стандартный визуализатор
// векторной графики для отображения этого текстового поля
t.antiAliasType = AntiAliasType.NORMAL;
```

В отличие от этого, следующий код создает объект `TextField`, а затем говорит приложению Flash Player отобразить этот объект с помощью встроенных шрифтов, используя визуализатор FlashType:

```
// Создаем объект TextField
var t:TextField = new TextField( );

// Говорим приложению Flash Player отобразить это текстовое поле
// с помощью встроенных шрифтов
t.embedFonts = true;

// Говорим приложению Flash Player использовать визуализатор FlashType
// для отображения этого текстового поля
t.antiAliasType = AntiAliasType.ADVANCED;
```

По умолчанию переменной `antiAliasType` присвоено такое значение, как `AntiAliasType.NORMAL` (стандартный визуализатор векторной графики).

По умолчанию приложение Flash Player отображает объекты TextField, переменной embedFonts которых присвоено значение true, с помощью стандартного визуализатора векторной графики.

На рис. 27.13 показан английский алфавит, для отображения которого с помощью шрифта Verdana размером 10 пунктов использовались оба визуализатора: визуализатор FlashType (слева) и стандартный визуализатор векторной графики (справа). На экране алфавит, отображенный с помощью визуализатора FlashType, в значительной степени более разборчив, чем алфавит, отображенный с помощью стандартного визуализатора векторной графики приложения Flash Player.

Для справки в листинге 27.20 показан код, который применялся для создания демонстрационных алфавитов, изображенных на рис. 27.13.

FlashType	Стандартный визуализатор векторной графики
abcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz

Рис. 27.13. Визуализатор FlashType в сравнении со стандартным визуализатором векторной графики приложения Flash Player

Листинг 27.20. Визуализатор FlashType в сравнении со стандартным визуализатором векторной графики приложения Flash Player

```
package {
  import flash.display.*;
  import flash.text.*;

  public class FlashTypeDemo extends Sprite {
    // Прямые слэши необходимы, однако регистр символов роли не играет.
    [Embed(source="c:/windows/fonts/verdana.ttf",
           fontFamily="Verdana")]
    private var verdana:Class;

    public function FlashTypeDemo ( ) {
      // Визуализатор FlashType
      var t:TextField = new TextField( );
      t.width = 200;
      t.embedFonts = true;
      t.htmlText = "<FONT FACE='Verdana' SIZE='10'>"
                   + "abcdefghijklmnopqrstuvwxyz</FONT>";
      t.antiAliasType = AntiAliasType.ADVANCED;
      addChild(t);

      // Стандартный визуализатор векторной графики
      var t2:TextField = new TextField( );
      t2.width = 200;
      t2.embedFonts = true;
      t2.htmlText = "<FONT FACE='Verdana' SIZE='10'>"
                    + "abcdefghijklmnopqrstuvwxyz</FONT>";
      t2.antiAliasType = AntiAliasType.NORMAL;
      addChild(t2);
      t2.x = 180;
    }
  }
}
```

Чтобы достичь наилучшего качества анимационного текста, используйте стандартный визуализатор векторной графики (присвойте переменной antiAliasType значение AntiAliasType.NORMAL). Чтобы добиться наилучшей разборчивости текста, используйте визуализатор FlashType (присвойте переменной antiAliasType значение AntiAliasType.ADVANCED).

Обратите внимание, что, если текст наклонен или перевернут, его отображение с помощью визуализатора FlashType автоматически отменяется.

Настройка визуализатора FlashType. Привлекательность и разборчивость текста — это весьма субъективные понятия. Язык ActionScript предлагает ряд расширенных инструментов для точной настройки конкретного поведения визуализатора FlashType.

Хотя детальное рассмотрение возможных настроек визуализатора FlashType выходит за рамки данной книги, для ознакомления в табл. 27.9 перечислены доступные инструменты и их основное назначение. Для дальнейшего изучения обратитесь к описанию каждого инструмента в справочнике по языку ActionScript корпорации Adobe.

Таблица 27.9. Переменные и методы, используемые для настройки параметров визуализатора FlashType

Переменная или метод	Описание
Переменная экземпляра sharpness класса TextField	Определяет резкость текста в текстовом поле. Может принимать целочисленные значения в диапазоне от –400 (размытый) до 400 (резкий)
Переменная экземпляра thickness класса TextField	Указывает толщину линий текста в текстовом поле. Может принимать целочисленные значения в диапазоне от –200 (тонкий) до 200 (толстый). Если переменной thickness поля присвоить большое значение, текст этого поля примет полужирное начертание
Переменная экземпляра gridFitType класса TextField	Определяет параметры хинтования на уровне пикселов, влияющие на разборчивость текста при различном выравнивании (по левому краю, по центру или по правому краю). Хинтование — это методика, с помощью которой ножки отображаемого глифа размещаются на целых пикселах, улучшая его читабельность
Статическая переменная displayMode класса TextRenderer	Настраивает алгоритм сглаживания визуализатора FlashType, улучшая результаты либо для ЖК-, либо для ЭЛТ-мониторов. Этот параметр применяется глобально ко всему тексту, отображаемому с помощью визуализатора FlashType
Статическая переменная maxLevel класса TextRenderer	Определяет уровень качества адаптивно выбираемых интервалов полей (Adaptively Sampled Distance Fields — ADFs) (часть внутренней структуры визуализатора FlashType для описания контуров глифов). Этот параметр применяется глобально ко всему тексту, отображаемому с помощью визуализатора FlashType (однако Flash Player автоматически увеличивает значение этого параметра для любого отдельного глифа, размер шрифта которого превышает 64 пиксела). Большие значения снижают производительность
Статический метод setAdvancedAntiAliasingTable( )	Присваивает значения, которые точно определяют вес и резкость конкретного шрифта при использовании указанного размера, стиля и типа цвета (светлый или темный)

Теперь перейдем от вопросов, связанных с форматированием и шрифтами, к вопросам получения данных через текстовые поля.

Ввод через текстовые поля

Текстовые поля могут получать разнообразные виды пользовательского ввода, включая ввод и выделение текста, активизацию гипертекстовой ссылки, фокус ввода с клавиатуры, прокрутку и взаимодействие с мышью. В этом разделе мы познакомимся

с вводом, выделением текста и гипертекстовыми ссылками. Информацию о фокусе ввода с клавиатуры, прокрутке и взаимодействию с мышью можно найти в гл. 22.

Ввод текста

Способность каждого текстового поля получать пользовательский ввод определяется значением переменной `type` этого поля. По умолчанию для текстовых полей, создаваемых с помощью кода на языке ActionScript, переменной экземпляра `type` присваивается значение `TextFieldType.DYNAMIC` — это значит, что текст может изменяться с помощью кода на языке ActionScript, при этом пользователь не может вносить изменения в текст. Чтобы текстовое поле могло получать пользовательский ввод, мы должны присвоить переменной `type` значение `TextFieldType.INPUT`, как показано в следующем коде:

```
var t:TextField = new TextField( );
t.type = TextFieldType.INPUT;
```

Когда переменной `type` объекта `TextField` присвоено значение `TextFieldType.INPUT`, пользователь может добавлять или удалять текст из этого текстового поля. Любые изменения, вносимые пользователем, автоматически отражаются переменными `text` и `htmlText`.

Чтобы получать уведомления о внесении пользователем изменений в текстовое поле, мы можем зарегистрировать в этом текстовом поле приемники для событий `TextEvent.TEXT_INPUT` и `Event.CHANGE`. Диспетчеризация события `TextEvent.TEXT_INPUT` происходит в том случае, когда пользователь пытается изменить текст в текстовом поле, перед тем как будут обновлены значения переменных `text` и `htmlText`. Диспетчеризация события `Event.CHANGE` происходит после того, как в ответ на пользовательский ввод будут обновлены значения переменных `text` и `htmlText`. Подробную информацию по событиям `TextEvent.TEXT_INPUT` и `Event.CHANGE` можно найти в гл. 22.

По умолчанию пользователям не разрешается вводить разрывы строк в текстовые поля. Чтобы разрешить это (например, в результате нажатия клавиши Enter), присвойте переменной `multiline` значение `true`, как показано в следующем коде:

```
var t:TextField = new TextField( );
t.type = TextFieldType.INPUT;
t.multiline = true;
```

Чтобы ограничить набор символов, которые пользователь может вводить в текстовое поле, используйте переменную экземпляра `restrict` класса `TextField`. Например, следующее текстовое поле позволяет вводить только цифры, что может потребоваться для поля ввода номера кредитной карты:

```
var t:TextField = new TextField( );
t.width = 200;
t.height = 20;
t.border     = true;
t.background = true;
t.type = TextFieldType.INPUT;
t.restrict = "0-9";
```

Чтобы ограничить количество символов, которое пользователь может ввести в текстовое поле, используйте переменную экземпляра `maxChars` класса `TextField`. Например, следующее текстовое поле позволяет ввести только восемь символов, что может потребоваться для поля ввода имени на форме регистрации:

```
var t:TextField = new TextField( );
t.width = 100;
t.height = 20;
t.border     = true;
t.background = true;
t.type = TextFieldType.INPUT;
t.maxChars = 8;
```

Для сокрытия всех вводимых символов с целью защиты экрана используйте переменную экземпляра `displayAsPassword` класса `TextField`. Когда переменной `displayAsPassword` присвоено значение `true`, все символы отображаются в виде звездочек (*). Например, слова «hi there» отображаются в виде «********». Это позволяет пользователю вводить текст, не беспокоясь о том, что посторонний человек сможет его увидеть. Следующий код демонстрирует текстовое поле, которое скрывает символы, что может потребоваться для поля ввода пароля на форме регистрации:

```
var t:TextField = new TextField( );
t.width = 100;
t.height = 20;
t.border     = true;
t.background = true;
t.type = TextFieldType.INPUT;
t.displayAsPassword = true;
```

Форматирование пользовательского ввода. По умолчанию новый текст, вводимый пользователем, автоматически принимает форматирование символа, находящегося перед точкой вставки, или символа, находящегося в позиции 0, если новый текст добавляется перед этой позицией. Если поле было пустым, новый текст форматируется в соответствии с используемым по умолчанию форматом данного поля (который задается через переменную `defaultTextFormat`, как было рассмотрено ранее в разд. «Форматирование по умолчанию для текстовых полей» разд. «Форматирование текстовых полей»).

Чтобы переопределить автоматическое форматирование, применяемое к новому текстовому вводу, выполните такую последовательность действий.

1. Перехватите ввод с помощью события `TextEvent.TEXT_INPUT`.
2. Вручную добавьте эквивалентный текст.
3. Примените форматирование к тексту, вставленному вручную.

Эта методика продемонстрирована в листинге 27.21, представляющем пример класса `FormattedInputDemo`. Понять код вам помогут комментарии.

Листинг 27.21. Форматирование пользовательского ввода

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;
```

```
public class FormattedInputDemo extends Sprite {
  public function FormattedInputDemo ( ) {
    // Создаем объекты TextFormat
    var boldFormat:TextFormat = new TextFormat( );
    boldFormat.bold = true;
    var italicFormat:TextFormat = new TextFormat( );
    italicFormat.italic = true;

    // Создаем текстовое поле
    var t:TextField = new TextField( );
    t.text = "lunchtime";

    // Форматируем слово "lunch" с использованием курсива
    t.setTextFormat(italicFormat, 0, 5);
    // Форматируем слово "time", используя полужирное начертание
    t.setTextFormat(boldFormat, 5, 9);
    t.type = TextFieldType.INPUT;

    // Регистрируем приемник для событий TextEvent.TEXT_INPUT
    // в объекте t
    t.addEventListener(TextEvent.TEXT_INPUT, textInputListener);

    // Добавляем текстовое поле в список отображения
    addChild(t);
  }

  // Вызывается всякий раз, когда пользователь пытается добавить
  // новый текст в объект t
  private function textInputListener (e:TextEvent):void {
    // Получаем ссылку на текстовое поле, получившее ввод
    var t:TextField = TextField(e.target);

    // Предотвращаем добавление текста, введенного пользователем
    // в текстовое поле
    e.preventDefault( );

    // Добавляем текст, введенный пользователем вручную. В этом случае
    // происходит немедленное обновление переменной text объекта
    // TextField, позволяя нам отформатировать новый текст внутри
    // данной функции.
    t.replaceText(t.caretIndex, t.caretIndex, e.text);

    // Устанавливаем форматирование для нового текста
    var regularFormat:TextFormat = new TextFormat( );
    regularFormat.bold   = false;
    regularFormat.italic = false;
    t.setTextFormat(regularFormat,
                    t.caretIndex,
                    t.caretIndex+e.text.length)

    // Устанавливаем точку вставки в конец нового текста, чтобы
    // пользователь считал, что это он ввел текст
```

```
            var newCaretIndex:int = t.caretIndex + e.text.length;
            t.setSelection(newCaretIndex, newCaretIndex);
        }
    }
}
```

Выделение текста

По умолчанию текст во всех текстовых полях, создаваемых программным путем, может быть выделен пользователем. Чтобы отключить возможность выделения для некоторого поля, присвойте переменной `selectable` значение `false`. Обычно выделение текстового поля показывается только в том случае, когда данное поле имеет фокус. Чтобы выделение текстового поля отображалось даже в тех случаях, когда текстовое поле не имеет фокуса, присвойте переменной `alwaysShowSelection` значение `true`. В приложении Flash Player 9 установить цвет выделения невозможно; поддержка настраиваемого цвета выделения может быть реализована в будущих версиях приложения Flash Player.

Чтобы определить индекс первого выделенного символа в текстовом поле, используйте переменную экземпляра `selectionBeginIndex` класса `TextField`. Для определения индекса последнего выделенного символа в текстовом поле применяйте переменную экземпляра `selectionEndIndex` класса `TextField`. Чтобы определить позицию точки вставки (курсора), используйте переменную экземпляра `caretIndex` класса `TextField`. Чтобы программным путем выделить символы в текстовом поле или установить точку вставки, применяйте метод экземпляра `setSelection( )` класса `TextField`.

Стоит отметить, что в приложении Flash Player отсутствуют события, обозначающие изменение выделения текстового поля. Чтобы выявить изменения в выделении поля, периодически проверяйте значения переменных `selectionBeginIndex` и `selectionEndIndex`.

Чтобы заменить текущее выделение новым текстом, что может потребоваться в приложении с возможностью редактирования слов в стиле текстового процессора, используйте метод экземпляра `replaceSelectedText( )` класса `TextField`. Стоит отметить, однако, что метод `replaceSelectedText( )` работает только в тех случаях, когда текстовое поле имеет фокус или когда переменной `alwaysShowSelection` присвоено значение `true`. Метод `replaceSelectedText( )` представляет собой удобную версию метода `replaceText( )`, который был рассмотрен ранее в разд. «Изменение содержимого текстового поля». Метод `replaceSelectedText( )` работает точно так же, как и метод `replaceText( )`, за исключением того, что он автоматически устанавливает значения параметров *индексНачала* и *индексКонца* в соответствии с текущим выделением.

Гипертекстовые ссылки

Чтобы добавить гипертекстовую ссылку в поле, мы используем переменную экземпляра `url` класса `TextFormat` или тег `<A>` языка HTML. Обычно гипертекстовые ссылки используются для открытия конкретных ресурсов, находящихся

по указываемым URL-адресам. Например, следующий код создает текстовое поле, содержащее гипертекстовую ссылку, при активизации которой приложение Flash Player открывает сайт издательства O'Reilly в браузере, используемом в операционной системе по умолчанию.

```
var t:TextField = new TextField( );
t.htmlText = "To visit O'Reilly's web site, "
             + "<a href='http://www.oreilly.com'>click here</a>";
t.autoSize = TextFieldAutoSize.LEFT;
```

Кроме того, гипертекстовые ссылки могут использоваться для выполнения кода на языке ActionScript. Подробную информацию можно найти в подразд. «Событие TextEvent.LINK» разд. «События текстового ввода» гл. 22.

Мы почти рассмотрели вопросы, касающиеся текстовых полей. Однако перед тем, как перейти к следующей главе, кратко познакомимся с представлением текстовых полей, создаваемых вручную в среде разработки Flash, в языке ActionScript.

Текстовые поля и среда разработки Flash

В среде разработки Flash текстовые поля могут создаваться вручную с помощью инструмента **Text** (Текст). На этапе разработки каждое создаваемое вручную поле может иметь один из трех видов: статический, динамический или вводимый текст. На этапе выполнения каждое создаваемое вручную текстовое поле в коде на языке ActionScript представляется объектом, который соответствует типу поля, указываемому на этапе разработки.

Текстовые поля вида «статический текст» представляются экземплярами класса `StaticText`. Поля вида «динамический текст» представляются экземплярами класса `TextField`, переменной `type` которых присвоено значение `TextFieldType.DYNAMIC`. Текстовые поля вида «вводимый текст» представляются экземплярами класса `TextField`, переменной `type` которых присвоено значение `TextFieldType.INPUT`.

Текстовое содержимое текстовых полей вида «статический текст» может быть прочитано на этапе выполнения из кода на языке ActionScript, однако оно не может быть изменено. В отличие от этого, текстовое содержимое текстовых полей вида «динамический текст» или «вводимый текст» может быть не только прочитано, но и изменено. Таким образом, авторы, применяющие среду разработки Flash, должны применять поля вида «статический текст» в том случае, когда содержимое текстового поля не должно изменяться на этапе выполнения. Для создания текстовых полей, содержимое которых может быть изменено на этапе выполнения, авторы, использующие среду разработки Flash, должны применять виды «динамический текст» или «вводимый текст».

Чтобы получить доступ ко всему тексту во всех статических текстовых полях некоторого экземпляра класса `DisplayObjectContainer`, используйте класс `TextSnapshot` (основное назначение которого заключается в предоставлении возможности выделения текста в нескольких отдельных объектах `StaticText`).

Текстовые поля вида «статический текст» не могут создаваться с помощью кода на языке ActionScript; классы StaticText и TextSnapshot существуют исключительно для предоставления программного доступа к этим полям, создаваемым в среде разработки Flash.

Точно так же, как программисты на языке ActionScript могут устанавливать параметры отображения текста на этапе выполнения, авторы, использующие среду разработки Flash, могут использовать палитру Properties (Свойства) для выбора режима отображения текстовых полей на этапе компиляции. Параметры отображения, предоставляемые средой разработки Flash (и их эквиваленты в языке ActionScript), перечислены в табл. 27.10.

Таблица 27.10. Параметры отображения текста, предоставляемые средой разработки Flash

Значение на палитре Properties (Свойства)	Описание	Эквивалент в языке ActionScript
Use device fonts (Использовать шрифты устройства)	Полагаемся на локальную среду воспроизведения для отображения текста с помощью шрифтов, установленных в системе конечного пользователя	Присвоить переменной embedFonts значение false
Bitmap text (no anti-alias) (Растровый текст (без сглаживания))	Когда выбран вид Bitmap text (Растровый текст), компилятор выравнивает фигуры по целым пикселам при вычислении контуров глифов (поэтому шрифт не выглядит сглаженным). На этапе выполнения эти контуры глифов отображаются с помощью встроенного визуализатора векторной графики приложения Flash Player, а не визуализатора FlashType	Встроить шрифт, установив флажок Bitmap text (Растровый текст) в среде разработки Flash, а затем присвоить переменной embedFonts значение false. Недоступно, если компиляция выполняется с помощью приложения Flex Builder 2 или компилятора mxmlc
Anti-alias for animation (Сглаживание для анимации)	Отображает текст с помощью стандартного визуализатора векторной графики приложения Flash Player	Присвоить переменной antiAliasType значение AntiAliasType.NORMAL
Anti-alias for readability (Сглаживание для читабельности)	Выводит текст с помощью визуализатора FlashType, используя параметры по умолчанию	Присвоить переменной antiAliasType значение AntiAliasType.ADVANCED
Custom anti-alias (Настраиваемое сглаживание)	Отображает текст с помощью визуализатора FlashType, используя настраиваемые параметры	Присвоить переменной antiAliasType значение AntiAliasType.ADVANCED и определить настраиваемые параметры, используя методики, которые были рассмотрены ранее в разд. «Отображение текста с помощью встраиваемых шрифтов»

Загрузка... Пожалуйста, подождите...

На протяжении последних восьми глав мы рассмотрели множество вопросов, касающихся создания и управления визуальным содержимым с помощью API отображения. В следующей главе мы завершим изучение экранного программирования, познакомившись с инструментами языка ActionScript для загрузки внешних отображаемых элементов.

Загрузка внешних отображаемых элементов

В ActionScript существует три способа добавления внешнего отображаемого элемента в приложение программным путем:

- использование класса `flash.display.Loader` для загрузки элемента на этапе выполнения;
- применение класса `flash.net.Socket` совместно с методом экземпляра `loadBytes( )` класса `Loader` для загрузки элемента на этапе выполнения через открытый сокет TCP/IP;
- использование тега метаданных `[Embed]` для встраивания элемента, находящегося в локальной файловой системе, на этапе компиляции.

Классы `Loader` и `Socket` являются собственными классами API среды выполнения Flash, в то время как для использования тега метаданных `[Embed]` требуется платформа разработки Flex. Все три подхода поддерживают следующие форматы отображаемых элементов:

- SWF (скомпилированные приложения Flash);
- JPEG, GIF или PNG (растровые изображения).

Кроме того, тег метаданных `[Embed]` поддерживает отображаемые элементы в формате SVG.

Класс Loader заменяет следующие инструменты языка ActionScript 2.0, предназначенные для загрузки элементов: класс MovieClipLoader; глобальные функции loadMovie( ) и loadMovieNum( ); методы экземпляра loadMovie( ) и loadMovieNum( ) класса MovieClip.

В этой главе будет рассказано, как использовать классы `Loader` и `Socket`, а также тег метаданных `[Embed]` для загрузки внешних отображаемых элементов. Подробную информацию о загрузке шрифтов вы сможете найти в гл. 27. Сведения о загрузке содержимого XML можно найти в гл. 18. Кроме того, информацию о загрузке других неотображаемых элементов, например переменных, бинарных данных или звуковых файлов, можно найти в описании классов `URLLoader`, `URLStream` и `Sound` в справочнике по языку ActionScript корпорации Adobe.

Использование класса Loader для загрузки отображаемых элементов на этапе выполнения

Класс `Loader` применяется для загрузки внешних отображаемых элементов на этапе выполнения. Элемент может быть получен либо по протоколу HTTP, либо из локальной файловой системы. Чтобы воспользоваться классом `Loader`, необходимо выполнить три основных шага.

1. Создать экземпляр класса `flash.display.Loader`.
2. Создать экземпляр класса `flash.net.URLRequest`, определяющий местоположение элемента.
3. Передать экземпляр класса `URLRequest` в метод `load( )` класса `Loader`.

В следующих разделах мы создадим пример класса `SunsetViewer`, в котором подробно рассматриваются описанные шаги. Класс из нашего примера будет загружать единственное растровое изображение `sunset.jpg`. Освоив базовые положения загрузки элементов, вы узнаете, как отслеживать ход выполнения операции загрузки с помощью класса `flash.display.LoaderInfo`. Наконец, мы рассмотрим код, необходимый для обращения к загруженному элементу и его добавления в список отображения.

Операции загрузки подчиняются ограничениям безопасности приложения Flash Player. Полную информацию можно получить в гл. 19.

В классе `SunsetViewer` нашего примера мы предполагаем, что SWF-файл приложения `SunsetViewer.swf` будет размещен на сайте в той же директории, где находится загружаемое изображение `sunset.jpg`.

Создание экземпляра класса Loader

Как мы уже знаем, первым шагом в загрузке любого отображаемого элемента на этапе выполнения с помощью класса `Loader` является создание экземпляра класса `Loader`. Этот экземпляр управляет операцией загрузки и предоставляет доступ к загруженному элементу. Мы создадим наш экземпляр класса `Loader` в методе-конструкторе класса `SunsetViewer` и присвоим его переменной экземпляра `loader`, как показано в листинге 28.1.

Листинг 28.1. Создание экземпляра класса Loader

```
Package {
  import flash.display.*;

  public class SunsetViewer extends Sprite {
    private var loader:Loader;

    public function SunsetViewer ( ) {
      loader = new Loader( );  // Создаем экземпляр класса Loader
    }
  }
```

Определение местоположения элемента

Чтобы загрузить внешний отображаемый элемент с помощью экземпляра класса `Loader`, мы должны указать местоположение этого элемента с помощью объекта `flash.net.URLRequest`. Каждый отдельный объект `URLRequest` описывает местоположение одного внешнего ресурса, находящегося либо в сети, либо в локальной файловой системе. Чтобы создать объект `URLRequest`, который определяет местоположение элемента, используйте следующий обобщенный код, присваивающий местоположение элемента переменной экземпляра `url`:

```
var urlRequest:URLRequest = new URLRequest( );
urlRequest.url = "адресURLЭлемента";
```

Кроме того, местоположение элемента может быть передано в качестве параметра конструктору класса `URLRequest`, как показано в следующем коде:

```
var url:URLRequest = new URLRequest("адресURLЭлемента");
```

В обоих случаях *адресURLЭлемента* — это строка, содержащая стандартный адрес URL. Например:

```
new URLRequest("http://www.example.com/image.jpg");
```

Набор сетевых протоколов, которые допускается использовать в строке *адресURLЭлемента*, зависит от операционной системы. Например, протоколы `http://`, `https://` и `ftp://` поддерживаются всеми операционными системами Windows, Macintosh и UNIX, однако обращение к содержимому справки Windows (`ms-its:`) может поддерживаться только в операционной системе Windows. Из соображений безопасности приложение Flash Player может также блокировать некоторые протоколы. Тем не менее в настоящее время корпорация Adobe не публикует список блокируемых протоколов. Более того, среда выполнения Flash не генерирует никаких сообщений об ошибках безопасности, относящихся конкретно к блокированию протокола. По этой причине при работе с редко используемыми протоколами не забывайте, что операции загрузки с применением таких протоколов могут потерпеть неудачу, не вызвав никаких ошибок.

Помимо определения адреса URL, каждый объект `URLRequest` может также предоставлять дополнительную информацию для запрашиваемого по HTTP-протоколу ресурса. Чтобы указать заголовок HTTP-запроса, используемый метод, данные для метода POST, строку запроса и тип содержимого MIME, просто установите соответствующие переменные объекта `URLRequest`, как описано в справочнике по языку ActionScript корпорации Adobe. Например, чтобы определить заголовки HTTP-запроса, установите переменную экземпляра `requestHeaders` класса `URLRequest`.

Местоположение элемента может быть указано в виде абсолютного или относительного URL-адреса. Однако стоит отметить, что система приложения Flash Player, отвечающая за разрешение относительных адресов URL, зависит от способа запуска приложения Flash Player.

- Если приложение Flash Player запускается с целью отображения SWF-файла, встроенного в веб-страницу с помощью тега `<OBJECT>` или `<EMBED>`, все от-

носительные адреса URL разрешаются относительно этой веб-страницы, а не относительно какого-либо SWF-файла. Более того, если веб-страница была открыта локально, относительные адреса URL разрешаются локально. Если веб-страница была открыта с сайта, относительные адреса URL разрешаются с использованием адреса этого сайта.

- Когда приложение Flash Player запускается как автономное приложение или в результате непосредственного открытия SWF-файла в браузере, поддерживающем формат Flash, все относительные адреса URL разрешаются относительно *первого* SWF-файла, открытого в приложении Flash Player, — этот файл называется *владельцем сцены*. Более того, если владелец сцены был открыт локально, относительные адреса URL разрешаются локально; если владелец сцены был открыт с сайта, относительные адреса URL разрешаются с использованием адреса этого сайта.

Даже если первый SWF-файл, открытый в приложении Flash Player, будет удален со сцены, он останется владельцем сцены и по-прежнему будет оказывать влияние на разрешение относительных URL-адресов.

Рассмотрим пример с использованием относительных адресов URL, который демонстрирует две описанные системы разрешения. Предположим, что мы встроили приложение `SunsetViewer.swf` в веб-страницу `SunsetViewer.html` и сохранили эти два файла в следующих отдельных директориях:

```
/viewer/SunsetViewer.html
/viewer/assets/SunsetViewer.swf
```

Предположим также, что из приложения `SunsetViewer.swf` мы хотим загрузить изображение `sunset.jpg`, которое тоже находится в директории `/assets/`:

```
/viewer/assets/sunset.jpg
```

Если мы считаем, что пользователь будет запускать приложение `SunsetViewer.swf`, открывая веб-страницу `SunsetViewer.html`, мы должны сформировать наш относительный URL-адрес относительно данной веб-страницы, как показано в следующей строке кода:

```
new URLRequest("assets/sunset.jpg");
```

Тем не менее, если мы считаем, что пользователь будет запускать приложение, непосредственно открывая файл `SunsetViewer.swf`, мы должны сформировать наш относительный URL-адрес относительно SWF-файла, а не веб-страницы, как показано в следующей строке кода:

```
new URLRequest("sunset.jpg");
```

При распространении содержимого, воспроизводимого в приложении Flash Player, формируйте все относительные адреса URL в соответствии с тем, как ваши пользователи будут запускать приложение Flash Player.

Даже если один SWF-файл загружает другой SWF-файл, который, в свою очередь, загружает внешние элементы, относительные адреса URL все равно разрешаются

либо относительно владельца сцены (в случае непосредственного запуска), либо относительно веб-страницы, содержащей встроенное приложение Flash Player (в случае открытия страницы).

Предположим, что мы открываем гипотетическое приложение `SlideShow.swf` непосредственно в приложении Flash Player. Кроме того, предположим, что приложение `SlideShow.swf` загружает приложение `SunsetViewer.swf` из предыдущего примера. В данном случае все относительные URL-адреса в приложении `SunsetViewer.swf` должны быть сформированы относительно файла `SlideShow.swf` (обратите внимание: *не* относительно приложения `SunsetViewer.swf`!). Подобным образом, если бы приложение `SlideShow.swf` открывалось через веб-страницу, все относительные адреса URL в приложении `SunsetViewer.swf` должны были быть сформированы относительно данной страницы.

Вы можете полностью избежать проблем, связанных с различными вариантами разрешения относительных адресов URL, сохранив все HTML-, SWF-файлы и файлы внешних отображаемых элементов в одной директории.

Вернемся к нашему примеру класса `SunsetViewer`. Ранее в листинге 28.1 мы создали экземпляр класса `Loader`. В листинге 28.2 представлен обновленный класс `SunsetViewer`, в который был добавлен код, создающий запрос для относительного URL-адреса `"sunset.jpg"`. Как было отмечено ранее, для упрощения нашего примера мы предполагаем, что файлы `sunset.jpg` и `SunsetViewer.swf` находятся в одной директории — это позволяет избежать любых сложностей, связанных с разрешением относительных адресов URL.

Листинг 28.2. Определение местоположения элемента

```
package {
  import flash.display.*;
  import flash.net.URLRequest;  // Импортируем класс URLRequest

  public class SunsetViewer extends Sprite {
    private var loader:Loader;

    public function SunsetViewer ( ) {
      loader = new Loader( );
      // Определяем местоположение элемента в виде "sunset.jpg"
      var urlRequest:URLRequest = new URLRequest("sunset.jpg");
    }
  }
}
```

Теперь, когда мы определили местоположение файла `sunset.jpg`, загрузим его.

Начало загрузки

До настоящего момента мы уже создали объекты `Loader` и `URLRequest`. Теперь соединим их, чтобы загрузить элемент. Чтобы начать загрузку, мы передаем наш экземпляр класса `URLRequest` в метод `load( )` экземпляра класса `Loader`, как показано в листинге 28.3.

Листинг 28.3. Начало загрузки

```
package {
  import flash.display.*;
  import flash.net.URLRequest;

  public class SunsetViewer extends Sprite {
    private var loader:Loader;

    public function SunsetViewer ( ) {
      loader = new Loader( );
      var url:URLRequest = new URLRequest("sunset.jpg");
      // Начинаем загрузку
      loader.load(url);
    }
  }
}
```

Итак, вот основной код, необходимый для загрузки внешнего отображаемого элемента на этапе выполнения:

```
var loader:Loader = new Loader( );
var url:URLRequest = new URLRequest("адресURLЭлемента");
loader.load(url);
```

Здесь *адресURLЭлемента* — это местоположение загружаемого элемента. Кроме того, допустимо и достаточно распространено объединять вторую и третью строки из предыдущего кода в одну, как показано в следующем коде:

```
var loader:Loader = new Loader( );
loader.load(new URLRequest("адресURLэлемента"));
```

Чтобы отменить выполняемую операцию загрузки, используйте метод экземпляра close( ) класса Loader.

После начала загрузки элемента мы в конечном итоге захотим обратиться к этому элементу и отобразить его на экране. Эти вопросы рассматриваются в двух следующих разделах.

Обращение к загруженному элементу

Благополучно обратиться к загруженному элементу можно только после того, как среда выполнения Flash проведет его инициализацию. На этапе инициализации среда выполнения Flash создает экземпляр элемента, добавляет его в объект `Loader`, загрузивший этот элемент, и выполняет любые задачи, необходимые для подготовки элемента к использованию.

Этап процесса инициализации, на котором создается экземпляр, отличается для разных типов элементов.

- Для растровых изображений экземпляр создается в тот момент, когда загрузка внешнего файла полностью завершена. Тогда загруженные пиксельные данные автоматически помещаются в объект `BitmapData`, который связывается

с новым объектом Bitmap. Загруженное изображение представляется объектом Bitmap.

- Для SWF-файлов экземпляр создается в тот момент, когда получены все элементы и классы для кадра 1 (включая основной класс SWF-файла). Тогда среда выполнения Flash создает экземпляр основного класса SWF-файла и выполняет его конструктор. Загруженный SWF-файл представляется экземпляром основного класса.

Для простоты в дальнейшем материале мы будем называть созданный экземпляр (либо объект Bitmap, либо экземпляр основного класса SWF-файла) объектом элемента.

После создания экземпляра загруженного элемента объект элемента автоматически добавляется в объект Loader. Объект элемента является первым и единственным возможным ребенком объекта Loader. Если элементом оказывается SWF-файл, то любой код, находящийся в первом кадре этого файла, выполняется сразу после добавления экземпляра его основного класса в объект Loader.

После того как объект элемента будет добавлен в объект Loader и процесс инициализации будет завершен, среда Flash выполнит диспетчеризацию события Event.INIT. Когда возникает событие Event.INIT, загруженный элемент считается готовым к использованию. Любой код, которому необходим доступ к загруженному элементу, должен выполняться только после возникновения события Event.INIT.

Не пытайтесь обращаться к загружаемому элементу до возникновения события Event.INIT.

Приемники, которые желают получать уведомления о возникновении события Event.INIT, должны быть зарегистрированы в объекте LoaderInfo объекта элемента, а не в объекте Loader, над которым изначально был вызван метод load( ). Объект LoaderInfo — это отдельный объект, который представляет информацию о загруженном элементе. Любой экземпляр класса Loader предоставляет ссылку на объект LoaderInfo своего загружаемого элемента через переменную экземпляра contentLoaderInfo. Таким образом, чтобы зарегистрировать приемник события для события Event.INIT некоторого элемента, мы используем следующий обобщенный код:

```
объектLoader.contentLoaderInfo.addEventListener(Event.INIT, приемникСобытияINIT);
```

Здесь *объектLoader* — это объект Loader, загружающий элемент, а *приемникСобытияINIT* — ссылка на функцию, которая будет обрабатывать событие Event.INIT. После возникновения события Event.INIT благополучно обратиться к загруженному элементу можно через переменную content или метод getChildAt( ) класса Loader, как показано ниже:

```
объектLoader.content
объектLoader.getChildAt(0)
```

Обратите внимание на значение 0 в выражении getChildAt(0). Элемент является единственным ребенком объекта Loader, поэтому он находится на глубине с индексом 0.

Следующий код демонстрирует приемник события `Event.INIT`, который устанавливает позицию загруженного и проинициализированного элемента. Для демонстрации двух различных способов обращения к загруженному элементу этот код устанавливает горизонтальную позицию с помощью переменной `content`, а вертикальную позицию — с использованием выражения `getChildAt(0)`.

```
private function initListener (e:Event):void {
  объектLoader.content.x = 50;
  объектLoader.getChildAt(0).y = 75;
}
```

В качестве альтернативы обратиться к загруженному элементу можно через объект `Event`, передаваемый в функцию-приемник события `Event.INIT`. Этот объект определяет переменную `target`, которая ссылается на объект `LoaderInfo` элемента. Каждый объект `LoaderInfo` ссылается на свой соответствующий элемент через переменную экземпляра `content`. Таким образом, внутри функции-приемника события `Event.INIT` ссылка на загруженный элемент может быть получена с помощью выражения *объектEvent*`.target.content`. Например, следующий код устанавливает горизонтальной позиции загруженного элемента значение 100:

```
private function initListener (e:Event):void {
  e.target.content.x = 100;
}
```

При обращении к загруженным элементам следует помнить, что, поскольку элемент не добавляется в свой объект `Loader` до тех пор, пока процесс загрузки данного элемента не будет успешно завершен, вызов метода *объектLoader*`.getChildAt(0)` до начала операции загрузки приведет к ошибке.

Напомним, что элемент растрового изображения не добавляется в свой объект Loader до тех пор, пока не будет полностью загружен внешний файл. Элемент SWF-файла не добавляется в свой объект Loader до тех пор, пока не будут получены все элементы и классы для первого кадра, включая основной класс SWF-файла.

Более того, перед началом загрузки переменная `content` содержит значение `null`. Это демонстрирует следующий код:

```
// Начинаем загрузку
var loader:Loader = new Loader( );
loader.load(new URLRequest("sunset.jpg"));

// Сразу же пытаемся обратиться к загружаемому элементу,
// не дожидаясь завершения загрузки
trace(loader.getChildAt(0));  // RangeError: Ошибка #2006:
                              // The supplied index is out of bounds.
                              // (Указанный индекс выходит за пределы.)
trace(loader.content);        // Выводит: null
```

Даже если операция загрузки началась, но элемент не был успешно загружен, обращение к переменной `content` или вызов метода `getChildAt(0)` приведет к следующей ошибке:

```
Error: Error #2099: The loading object is not sufficiently loaded
                    to provide this information.
```

На русском языке она будет выглядеть так: Загружаемый объект не был успешно загружен для предоставления данной информации.

Снова повторим: не пытайтесь обращаться к загружаемому элементу до момента возникновения события `Event.INIT`. На самом деле обратное также верно: перед тем как обратиться к родительскому объекту `Loader`, код в основном классе загруженного SWF-файла должен дождаться возникновения события `Event.INIT`. В частности, код в конструкторе основного класса загруженного SWF-файла не имеет доступа к родительскому объекту `Loader`. С другой стороны, код, помещенный в первый кадр временной шкалы загруженного SWF-файла (в среде разработки Flash), может обращаться к родительскому объекту `Loader` (до возникновения события `Event.INIT`).

Воспользуемся новыми знаниями по обращению к элементам на примере класса `SunsetViewer`. В листинге 28.4 показано, как обращаться к загруженному файлу `sunset.jpg` из функции-приемника события `Event.INIT`. Эта функция отображает высоту, ширину и угол поворота загруженного изображения с помощью методик обращения к элементу, рассмотренных в данном разделе.

Листинг 28.4. Обращение к загруженному элементу

```
package {
  import flash.display.*;
  import flash.net.URLRequest;
  import flash.events.*;

  public class SunsetViewer extends Sprite {
    private var loader:Loader;

    public function SunsetViewer ( ) {
      loader = new Loader( );

      // Регистрируем приемник для события Event.INIT
      loader.contentLoaderInfo.addEventListener(Event.INIT, initListener);
      var urlRequest:URLRequest = new URLRequest("sunset.jpg");
      loader.load(urlRequest);
    }

    // Приемник вызывается при возникновении события Event.INIT
    private function initListener (e:Event):void {
      // Обращаемся к элементу тремя различными способами
      trace(loader.content.width);
      trace(loader.getChildAt(0).height);
      trace(e.target.content.rotation);
    }
  }
}
```

Чтобы для загруженного элемента использовать методы и переменные класса Bitmap, MovieClip или основного класса SWF-файла, следуйте методикам, описанным далее в разд. «Проверка типов на этапе компиляции для динамически загружаемых элементов».

Отображение загруженного элемента на экране

Как только элемент будет готов к использованию, он также будет готов к добавлению в список отображения для дальнейшего вывода на экран. Чтобы добавить загруженный элемент в список отображения, мы используем метод экземпляра `addChild( )` класса `DisplayObjectContainer`, как и при добавлении любого другого отображаемого объекта в список отображения. В листинге 28.5 представлен код, который добавляет объект `Bitmap`, представляющий файл `sunset.jpg`, в основной класс приложения `SunsetViewer`.

Листинг 28.5. Добавление элемента в список отображения

```
package {
  import flash.display.*;
  import flash.net.URLRequest;
  import flash.events.*;

  public class SunsetViewer extends Sprite {
    private var loader:Loader;

    public function SunsetViewer ( ) {
      loader = new Loader( );
      loader.contentLoaderInfo.addEventListener(Event.INIT, initListener);
      var urlRequest:URLRequest = new URLRequest("sunset.jpg");
      loader.load(urlRequest);
    }

    private function initListener (e:Event):void {
      addChild(loader.content);  // Добавляем объект Bitmap
                                 // в объект SunsetViewer
    }
  }
}
```

Как показано в листинге 28.5, добавление объекта загруженного элемента в новый объект `DisplayObjectContainer` автоматически приводит к удалению этого элемента из его исходного родительского объекта `Loader`. В качестве примера добавим новый код в метод `initListener( )` из листинга 28.5. Этот код проверяет, сколько дочерних объектов имеет объект `loader` до и после добавления загруженного элемента (`sunset.jpg`) в объект `SunsetViewer`. Обратите внимание, что после перемещения элемента объект `loader` не имеет дочерних отображаемых объектов.

```
private function initListener (e:Event):void {
  trace(loader.numChildren);  // Выводит: 1 (единственным ребенком
                              // является элемент)
  addChild(loader.content);
  trace(loader.numChildren);  // Выводит: 0 (поскольку элемент был
                              // перемещен)
}
```

Альтернативная методика вывода загруженного элемента на экран заключается в добавлении в список отображения объекта `Loader` этого элемента, а не объекта элемента.

Класс Loader сам по себе является потомком класса DisplayObject, поэтому он непосредственно может быть добавлен в любой объект DisplayObjectContainer. Снова модифицируем метод initListener( ) из листинга 28.5. На этот раз мы добавим объект loader непосредственно в объект SunsetViewer. В результате этой операции мы неявно делаем объект Bitmap, представляющий файл sunset.jpg, правнуком объекта SunsetViewer.

```
private function initListener (e:Event):void {
  addChild(loader);  // Добавляем объект loader и его дочерний элемент
                     // в список отображения
}
```

Объект Loader фактически может быть добавлен в список отображения до начала операции загрузки. Когда операция загрузки отображаемого элемента будет завершена, этот элемент автоматически добавится в объект Loader и, как следствие, в список отображения. Данная методика продемонстрирована в листинге 28.6. Код из этого листинга добавляет объект loader в список отображения до начала операции загрузки файла sunset.jpg. После того как экземпляр файла sunset.jpg будет создан и проинициализирован, он будет добавлен в объект loader и, поскольку объект loader уже находится в списке отображения, появится на экране. Таким образом, нет надобности в приемнике события Event.INIT.

Листинг 28.6. Добавление объекта Loader в список отображения

```
package {
  import flash.display.*;
  import flash.net.URLRequest;

  public class SunsetViewer extends Sprite {
    private var loader:Loader;

    public function SunsetViewer ( ) {
      loader = new Loader( );
      addChild(loader);
      var urlRequest:URLRequest = new URLRequest("sunset.jpg");
      loader.load(urlRequest);
    }
  }
}
```

Следовательно, самый простой из возможных способов загрузить отображаемый элемент и отобразить его на экране можно представить следующим кодом:

```
var loader:Loader = new Loader( );
addChild(loader);
loader.load(new URLRequest(адресURLЭлемента));
```

Три предыдущие строки кода подходят для многих простых ситуаций, однако в тех случаях, когда управление загруженным элементом должно осуществляться независимо от его объекта Loader или когда длительность паузы перед отображением загруженного элемента должна определяться вручную, более подходящим является код, который был представлен ранее в листинге 28.5.

Мы уже знаем, как загружать и отображать внешний элемент, но зачастую в процессе загрузки элемента между этими двумя операциями существует заметная задержка. Далее мы рассмотрим, как использовать класс `LoaderInfo` для отображения хода выполнения операции загрузки.

Отображение хода загрузки

Чтобы отобразить ход загрузки элемента, мы должны выполнить четыре следующих обобщенных шага.

1. Перед загрузкой элемента создать графический индикатор хода загрузки (например, текстовое поле или «индикатор загрузки»).
2. После начала загрузки добавить индикатор хода выполнения в список отображения.
3. По мере загрузки элемента обновлять состояние индикатора хода выполнения (по большому счету, это делается для удобства пользователя).
4. Когда загрузка будет завершена, удалить индикатор хода выполнения с экрана.

Посмотрим, как применить описанные шаги на практике, добавив простой индикатор хода выполнения, который реализован на базе текстового поля, в наш класс `SunsetViewer`.

Начнем с создания в классе `SunsetViewer` новой переменной экземпляра `progressOutput`, которая ссылается на обычный объект `TextField`. Текстовое поле `progressOutput` будет отображать информацию о ходе загрузки.

```
private var progressOutput:TextField;
```

Далее мы создадим в классе `SunsetViewer` два новых метода: `createProgressIndicator( )` и `load( )`. Первый метод создает объект `TextField progressOutput`. Мы будем вызывать метод `createProgressIndicator( )` из конструктора класса `SunsetViewer`. Рассмотрим этот код:

```
private function createProgressIndicator ( ):void {
  progressOutput  = new TextField( );
  progressOutput.autoSize   = TextFieldAutoSize.LEFT;
  progressOutput.border     = true;
  progressOutput.background = true;
  progressOutput.selectable = false;
  progressOutput.text       = "LOADING...";
}
```

Второй метод добавляет объект `progressOutput` в список отображения и начинает загрузку элемента. Всякий раз, когда с помощью метода `load( )` инициируется загрузка, объект `progressOutput` помещается на экран; всякий раз, когда загрузка завершается, объект `progressOutput` удаляется с экрана. Данная архитектура позволяет классу `SunsetViewer` повторно использовать один и тот же объект `TextField` для отображения информации о ходе загрузки. Рассмотрим код для метода `load( )`:

```
private function load (urlRequest:URLRequest):void {
  // Начинаем загрузку
  loader.load(urlRequest);
```

```
  // Если объект progressOutput еще не является потомком данного объекта...
  if (!contains(progressOutput)) {
    // ...добавляем его
    addChild(progressOutput);
  }
}
```

В процессе загрузки мы ожидаем возникновения события `ProgressEvent.PROGRESS`, которое свидетельствует о появлении новой порции данных файла `sunset.jpg` и предоставляет самую последнюю информацию о ходе загрузки. Всякий раз, когда возникает событие `ProgressEvent.PROGRESS`, мы обновляем объект `progressOutput`. Получателем данного события является объект `LoaderInfo` нашего загружаемого элемента. Как мы уже знаем, обратиться к объекту `LoaderInfo` элемента можно через переменную экземпляра `contentLoaderInfo` класса `Loader`. Таким образом, чтобы зарегистрировать приемник для получения уведомлений о возникновении события `ProgressEvent.PROGRESS`, мы используем следующий код:

```
loader.contentLoaderInfo.addEventListener(ProgressEvent.PROGRESS,
                                          progressListener);
```

В приведенном коде `progressListener` — это ссылка на функцию, которую мы хотим выполнять при возникновении события `ProgressEvent.PROGRESS`. В функцию `progressListener` передается объект `ProgressEvent`, переменные которого содержат следующую информацию:

- размер файла загружаемого элемента (`bytesTotal`);
- количество байтов, полученных до настоящего момента (`bytesLoaded`).

Следующий код демонстрирует функцию `progressListener` для нашего класса `SunsetViewer`. Обратите внимание, как эта функция получает информацию о ходе выполнения загрузки из объекта `ProgressEvent e`:

```
private function progressListener (e:ProgressEvent):void {
  // Обновляем индикатор хода выполнения. 1 Кбайт равен 1024 байт,
  // поэтому делим результат на 1024, чтобы преобразовать его в килобайты.
  progressOutput.text = "LOADING: "
                      + Math.floor(e.bytesLoaded / 1024)
                      + "/" + Math.floor(e.bytesTotal / 1024) + " KB";
}
```

Когда элемент будет загружен полностью, среда Flash выполнит диспетчеризацию события `Event.COMPLETE`, получателем которого является объект `LoaderInfo` элемента. При возникновении события `Event.COMPLETE` мы можем удалить индикатор хода выполнения (`progressOutput`) из списка отображения.

Чтобы зарегистрировать приемник для событий `Event.COMPLETE` в объекте `LoaderInfo` нашего загружаемого элемента, мы используем следующий код:

```
loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
                                          completeListener);
```

Здесь `completeListener` — это ссылка на функцию, которую мы хотим выполнять при возникновении события `Event.COMPLETE`. Следующий код демонстрирует функцию `completeListener`. Ее роль заключается в простом удалении объекта `progressOutput` из списка отображения.

```
private function completeListener (e:Event):void {
  // Удаляем индикатор хода выполнения
  removeChild(progressOutput);
}
```

Наконец, чтобы облегчить повторное использование кода и упростить его чтение, мы переместим наш код для создания объекта `Loader` и код, регистрирующий приемники событий, в новый метод `createLoader( )`, представленный ниже. Обратите внимание, что код в этом методе регистрирует приемники не только для событий `ProgressEvent.PROGRESS` и `Event.COMPLETE`, но и для события `Event.INIT`, которое было рассмотрено ранее в подразд. «Обращение к загруженному элементу». Как и раньше, мы помещаем наш загруженный элемент в список отображения при возникновении события `Event.INIT`. Рассмотрим код для метода `createLoader( )`:

```
private function createLoader ( ):void {
  // Создаем объект Loader
  loader = new Loader( );

  // Регистрируем приемники для событий
  loader.contentLoaderInfo.addEventListener(ProgressEvent.PROGRESS,
progressListener);
  loader.contentLoaderInfo.addEventListener(Event.COMPLETE, completeListener);
  loader.contentLoaderInfo.addEventListener(Event.INIT, initListener);
}
```

Код, который удаляет индикатор хода выполнения из списка отображения (в нашем примере он представлен методом `completeListener( )`), должен всегда использовать событие `Event.COMPLETE` вместо `Event.INIT`. Не путайте эти два события. Событие `Event.INIT` выражает качественное состояние «готовности элемента», а событие `Event.COMPLETE` — количественное состояние завершения загрузки. Событие `Event.INIT` обозначает, что элемент готов к использованию, даже если — в случае SWF-файла — загрузка еще продолжается. В отличие от этого, событие `Event.COMPLETE` свидетельствует о получении всех байтов файла, содержащего элемент.

Используйте событие Event.INIT, чтобы определить момент, когда можно благополучно обращаться к элементу. Применяйте событие Event.COMPLETE, чтобы определить момент завершения операции загрузки.

Поскольку некоторые типы элементов могут быть проинициализированы до того, как будут полностью загружены, событие `Event.INIT` всегда возникает перед событием `Event.COMPLETE`. Предположим, что мы загружаем SWF-файл, представляющий анимацию из 2000 кадров. Когда первый кадр будет загружен и проинициализирован, возникнет событие `Event.INIT`. В этот момент мы добавляем анимацию в список отображения и разрешаем начать ее воспроизведение, хотя загрузка SWF-файла все еще продолжается. По мере загрузки SWF-файла индикатор будет представлять ход выполнения данной операции. Когда загрузка SWF-файла будет завершена, возникнет событие `Event.COMPLETE` и мы удалим индикатор загрузки с экрана.

В листинге 28.7 снова представлен наш класс `SunsetViewer`, но на этот раз он включает код для отображения индикатора хода загрузки, рассмотренный в текущем разделе.

Листинг 28.7. Отображение хода загрузки

```
package {
  import flash.display.*;
  import flash.net.URLRequest;
  import flash.events.*
  import flash.text.*;

  public class SunsetViewer extends Sprite {
    private var loader:Loader;              // Загрузчик элемента
    private var progressOutput:TextField;   // Поле, в котором будет
                                            // отображаться ход загрузки

    // Конструктор
    public function SunsetViewer ( ) {
      // Создаем объект Loader и регистрируем приемники событий
      createLoader( );

      // Создаем индикатор хода выполнения
      createProgressIndicator( );

      // Начинаем загрузку
      load(new URLRequest("sunset.jpg"));
    }

    private function createLoader ( ):void {
      // Создаем объект Loader
      loader = new Loader( );

      // Регистрируем приемники для событий
      loader.contentLoaderInfo.addEventListener(ProgressEvent.PROGRESS,
progressListener);
      loader.contentLoaderInfo.addEventListener(Event.COMPLETE, completeListener);
      loader.contentLoaderInfo.addEventListener(Event.INIT, initListener);
    }

    private function createProgressIndicator ( ):void {
      progressOutput = new TextField( );
      progressOutput.autoSize   = TextFieldAutoSize.LEFT;
      progressOutput.border     = true;
      progressOutput.background = true;
      progressOutput.selectable = false;
      progressOutput.text       = "LOADING...";
    }

    private function load (urlRequest:URLRequest):void {
      loader.load(urlRequest);
      if (!contains(progressOutput)) {
```

```
        addChild(progressOutput);
      }
    }

    // Приемник вызывается, когда появляются данные
    private function progressListener (e:ProgressEvent):void {
      // Обновляем индикатор хода выполнения.
      progressOutput.text = "LOADING: "
                          + Math.floor(e.bytesLoaded / 1024)
                          + "/" + Math.floor(e.bytesTotal / 1024) + " KB";
    }

    private function initListener (e:Event):void {
      addChild(loader.content);  // Добавляем загруженный элемент в список
                                 // отображения
    }

    // Приемник вызывается, когда загрузка элемента полностью завершена
    private function completeListener (e:Event):void {
      // Удаляем индикатор хода выполнения.
      removeChild(progressOutput);
    }
  }
}
```

Почему бы не использовать событие Event.OPEN? Если вы просматривали документацию по API среды выполнения Flash, то, возможно, обратили внимание на кажущееся удобным событие `Event.OPEN`, которое возникает в момент начала загрузки. Теоретически событие `Event.OPEN` предоставляет хорошее место, где можно размещать код для добавления индикатора хода выполнения в список отображения. Как мы уже знаем, код, отображающий ход выполнения, выполняет четыре основные задачи.

1. Создание индикатора хода выполнения.
2. Добавление индикатора хода выполнения в список отображения.
3. Обновление индикатора хода выполнения.
4. Удаление индикатора хода выполнения из списка отображения.

Первая операция обычно выполняется на этапе подготовки. Три оставшиеся операции соответствуют трем событиям загрузки: `Event.OPEN`, `Event.PROGRESS` и `Event.COMPLETE`. Вы можете поинтересоваться, почему же класс `SunsetViewer` из листинга 28.7 добавляет объект `progressOutput` в список отображения в методе `load( )`, а не в приемнике события `Event.OPEN`.

```
private function load (urlRequest:URLRequest):void {
  loader.load(urlRequest);
  if (!contains(progressOutput)) {
    // Почему мы делаем это здесь...
    addChild(progressOutput);
  }
}
```

```
private function openListener (e:Event):void {
  if (!contains(progressOutput)) {
    // ...а не здесь?
    addChild(progressOutput);
  }
}
```

На самом деле приемник события `Event.OPEN` теоретически мог бы быть прекрасным местом для добавления объекта `progressOutput` в список отображения. К сожалению, на практике специфическое поведение браузеров затрудняет использование события `Event.OPEN`, и поэтому в данной книге мы избегаем его применения. Исчерпывающую информацию по этому вопросу можно найти далее, в подразд. «Аспекты поведения при ошибках загрузки, зависящие от среды».

Мы уже знаем, как загружать внешний элемент, отображать его на экране и показывать ход выполнения его загрузки пользователю. Теперь рассмотрим код, необходимый для восстановления работоспособности программы после ошибок загрузки.

Обработка ошибок загрузки

Как мы уже знаем из гл. 19, всякий раз, когда попытка загрузить элемент терпит неудачу из-за ограничений безопасности, приложение Flash Player либо генерирует исключение `SecurityError`, либо выполняет диспетчеризацию события `SecurityErrorEvent.SECURITY_ERROR`. Всякий раз, когда такая попытка терпит неудачу по любой другой причине, среда выполнения Flash осуществляет диспетчеризацию события `IOErrorEvent.IO_ERROR`, получателем которого является объект `LoaderInfo` элемента. Обрабатывая это событие, мы можем попытаться восстановить работоспособность программы после любой ошибки загрузки, не связанной с ограничениями безопасности. Например, в приемнике события `IOErrorEvent.IO_ERROR` мы могли бы написать код, который просит пользователя проверить подключение к Интернету.

Добавим код, обрабатывающий ошибки загрузки, в наш класс `SunsetViewer`. Чтобы зарегистрировать приемник для получения уведомлений о возникновении событий `IOErrorEvent.IO_ERROR`, мы используем уже знакомый код:

```
loader.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
                                          ioErrorListener);
```

Здесь `ioErrorListener` — ссылка на функцию, которая будет обрабатывать данное событие. В следующем коде представлена функция `ioErrorListener`. В нашем приложении `SunsetViewer` функция `ioErrorListener( )` просто отображает сообщение об ошибке пользователю в текстовом поле `progressOutput`.

```
// Приемник вызывается при возникновении ошибки загрузки
private function ioErrorListener (e:IOErrorEvent):void {
  progressOutput.text = "LOAD ERROR";
}
```

В отличие от других событий загрузки, если при диспетчеризации приложением Flash Player события `IOErrorEvent.IO_ERROR`, получателем которого является

объект `LoaderInfo`, никакая функция-приемник не зарегистрирована для его обработки, среда выполнения Flash генерирует ошибку на этапе выполнения. Например:

```
Error #2044: Unhandled IOErrorEvent:. text=Error #2035: URL Not Found.
```

Безусловно, сообщение об ошибке **Unhandled IOErrorEvent** (Необработанное событие IOErrorEvent), как и обо всех ошибках на этапе выполнения, отображается только в отладочной версии приложения Flash Player. В рабочей версии приложения Flash Player среда Flash не выдаст пользователю никаких сообщений об ошибках загрузки. Вместо этого она полагает, что реагировать на ошибки соответствующим образом должен код приложения.

В листинге 28.8 представлена окончательная версия класса `SunsetViewer`, дополненная кодом для обработки ошибок загрузки. Новый код выделен полужирным шрифтом.

Листинг 28.8. Окончательная версия класса SunsetViewer, с обработкой ошибок загрузки

```
package {
  import flash.display.*;
  import flash.net.URLRequest;
  import flash.events.*
  import flash.text.*;

  public class SunsetViewer extends Sprite {
    private var loader:Loader;
    private var progressOutput:TextField;

    public function SunsetViewer ( ) {
      createLoader( );
      createProgressIndicator( );
      load(new URLRequest("sunset.jpg"));
    }

    private function createLoader ( ):void {
      loader = new Loader( );
      loader.contentLoaderInfo.addEventListener(ProgressEvent.PROGRESS,
                                                progressListener);
      loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
                                                completeListener);
      loader.contentLoaderInfo.addEventListener(Event.INIT,
                                                initListener);
      loader.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
                                                ioErrorListener);
    }

    private function createProgressIndicator ( ):void {
      progressOutput = new TextField( );
      progressOutput.autoSize   = TextFieldAutoSize.LEFT;
      progressOutput.border     = true;
      progressOutput.background = true;
      progressOutput.selectable = false;
```

```
      progressOutput.text       = "LOADING...";
    }

    private function load (urlRequest:URLRequest):void {
      loader.load(urlRequest);
      if (!contains(progressOutput)) {
        addChild(progressOutput);
      }
    }

    private function progressListener (e:ProgressEvent):void {
      progressOutput.text = "LOADING: "
                          + Math.floor(e.bytesLoaded / 1024)
                          + "/" + Math.floor(e.bytesTotal / 1024) + " KB";
    }

    private function initListener (e:Event):void {
      addChild(loader.content);
    }

    private function completeListener (e:Event):void {
      removeChild(progressOutput);
    }

    // Приемник вызывается при возникновении ошибки загрузки
    private function ioErrorListener (e:IOErrorEvent):void {
      progressOutput.text = "LOAD ERROR";
    }
  }
}
```

Аспекты поведения при ошибках загрузки, зависящие от среды

При осуществлении операций загрузки среда выполнения Flash полагается на свою локальную среду (то есть на операционную систему или приложение, в котором запускается среда выполнения Flash, — зачастую таким приложением является браузер). Как результат, некоторые аспекты поведения Flash, связанного с загрузкой данных, зависят от среды. Среда выполнения по мере возможности пытается оградить программиста от подобных аспектов поведения. Тем не менее в приложении Flash Player 9 существуют два аспекта поведения, которые представлены в табл. 28.1, — они являются уникальными для приложения Internet Explorer в операционной системе Windows и требуют особого внимания программиста.

Таблица 28.1. Аспекты связанного с загрузкой поведения приложения Internet Explorer

Особенность	Поведение Internet Explorer	Поведение автономного проигрывателя, приложений firefox и Adobe AIR
Event.OPEN	Все операции загрузки генерируют событие Event.OPEN, даже если в дальнейшем они завершатся неудачей по причине «файл не найден»	Событие Event.OPEN не генерируется для операций загрузки, которые завершаются неудачей по причине «файл не найден»

Особенность	Поведение Internet Explorer	Поведение автономного проигрывателя, приложений firefox и Adobe AIR
Переменная экземпляра text класса IOErrorEvent	Когда загрузка завершается неудачей по причине «файл не найден», переменной text присваивается значение «Error #2036: Load Never Completed» (Загрузка не будет завершена)	Когда загрузка завершается неудачей по причине «файл не найден», переменной text присваивается значение Error #2035: URL Not Found (Адрес URL не найден)

Ни одно из описанных в табл. 28.1 поведений не является «правильным». Каждое поведение просто зависит от среды, под управлением которой запущена среда Flash. Тем не менее, поскольку согласованность между поведениями всех сред отсутствует, необходимо проявлять осторожность при написании кода, который использует событие `Event.OPEN` или переменную экземпляра `text` класса `IOErrorEvent`. Чтобы добиться платформенной независимости, выполняйте две следующие рекомендации.

- Не используйте значение переменной экземпляра `text` класса `IOErrorEvent` для принятия решений в логике ветвления. Используйте эту переменную только для отладочных целей.
- Избегайте применения события `Event.OPEN` для любых целей, кроме отладки.

Рассмотрим один пример, который объясняет, почему использование события `Event.OPEN` может вызвать проблемы в приложении. Предположим, что приложение использует пользовательский класс `LoadBar` для отображения хода выполнения операции загрузки. Приложение добавляет экземпляр класса `LoadBar` в список отображения всякий раз, когда начинается операция загрузки, внутри приемника события `Event.OPEN`:

```
private function openListener (e:Event):void {
  addChild(loadBar);
}
```

Теперь предположим, что приложение пытается загрузить файл, который не может быть найден. Если приложение будет запущено в браузере Internet Explorer, метод `openListener( )` выполнится и экземпляр класса `LoadBar` появится на экране. Однако если приложение будет запущено в любом другом браузере, метод `openListener( )` *не* выполнится и экземпляр класса `LoadBar` не появится на экране. В лучшем случае разработчик заметит данное несоответствие и напишет код для браузера Internet Explorer, удаляющий экземпляр класса `LoadBar` с экрана в случае ошибки загрузки. Подобный код усложняет приложение и увеличивает вероятность появления ошибок. В худшем случае разработчик не заметит данное несоответствие и в Internet Explorer экземпляр класса `LoadBar` будет находиться на экране в течение всего жизненного цикла приложения.

Чтобы полностью избежать подобной проблемы, безопаснее всего вообще не использовать событие `Event.OPEN`. Вместо этого просто применяйте методику, которая была рассмотрена в коде из листинга 28.7: добавляйте на экран любые индикаторы хода загрузки вручную перед началом загрузки.

В будущих версиях языка ActionScript аспекты поведения, зависящие от среды, которые были перечислены в табл. 28.1, возможно, будут стандартизованы, и вам больше не придется избегать использования события `Event.OPEN`.

Отладка с использованием класса HTTPStatusEvent

Когда HTTP-клиент запрашивает некий элемент по протоколу HTTP, HTTP-сервер возвращает код статуса, который сообщает о том, как был обработан данный запрос. Например, если HTTP-запрос был обработан успешно, HTTP-сервер вернет код статуса 200. Если обработка HTTP-запроса завершилась неудачно, сервер отправит статус ошибки, описывающий возникшую проблему. Коды статуса HTTP для ошибок загрузки зачастую представляют более подробную информацию, чем общее событие `IOErrorEvent.IO_ERROR` языка ActionScript, поэтому эти коды полезно использовать при отладке. Однако поддержка кодов статуса HTTP реализована не во всех средах.

Версии приложения Flash Player, реализованные в виде модулей расширения браузеров Netscape, Mozilla (Firefox), Safari, Opera и Internet Explorer (версия для операционной системы Macintosh), не поддерживают коды статуса HTTP.

При получении кода статуса HTTP от сервера приложение Flash Player осуществляет диспетчеризацию события `HTTPStatusEvent.HTTP_STATUS`, получателем которого является объект `LoaderInfo` загружаемого элемента. Чтобы зарегистрировать приемник для получения уведомлений о возникновении события `HTTPStatusEvent.HTTP_STATUS`, мы используем следующий код:

```
объектLoader.contentLoaderInfo.addEventListener(HTTPStatusEvent.HTTP_STATUS,
                                                httpStatusListener);
```

Здесь *объектLoader* — объект `Loader`, загружающий элемент, а *httpStatusListener* — ссылка на функцию, которая будет обрабатывать данное событие. В функцию *httpStatusListener* передается объект `HTTPStatusEvent`, переменная `status` которого содержит код статуса HTTP. Следующий код демонстрирует типовую функцию *httpStatusListener*. Обратите внимание на способ получения кода статуса HTTP из объекта `HTTPStatusEvent e`:

```
private function httpStatusListener (e:HTTPStatusEvent):void {
  trace("http status: " + e.status);
}
```

Как бы удивительно это ни звучало, фактически приложение Flash Player осуществляет диспетчеризацию события `HTTPStatusEvent.HTTP_STATUS` для каждой отдельной операции загрузки, даже если не получает код статуса HTTP от сервера. В тех случаях, когда никакой код статуса HTTP не получен, переменной экземпляра `status` класса `HTTPStatusEvent` присваивается значение `0`. Например, во всех следующих ситуациях переменной `status` присваивается значение `0`:

- файл загружается локально или из некоторого источника не по протоколу HTTP;
- сервер HTTP недоступен;
- запрашиваемый URL-адрес указан неправильно;
- коды статуса HTTP не поддерживаются средой (например, приложение Flash Player выполняется в браузере Mozilla Firefox).

Мы рассмотрели все общие вопросы, относящиеся к загрузке отображаемых элементов на этапе выполнения с помощью объекта `Loader`. В следующих разделах рассматриваются вопросы, касающиеся использования загруженных SWF-файлов.

Проверка типов на этапе компиляции для динамически загружаемых элементов

Ранее из подразд. «Обращение к загруженному элементу» разд. «Использование класса Loader для загрузки отображаемых элементов на этапе выполнения» мы узнали, что переменная экземпляра `content` класса `Loader` ссылается на объект, представляющий загруженный элемент. Мы также узнали, что в зависимости от типа загруженного элемента переменная `content` может ссылаться либо на экземпляр класса `Bitmap`, либо на экземпляр основного класса SWF-файла. Экземпляры этих несопоставимых классов допускается присваивать переменной `content`, поскольку ее типом данных является `DisplayObject`, а класс `Bitmap` и все основные классы SWF-файлов наследуются от класса `DisplayObject`. Как результат, любым объектом, присвоенным переменной `content`, можно управлять с помощью переменных и методов класса `DisplayObject`, но над ним нельзя вызывать более специфические переменные и методы класса `Bitmap` или основного класса SWF-файла.

Например, с помощью следующего кода допустимо обратиться к переменной экземпляра `width` класса `DisplayObject` объекта, на который ссылается переменная `content`:

```
// Класс DisplayObject определяет переменную width,
// поэтому ошибки не будет
loader.content.width
```

Следующий код подобным образом пытается обратиться к переменной экземпляра `bitmapData` класса `Bitmap` объекта, на который ссылается переменная `content`. Однако на этот раз данный код вызывает ошибку компиляции, поскольку в классе `DisplayObject` не определена переменная `bitmapData`.

```
ОШИБКА: "Access of possibly undefined property bitmapData through a
         reference with static type flash.display:DisplayObject."
        (Обращение к возможно неопределенному свойству bitmapData через
         ссылку на статический класс flash.display:DisplayObject)

loader.content.bitmapData.getPixel(0, 0)
```

Чтобы избежать ошибок на этапе компиляции при обращении к методам и переменным класса `Bitmap` через переменную `content`, мы приводим ее значение к типу `Bitmap`, как показано в следующем коде:

```
Bitmap(loader.content).bitmapData.getPixel(1, 1);
```

Операция приведения типов сообщает компилятору, что загруженный элемент является экземпляром класса `Bitmap`, в котором определена переменная `bitmapData`.

Подобным образом при использовании методов и переменных класса MovieClip над загруженным SWF-файлом мы приводим значение переменной content к типу MovieClip. Например, следующий код начинает воспроизведение гипотетической анимации путем вызова метода экземпляра play( ) класса MovieClip над загруженным элементом. Операция приведения типов (обязательная) сообщает компилятору, что загруженный элемент является потомком класса MovieClip и, следовательно, поддерживает метод play( ).

```
MovieClip(loader.content).play( );
```

Точно так же, когда используются пользовательские методы и переменные основного класса загруженного SWF-файла, вполне естественно ожидать, что значение переменной content приводится к этому основному классу. Предположим, что приложение Main.swf загружает другое приложение Module.swf, основным классом которого является класс Module. Предположим также, что класс Module определяет собственный метод start( ). Когда приложение Main.swf загружает приложение Module.swf, среда выполнения Flash автоматически создает экземпляр класса Module и присваивает его переменной content. Таким образом, чтобы вызвать метод start( ) над загруженным экземпляром класса Module, можно предположить, что необходимо использовать следующую операцию приведения типов:

```
Module(loader.content).start( );
```

Хотя, по существу, предыдущий код корректен, фактически он будет вызывать ошибку на этапе компиляции до тех пор, пока не будут приняты особые меры при компиляции файла Main.swf. Рассмотрим почему.

Предположим, что приложения Main.swf и Module.swf созданы в виде отдельных проектов в приложении Flex Builder 2. Эти два проекта созданы с учетом того, что приложения должны быть автономными и независимыми, поэтому в них используется совершенно разный код и они никак не ссылаются друг на друга. В проекте приложения Module.swf определяется класс Module, однако приложение Main.swf ничего не знает про этот класс. При построении файла Main.swf компилятор встречает следующий код:

```
Module(loader.content).start( )
```

и не может разрешить ссылку на класс Module, используя пути, указанные для свойства ActionScript Build Path (Путь компиляции ActionScript) приложения Main.swf. Не найдя класс Module, компилятор предполагает, что выражение Module(loader.content) является вызовом метода. Однако метода с именем Module не существует, поэтому компилятор генерирует следующую ошибку типа:

```
1180: Call to a possibly undefined method Module.
```

По-русски она будет звучать так: Вызов, возможно, неопределенного метода Module.

Существует два способа решения данной проблемы: мы можем отключить проверку типов на этапе компиляции или предоставить компилятору доступ к классу Module при компиляции файла Main.swf. Эти способы рассматриваются в двух следующих разделах.

Отключение проверки типов на этапе компиляции

Чтобы отключить проверку типов на этапе компиляции при применении пользовательских методов и переменных основного класса загруженного SWF-файла, мы можем привести значение переменной `loader.content` к типу данных `Object`, как показано в следующем коде:

```
Object(loader.content).start( );  // Ошибки компиляции не будет
```

Кроме того, можно присвоить то значение, на которое ссылается переменная `loader.content`, нетипизированной переменной:

```
var module:* = loader.content;
module.start( );  // Ошибки компиляции не будет
```

В качестве альтернативы мы можем обратиться к загруженному объекту с помощью выражения *событиеInit*`.target.content` из функции-приемника события `Event.INIT`. Проверки типа значения переменной экземпляра `target` класса `Event` на этапе компиляции не происходит, поскольку типом данных объекта `target` является `Object`.

```
private function initListener (e:Event):void {
  e.target.content.start( );  // Ошибки компиляции не будет
}
```

В каждом из перечисленных случаев компилятор не генерирует ошибку при обращении к методу `start( )`. Вместо этого проверка типа нашего значения переменной `loader.content` откладывается до этапа выполнения. Тем не менее, как мы уже знаем из гл. 8, отключение проверки типов на этапе компиляции приводит к потере производительности. В тех случаях, когда проверка типов откладывается до этапа выполнения, сообщения об ошибках не появятся, пока не будет выполнен потенциальный проблемный код, поэтому время отладки увеличивается.

Чтобы избежать увеличения времени отладки, мы можем предоставить компилятору доступ к классу `Module` при компиляции приложения `Main.swf`. Этот процесс подробно описан в следующем разделе.

Предоставление компилятору доступа к загружаемому классу

Чтобы избежать ошибок компиляции при приведении значения переменной `loader.content` к основному классу загружаемого SWF-файла, мы можем предоставить компилятору языка ActionScript доступ к этому классу на этапе компиляции. Для этого существуют три различные методики: *путей исходных файлов*, *путей библиотек* и *путей внешних библиотек*. Каждая из перечисленных методик рассматривается в трех следующих разделах на примере класса `Module` из предыдущего раздела. В каждом случае компилятору предоставляется доступ к классу `Module` при компиляции приложения `Main.swf`.

Три методики, рассматриваемые в следующих разделах, подходят для различных ситуаций. Используйте методику *путей исходных файлов* в том случае, когда верны оба следующих утверждения:

- допускается увеличение общего размера файлов приложения;
- вы хотите сделать так, чтобы исходный код загружаемого SWF-файла (в нашем примере `Module.swf`) был непосредственно доступен автору приложения, из которого происходит обращение к этому SWF-файлу (`Main.swf` в нашем примере).

Используйте методику *путей библиотек*, когда допускается увеличение общего размера файлов приложения и когда верно одно из следующих утверждений:

- вы не хотите, чтобы исходный код загружаемого SWF-файла был непосредственно доступен автору приложения, из которого происходит обращение к этому SWF-файлу;
- время, необходимое для компиляции SWF-файла, к которому происходит обращение, должно быть минимизировано.

Используйте методику *путей внешних библиотек*, когда увеличение размера файла всего приложения не допускается.

Стоит отметить, что во всех трех последующих примерах использования методик, если приложение Main.swf уже содержит класс с именем Module, вместо класса из приложения Module.swf применяется версия данного класса из приложения Main.swf. Чтобы избежать подобного поведения, всегда уточняйте имена ваших классов, используя уникальное имя пакета (как было рассмотрено в гл. 1).

Включение файла класса Module в пути исходных файлов приложения Main.swf

Первая методика для предоставления компилятору доступа к классу `Module` при компиляции приложения `Main.swf` заключается во включении файла класса `Module` в *пути исходных файлов* приложения `Main.swf`.

Следующие шаги описывают данный процесс для приложения Flex Builder 2.

1. В палитре Navigator (Навигатор) выберите папку проекта приложения `Main.swf`.
2. В меню Project (Проект) выберите команду Properties (Свойства).
3. В появившемся окне Properties (Свойства) установите флажок ActionScript Build Path (Путь компиляции ActionScript).
4. На вкладке Source path (Пути исходных файлов) нажмите кнопку Add Folder (Добавить путь).
5. В открывшемся окне Add Folder (Добавление папки) укажите путь к папке, содержащей файл класса `Module`.
6. В окне Add Folder (Добавление папки) нажмите кнопку OK.
7. В окне Properties (Свойства) нажмите кнопку OK.

Следующие шаги описывают эквивалентный процесс включения файла класса `Module` в пути исходных файлов приложения `Main.swf` для приложения Flash CS3.

1. Откройте соответствующий FLA-файл приложения `Main.swf` — `Main.fla`.

Отметим, что шаги, описывающие процесс создания файла `Main.fla`, не рассматриваются в этом разделе. Предполагается, что файл `Main.fla` является FLA-файлом, в качестве класса документа которого выбран класс `Main`.

2. Выберите команду меню File ▸ Publish Settings (Файл ▸ Настройки публикации).
3. На вкладке Flash окна Publish Settings (Настройки публикации) нажмите кнопку Settings (Параметры), расположенную возле раскрывающегося списка ActionScript version (Версия ActionScript) с установленным значением ActionScript 3.0.
4. В появившемся окне ActionScript 3.0 Settings (Параметры ActionScript 3.0) нажмите кнопку со знаком «плюс» и укажите путь к папке, содержащей файл класса `Module`.

Как только файл класса `Module` будет включен в *пути исходных файлов* приложения `Main.swf`, компилятор сможет проверять типы для любого обращения к классу `Module`, происходящего в приложении `Main.swf`. Кроме того, компилятор добавит байт-код для класса `Module` (и для всех зависимых определений) непосредственно в файл `Main.swf`, гарантируя доступность класса `Module` из приложения `Main.swf` на этапе выполнения. Таким образом, методика *путей исходных файлов* увеличивает общий размер файлов приложения, поскольку класс `Module` и его зависимые определения включаются в файлы `Main.swf` и `Module.swf`. Более того, при использовании методики *путей исходных файлов* компиляция класса `Module` осуществляется с нуля всякий раз, когда компилируется приложение `Main.swf`, что может приводить к временным затратам.

Включение файла класса Module в пути библиотек приложения Main.swf

Вторая методика для предоставления компилятору доступа к классу `Module` при компиляции приложения `Main.swf` заключается в создании SWC-файла, содержащего класс `Module`, и включении этого SWC-файла в *пути библиотек* приложения `Main.swf`.

Чтобы создать SWC-файл, содержащий класс `Module`, в приложении Flex Builder 2, мы используем консольный компилятор компонентов *compc* (компилятор compc находится во вложенной папке Flex SDK 2\bin внутри папки, в которую было установлено приложение Flex Builder 2). Ниже представлен общий вид команды, используемой для компиляции SWC-файла с помощью компилятора compc:

```
compc -source-path путь_к_определениям -output путь_к_SWC_файлу -include-classes
именаОпределений
```

Здесь *путь_к_определениям* — список местоположений, в которых компилятор должен искать классы и другие определения при создании SWC-файла, *путь_к_SWC_файлу* — путь к создаваемому SWC-файлу, а *именаОпределений* — список определений, которые будут включены в SWC-файл (компилятор автоматически включает все зависимые определения). Предположим, мы работаем в операционной системе Windows XP и хотим создать SWC-файл с именем `module.swc` в папке `c:\apps\module\bin\`. Мы хотим, чтобы файл `module.swc` включал класс `Module`, файл которого находится в папке `c:\apps\module\src`. Чтобы создать файл `module.swc`, мы используем следующую команду:

```
compc -source-path c:\apps\module\src -output c:\apps\module\bin\module.swc -
includeclasses
Module
```

Стоит отметить, что, несмотря на свое название, параметр -include-classes компилятора compc может использоваться для включения любых типов определений, а не только классов. Возможно, в будущих версиях компилятора появится параметр с более подходящим названием — -include-definitions.

Теперь рассмотрим эквивалентный процесс создания SWC-файла, содержащего класс `Module`, в приложении Flash CS3.

1. Создайте новый документ Flash (FLA-файл) с именем `Module.fla`.
2. В поле **Document class** (Класс документа) палитры **Properties** (Свойства) введите значение `Module`.
3. Выберите команду меню **File** ▸ **Publish Settings** (Файл ▸ Настройки публикации).
4. На вкладке **Formats** (Форматы) в области **Type** (Тип) снимите флажок **HTML**.
5. На вкладке **Flash** в области **Options** (Параметры) установите флажок **Export SWC** (Экспорт SWC).
6. Нажмите кнопку **Publish** (Опубликовать), а затем кнопку **OK**.

Создав файл `module.swc`, мы включаем его в *пути библиотек* при компиляции приложения `Main.swf`. Для этого в приложении Flex Builder 2 нужно выполнить следующие шаги.

1. На палитре **Navigator** (Навигатор) выберите директорию проекта приложения `Main.swf`.
2. В меню **Project** (Проект) выберите команду **Properties** (Свойства).
3. В появившемся окне **Properties** (Свойства) установите флажок **ActionScript Build Path** (Путь компиляции ActionScript).
4. На вкладке **Library path** (Пути библиотек) нажмите кнопку **Add SWC** (Добавить SWC).
5. В окне **Add SWC** (Добавление SWC) укажите путь к файлу `module.swc`.
6. В окне **Add SWC** (Добавление SWC) нажмите кнопку **OK**.
7. Теперь в окне **Properties** (Свойства) нажмите кнопку **OK**.

Чтобы включить файл `module.swc` в пути библиотек при компиляции приложения `Main.swf` в приложении Flash CS3, выполните следующие шаги.

1. Во вложенной папке **Configuration\Components** внутри папки, в которую была установлена среда разработки Flash, создайте новую папку с именем `Module`. В Windows XP по умолчанию используется следующий путь к вложенной папке **Configuration\Components**: `C:\Program Files\Adobe\Adobe Flash CS3\en\Configuration\Components`. В операционной системе Mac OS X по умолчанию используется такой путь к вложенной папке **Configuration\Components**: `Macintosh HD:Applications:Adobe Flash CS3:Configuration:Components`.
2. Скопируйте файл `module.swc` в папку **Module**, созданную на шаге 1. В результате копирования файла `module.swc` во вложенную папку **Configuration\Components**

этот файл будет добавлен на палитру Components (Компоненты) среды разработки Flash.

3. В среде разработки Flash откройте палитру Components (Компоненты) (команда меню Window ▸ Components (Окно ▸ Компоненты)).
4. Откройте меню Options (Параметры), щелкнув на значке в правом верхнем углу палитры Components (Компоненты), и выберите команду Reload (Обновить). Папка Module появится на палитре Components (Компоненты).
5. На палитре Components (Компоненты) откройте папку Module.
6. Откройте палитру Library (Библиотека) файла `Main.fla` (команда меню Window ▸ Library (Окно ▸ Библиотека)).
7. Перетащите компонент `Module` с палитры Components (Компоненты) на палитру Library (Библиотека) файла `Main.fla`.

Как только файл `module.swc` будет включен в *пути библиотек* приложения `Main.swf`, компилятор сможет проверять типы для любого обращения к классу `Module`, происходящего в приложении `Main.swf`. Кроме того, компилятор копирует байт-код класса `Module` (и всех зависимых классов) непосредственно из файла `module.swc` в приложение `Main.swf`, гарантируя доступность класса `Module` из приложения `Main.swf` на этапе выполнения. Таким образом, как и в случае с методикой *путей исходных файлов*, методика *путей библиотек* увеличивает общий размер файлов приложения, поскольку класс `Module` и его зависимые определения включаются и в `Main.swf`, и в `Module.swf`. Тем не менее, поскольку при использовании методики *путей библиотек* класс `Module` не компилируется с нуля всякий раз, когда компилируется приложение `Main.swf`, компиляция приложения `Main.swf` с использованием методики *путей библиотек* обычно быстрее, чем с использованием методики *путей исходных файлов*.

Копирование заранее скомпилированного байт-кода из SWC- в SWF-файл оказывается быстрее, чем компиляция приложений из исходных файлов определений.

Безусловно, всякий раз, когда изменяется класс `Module`, сам файл `module.swc` необходимо создавать заново. Следовательно, если класс `Module` изменяется гораздо чаще, чем приложение `Main.swf`, время, сэкономленное при копировании байт-кода непосредственно из файла `module.swc` в файл `Main.swf`, будет полностью потрачено на компиляцию файла `module.swc`. Выражаясь современным языком, ваши результаты могут отличаться.

Включение файла класса Module в пути внешних библиотек приложения Main.swf

Третья методика для предоставления компилятору доступа к классу `Module` при компиляции приложения `Main.swf` заключается в создании SWC-файла, содержащего класс `Module`, и включении этого SWC-файла в *пути внешних библиотек* приложения `Main.swf`.

Применение этой методики начинается с выполнения инструкций из предыдущего раздела по созданию файла `module.swc`. Создав этот файл, мы включаем его

в пути внешних библиотек приложения `Main.swf`. Для этого в приложении Flex Builder 2 необходимо выполнить следующие шаги.

1. На палитре Navigator (Навигатор) выберите директорию проекта приложения `Main.swf`.
2. В меню Project (Проект) выберите команду Properties (Свойства).
3. В появившемся окне Properties (Свойства) установите флажок ActionScript Build Path (Путь компиляции ActionScript).
4. На вкладке Library path (Пути библиотек) нажмите кнопку Add SWC (Добавить SWC).
5. В окне Add SWC (Добавление SWC) укажите путь к файлу `module.swc`.
6. В окне Add SWC (Добавление SWC) нажмите кнопку OK.
7. В дереве списка Build path libraries (Пути библиотек компиляции) раскройте файл `module.swc`.
8. Среди раскрывшихся свойств файла `module.swc` выделите свойство Link Type (Тип связи) с установленным значением Merged into code (Внедрить в код).
9. Нажмите кнопку Edit (Изменить).
10. В появившемся диалоговом окне Library Path Item Options (Параметры элемента путей библиотек) для параметра Link Type (Тип связи) выберите значение External (Внешний).
11. В окне Library Path Item Options (Параметры элемента путей библиотек) нажмите кнопку OK.
12. В окне Properties (Свойства) нажмите кнопку OK.

В приложении Flash CS3, чтобы включить файл `module.swc` в *пути внешних библиотек*, мы просто помещаем его в ту же папку, где находится файл `Main.fla` (или в любую другую папку из путей к классам файла `Main.fla`), и удаляем компонент `Module` с палитры Library (Библиотека) файла `Main.fla`.

Как только файл `module.swc` будет включен в *пути внешних библиотек* приложения `Main.swf`, компилятор сможет проверять типы для любого обращения к классу `Module`, происходящего в приложении `Main.swf`. Однако, в отличие от методик *путей библиотек* и *путей исходных файлов*, когда приложение `Main.swf` компилируется с использованием методики *путей внешних библиотек*, компилятор *не* копирует байт-код класса `Module` в приложение `Main.swf`. Таким образом, общий размер файлов приложения будет минимальным. Тем не менее исключение байт-кода класса `Module` из приложения `Main.swf` приводит к новой проблеме: на этапе выполнения любая ссылка на класс `Module` из приложения `Main.swf` оказывается неизвестной для среды выполнения Flash. В связи с этим следующий код:

```
Module(loader.content).start( )
```

вызовет такую ошибку на этапе выполнения:

```
ReferenceError: Error #1065: Variable Module is not defined.
```

На русском языке она будет выглядеть так: Ошибка обращения: переменная Module не определена.

Чтобы избежать данной ошибки, мы должны заставить среду выполнения Flash импортировать классы приложения `Module.swf` в *домен приложения* файла `Main.swf` на этапе выполнения.

Домен приложения SWF-файла предоставляет доступ к классам этого файла. Домены приложения определяют, как загруженные SWF-файлы совместно используют классы и другие определения. Дополнительную информацию можно найти в разделе Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Client System Environment ▸ ApplicationDomain class документации корпорации Adobe. Кроме того, обратитесь к гл. 31.

Чтобы импортировать классы приложения `Module.swf` в домен приложения файла `Main.swf`, при создании запроса на загрузку приложения `Module.swf` мы используем объект `LoaderContext`. Рассмотрим, как выглядит код, добавляемый в основной класс приложения `Main.swf`:

```
// Сначала импортируем классы ApplicationDomain и LoaderContext...
import flash.system.*;

// ...затем в классе используем объект LoaderContext, чтобы импортировать
// классы и другие определения файла Module.swf в домен приложения
// файла Main.swf
loader.load(new URLRequest("Module.swf"),
            new LoaderContext(false, ApplicationDomain.currentDomain));
```

В результате выполнения предыдущего кода классы (и другие определения) файла `Module.swf` становятся непосредственно доступными для кода приложения `Main.swf` — будто они были определены в приложении `Main.swf`.

Стоит отметить, что, если файлы `Main.swf` и `Module.swf` находятся в разных удаленных регионах или если файл `Main.swf` находится в локальной области действия и его тип безопасности песочницы отличается от типа безопасности песочницы файла `Module.swf`, попытка импортировать классы файла `Module.swf` в домен приложения файла `Main.swf` завершится неудачно без сообщения об ошибке. В данном случае следующий код:

```
Module(loader.content).start( )
```

вызовет ту же ошибку, которая возникла бы в случае, когда классы приложения `Module.swf` вообще не импортируются в домен приложения файла `Main.swf`:

```
ReferenceError: Error #1065: Variable Module is not defined.
```

По-русски ошибка выглядит так: **Ошибка обращения: переменная Module не определена.**

В определенных ситуациях избежать подобного ограничения безопасности можно с помощью *импортирующей загрузки*, при которой приложение `Main.swf` использует объект `LoaderContext`, чтобы импортировать файл `Module.swf` в свой домен безопасности. Это демонстрирует следующий код:

```
var loaderContext:LoaderContext = new LoaderContext( );
loaderContext.applicationDomain = ApplicationDomain.currentDomain;
var loader:Loader = new Loader( );
loader.load(new URLRequest("Module.swf"), loaderContext);
```

Полную информацию по импортирующей загрузке можно получить в разд. «Импортирующая загрузка» гл. 19.

Для обзора и для справки в листинге 28.9 представлен код для классов `Main` и `Module`, рассмотренных в данном разделе. Предполагается, что файл `module.swc` был создан и включен в пути внешних библиотек приложения `Module.swf`.

Листинг 28.9. Классы Main и Module

```
// Класс Main
package {
  import flash.display.*;
  import flash.net.*;
  import flash.events.*;
  import flash.system.*;

  public class Main extends Sprite {
    private var loader:Loader;

    public function Main( ) {
      loader = new Loader( );
      loader.contentLoaderInfo.addEventListener(Event.INIT,
                                                initListener);
      loader.load(new URLRequest("Module.swf"),
                  new LoaderContext(false,
                                    ApplicationDomain.currentDomain));
    }

    private function initListener (e:Event):void {
      trace("init");
      Module(e.target.content).start( );
    }
  }
}

// Класс Module
package {
  import flash.display.Sprite;

  public class Module extends Sprite {
    public function Module( ) {
    }

    public function start ( ):void {
      trace("Module.start( ) was invoked...");
    }
  }
}
```

Мы рассмотрели несколько методик для обращения к основному классу загружаемого элемента, не вызывая ошибки типов. Из следующего раздела мы узнаем, как благополучно обращаться к элементам, созданным во втором и последующих кадрах загруженного SWF-файла.

Обращение к элементам в многокадровых SWF-файлах

Ранее из подразд. «Обращение к загруженному элементу» разд. «Использование класса Loader для загрузки отображаемых элементов на этапе выполнения» мы узнали, что в тех случаях, когда один SWF-файл загружает другой SWF-файл, все отображаемые элементы и объекты, создаваемые программным путем и размещаемые в первом кадре загружаемого SWF-файла, становятся доступными сразу после возникновения события `Event.INIT`. Таким образом, код в приемнике события `Event.INIT` может сразу же выполнять действия над этими элементами и объектами. Тем не менее код в приемнике события `Event.INIT` не может выполнять действия над элементами и объектами, которые размещаются в последующих кадрах загруженного SWF-файла.

Любой код, желающий обратиться к элементам и объектам, которые размещаются во втором или последующих кадрах загруженного SWF-файла, должен сначала убедиться в существовании этих элементов и объектов. Убедиться, что нужные элементы и объекты существуют в загруженном SWF-файле, можно двумя способами. В SWF-файле, из которого происходит обращение, нужно:

- периодически проверять существование элемента или объекта с помощью объекта `Timer`;
- зарегистрировать приемник для пользовательского события, генерируемого загруженным SWF-файлом в тот момент, когда элемент или объект становится доступным.

Рассмотрим каждую из перечисленных методик на примере. Мы снова будем использовать сценарий из предыдущего раздела, в котором приложение `Main.swf` загружает приложение `Module.swf`. Предположим, что во втором кадре основной временной шкалы приложения `Module.swf` размещается сценарий, который создает объект `TextField t`. Приложение `Main.swf` загружает приложение `Module.swf` и желает обратиться к объекту `t`. Рассмотрим сценарий временной шкалы приложения `Module.swf`:

```
stop( );
var t:TextField = new TextField( );
t.text = "hello";
addChild(t);
```

В листинге 28.10 показано, как приложение `Main.swf` загружает приложение `Module.swf`, а затем периодически проверяет существование объекта `TextField` перед его использованием.

Листинг 28.10. Периодическая проверка существования загруженного объекта

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.net.*;
  import flash.utils.*;
```

```
  public class Main extends Sprite {
    private var loader:Loader;

    public function Main( ) {
      // Загружаем файл Module.swf
      loader = new Loader( );
      loader.contentLoaderInfo.addEventListener(Event.INIT,
                                                initListener);
      loader.load(new URLRequest("Module.swf"));
    }

    private function initListener (e:Event):void {
      // Загруженный SWF-файл был проинициализирован,
      // поэтому начинаем периодическую проверку существования
      // объекта TextField.
      var timer:Timer = new Timer(100, 0);
      timer.addEventListener(TimerEvent.TIMER, timerListener);
      timer.start( );
    }

    private function timerListener (e:TimerEvent):void {
      // Проверяем, был ли создан объект TextField
      // загруженного SWF-файла
      if (loader.content.hasOwnProperty("t")) {
        // Объект TextField уже существует, поэтому мы можем благополучно
        // обратиться к нему
        trace(Object(loader.content).t.text);

        // Останавливаем таймер
        e.target.stop( );
      }
    }
  }
}
```

Теперь снова предположим, что приложение `Main.swf` загружает приложение `Module.swf` и желает обратиться к объекту `t`. На этот раз, однако, основной класс `Module` приложения `Module.swf` рассылает пользовательское событие `Module.ASSETS_READY`, когда объект `t` становится доступным. Приложение `Main.swf` регистрирует приемник для события `Module.ASSETS_READY` и обращается к объекту `t`, когда возникает данное событие. Рассмотрим код для класса `Module`, в котором определена константа события:

```
package {
  import flash.display.MovieClip;

  class Module extends MovieClip {
    // Определяем константу события
    public static const ASSETS_READY:String = "ASSETS_READY";
  }
}
```

Теперь рассмотрим сценарий, размещаемый во втором кадре основной временной шкалы приложения `Module.swf`, который осуществляет диспетчеризацию события, обозначающего доступность объекта `t`:

```
stop( );

var t:TextField = new TextField( );
t.text = "hello";
addChild(t);

dispatchEvent(new Event(Module.ASSETS_READY));
```

Наконец, в листинге 28.11 представлен код для основного класса приложения `Main.swf`. Предполагается, что на этапе компиляции компилятор языка ActionScript не имеет доступа к определениям классов загружаемого SWF-файла. В итоге этот код ссылается на событие `Module.ASSETS_READY` по его строковому имени `"ASSETS_READY"`:

```
loader.content.addEventListener("ASSETS_READY", assetsReadyListener);
```

Подобным образом в данном коде значение переменной `loader.content` приводится к типу `Object`, поэтому проверка типов откладывается до этапа выполнения:

```
Object(loader.content).t.text
```

Полную информацию о проверке типов загружаемых элементов можно найти в разд. «Проверка типов на этапе компиляции для динамически загружаемых элементов».

Листинг 28.11. Обработка события, сообщающего о доступности загруженного объекта

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.net.*;
  import flash.utils.*;

  public class Main extends Sprite {
    private var loader:Loader;

    public function Main( ) {
      // Загружаем файл Module.swf
      loader = new Loader( );
      loader.contentLoaderInfo.addEventListener(Event.INIT,
                                                initListener);
      loader.load(new URLRequest("Module.swf"));
    }

    private function initListener (e:Event):void {
      // Загруженный SWF-файл был проинициализирован, поэтому регистрируем
      // приемник для события Module.ASSETS_READY.
      loader.content.addEventListener("ASSETS_READY",
                                      assetsReadyListener);
    }
```

```
    private function assetsReadyListener (e:Event):void {
      // Объект TextField уже существует, поэтому мы можем благополучно
      // обратиться к нему
      trace(Object(loader.content).t.text);
    }
  }
}
```

До сих пор при рассмотрении вопросов, касающихся загрузки элементов, мы ограничивались автоматически создаваемым объектом элемента, на который ссылается переменная `loader.content`. Теперь рассмотрим, как создавать новые дополнительные экземпляры загруженных элементов вручную.

Создание экземпляра элемента, загружаемого на этапе выполнения

Методика создания нового экземпляра элемента, загружаемого на этапе выполнения, зависит от того, чем является данный элемент: SWF-файлом или растровым изображением. В двух следующих разделах описывается процесс создания экземпляров обоих типов элементов.

Создание экземпляра загруженного SWF-файла

Чтобы создать новый экземпляр загруженного SWF-файла, мы должны сначала получить ссылку на основной класс этого SWF-файла. Как только ссылка на класс будет получена, мы используем оператор `new` для создания экземпляра. Существуют два основных подхода для получения ссылки на основной класс загруженного SWF-файла:

- получить прямую ссылку на класс с помощью методик *путей исходных файлов*, *путей библиотек* или *путей внешних библиотек*, которые были рассмотрены ранее в разд. «Проверка типов на этапе компиляции для динамически загружаемых элементов»;
- получить ссылку на класс с помощью метода экземпляра `getDefinition( )` класса `ApplicationDomain`.

Рассмотрим оба подхода на примерах, снова обратившись к сценарию «приложение `Main.swf` загружает приложение `Module.swf`» из предыдущих разделов.

Предположим, что мы хотим создать новый экземпляр приложения `Module.swf` в приложении `Main.swf`. Сначала мы сделаем так, чтобы класс `Module` был непосредственно доступен для приложения `Main.swf`, используя методику *путей исходных файлов*, методику *путей библиотек* или методику *путей внешних библиотек*, которые были рассмотрены ранее. Повторяя изученный материал, напомним, что как только класс `Module` станет доступен для приложения `Main.swf`, мы сможем ссылаться на него напрямую, как показано в операции приведения, взятой из кода приемника события `Event.INIT`, представленного в листинге 28.9:

```
private function initListener (e:Event):void {
  trace("init");
```

```
  Module(e.target.content).start( );  // Прямая ссылка на класс Module
}
```

Точно так же, чтобы создать новый экземпляр класса `Module`, мы просто используем оператор `new`:

```
private function initListener (e:Event):void {
  var moduleObj:Module = new Module( );
}
```

Теперь предположим, что на этапе компиляции приложение `Main.swf` не имеет доступа к классу `Module`, но мы по-прежнему хотим создать новый экземпляр приложения `Module.swf` в приложении `Main.swf`. В подобной ситуации мы должны получить ссылку на класс `Module`, используя метод экземпляра `getDefinition( )` класса `ApplicationDomain`. Когда в данный метод передается имя класса, он возвращает ссылку на указанный класс. Возвращенную ссылку можно присвоить переменной типа `Class` для использования в последующих операциях создания экземпляров. Следующий код демонстрирует общую методику:

```
var SomeClass:Class = некийApplicationDomain.getDefinition("ИмяНекоегоКласса");
var obj:Object = new SomeClass( );
```

Здесь *некийApplicationDomain* — ссылка на объект `ApplicationDomain` SWF-файла, а *имяНекоегоКласса* — полностью уточненное строковое имя получаемого класса. Таким образом, чтобы получить ссылку на класс `Module` из приложения `Main.swf`, необходимо следующее:

- ссылка на объект `ApplicationDomain` приложения `Module.swf`;
- полностью уточненное имя класса `Module`.

К объекту `ApplicationDomain` SWF-файла можно обратиться через его объект `LoaderInfo`, который доступен через переменную `loaderInfo` любого экземпляра класса `DisplayObject` данного SWF-файла. Полностью уточненное имя для основного класса SWF-файла можно получить с помощью метода `flash.utils.getQualifiedClassName( )`. Как только приложение `Module.swf` будет загружено, внутри приемника события `Event.INIT` в приложении `Main.swf` мы можем использовать следующий код для получения ссылки на основной класс приложения `Module.swf`:

```
var ModuleClassName:String = getQualifiedClassName(e.target.content);
var appDomain:ApplicationDomain =
                          e.target.content.loaderInfo.applicationDomain;
// После выполнения следующей строки кода переменная ModuleClass будет
// ссылаться на основной класс приложения Module.swf
var ModuleClass:Class = appDomain.getDefinition(ModuleClassName);
```

Получив ссылку на класс `Module`, мы можем использовать ее для создания *новых* объектов:

```
var newModule:Object = new ModuleClass( );
```

Как всегда, обязательно дождитесь завершения процесса инициализации загружаемого SWF-файла перед тем, как обратиться к нему. Метод getDefinition( ) должен применяться только после того, как среда выполнения Flash осуществит диспетчеризацию события Event.INIT.

Обратите внимание, что в предыдущем коде типом данных переменной newModule является Object, а не Module, поскольку в данном примере приложение Main.swf не имеет непосредственного доступа к основному классу приложения Module.swf. Таким образом, для любых последующих обращений к методам и переменным класса Module через переменную newModule типы не будут проверяться вплоть до этапа выполнения. Если проверка типов требуется на этапе компиляции, вместо метода getDefinition( ) используйте методики *путей исходных файлов*, *путей библиотек* или *путей внешних библиотек*.

Стоит отметить, что методики, рассмотренные в этом разделе, можно применять не только для создания нового экземпляра SWF-файла, но и для создания экземпляра любого символа из этого SWF-файла. Предположим, что мы хотим создать экземпляр символа с именем Ball из приложения Module.swf. Для этого нам придется экспортировать символ Ball для кода на языке ActionScript, а затем выполнить одно из следующих действий:

- получить ссылку на экспортированный класс Ball, используя метод экземпляра getDefinition( ) класса ApplicationDomain;
- сделать так, чтобы класс Ball был непосредственно доступен для приложения Main.swf, используя методику *путей исходных файлов*, *путей библиотек* или *путей внешних библиотек*.

Создание экземпляра загруженного изображения

В отличие от элементов, представляющих SWF-файлы, новая копия загруженного элемента, представляющего изображение, не может быть создана с помощью оператора new. Вместо этого, чтобы создать новую копию загруженного элемента, представляющего изображение, мы должны создать копию пиксельных данных изображения и связать созданную копию данных с новым объектом Bitmap.

Как мы уже знаем, после завершения загрузки файла изображения загруженные пиксельные данные автоматически помещаются в объект BitmapData. Чтобы создать копию пиксельных данных загруженного изображения, мы вызываем метод BitmapData.clone( ) над этим объектом BitmapData. Данная методика продемонстрирована в следующем коде. Этот код создает копию данных загруженного изображения и передает созданную копию данных в конструктор нового объекта Bitmap. Новый объект Bitmap будет являться копией загруженного элемента растрового изображения. Обращаться к загруженному элементу, как обычно, можно только после возникновения события Event.INIT.

```
private function initListener (e:Event):void {
  // Переменная e.target.content ссылается на объект элемента,
  // представляющего загруженное растровое изображение
  var newImage:Bitmap = new Bitmap(e.target.content.bitmapData.clone( ));

  // Переменная newImage теперь содержит копию загруженного
  // растрового изображения
}
```

Использование сокетов для загрузки отображаемых элементов на этапе выполнения

В начале этой главы мы узнали, что язык ActionScript предоставляет два различных механизма для добавления внешнего отображаемого элемента в приложение на этапе выполнения, а именно классы:

- `flash.display.Loader`;
- `flash.net.Socket`, применяемый совместно с методом `loadBytes( )` класса `Loader`.

Теперь, когда мы освоили применение класса `Loader`, рассмотрим, как он может быть использован совместно с классом `Socket` для получения отображаемых элементов непосредственно через сокет TCP/IP. Методика, описываемая в данном разделе, может применяться для загрузки элементов в приложении, в котором широко используются сокеты, например в многопользовательской игре, или просто для того, чтобы предотвратить появление загруженного элемента в кэше конечного пользователя.

Общий процесс получения отображаемых элементов непосредственно через сокет TCP/IP заключается в следующем.

1. Подключиться к серверу, который может передавать среде выполнения Flash файлы GIF, PNG, JPEG или SWF в бинарном формате.
2. Получить байты для желаемого элемента.
3. Преобразовать загруженные байты в объект элемента, пригодный для отображения на экране.

Для выполнения первых двух шагов мы используем класс `flash.net.Socket` языка ActionScript. Взаимодействие класса `Socket` с сервером осуществляется посредством формата бинарных данных (необработанных байтов). Для выполнения последнего шага мы используем метод экземпляра `loadBytes( )` класса `Loader`. Метод `loadBytes( )` преобразует необработанные байты в отображаемый объект языка ActionScript.

Описанный процесс подробно рассматривается в следующих разделах.

Серверная часть: отправка элемента

С помощью класса `Socket` мы можем получать байты для отображаемого элемента с любого сервера, который знает, как отправлять файлы GIF, PNG, JPEG или SWF в бинарном формате. Например, класс `Socket` может быть использован для получения изображений с большинства почтовых, новостных и чат-серверов — они все обычно поддерживают передачу изображений в бинарном формате.

Вместо того чтобы изучать процесс загрузки элемента с сервера существующего типа (например, почтового, новостного или чат-сервера), рассмотрим более законченный

сценарий, в котором мы разработаем не только клиента, функционирующего в среде выполнения Flash и получающего элемент, но и сервер, отправляющий данный элемент. Мы назовем наш собственный сервер именем FileSender и напишем его на языке Java. Поведение сервера FileSender чрезвычайно простое: при подключении нового клиента сервер автоматически отправляет этому клиенту один файл, затем передает ASCII-символ 4 (завершение передачи) и закрывает соединение.

Отметим, что поведение сервера FileSender полностью автоматизировано: клиенту не требуется запрашивать элемент с сервера или отправлять подтверждение о получении элемента в каком-либо виде. Данная архитектура позволяет сконцентрироваться исключительно на процессе передачи элемента.

В листинге 28.12 представлен исходный код на языке Java для сервера FileSender, любезно предоставленный Дереком Клейтоном (Derek Clayton) для этой книги.

Листинг 28.12. Сервер FileSender

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.IOException;
import java.io.InputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.BufferedOutputStream;

/**
 * FileSender – это простой сервер, который принимает сокетное соединение
 * и передает файл, после чего соединение закрывается.
 *
 * Использование: java FileSender [порт] [имя_файла]
 *
 * [порт] = порт, на котором сервер будет ожидать подключения (на всех
 *          локальных IP-адресах)
 * [имя_файла] = путь к передаваемому файлу
 *
 */
public class FileSender implements Runnable {
    private int port;
    private File file;
    private String filename;
    private ServerSocket server;
    private Thread thisThread;
    private byte[] bytes;

    public FileSender(int p, String f) {
        port = p;
        filename = f;
    }

    public void start( ) {
        InputStream is = null;
        try {
```

```
            // --- читаем файл в наш массив байтов
            file = new File(filename);
            is = new FileInputStream(file);
            bytes = new byte[(int)file.length( )+1];
            int offset = 0;
            int byteRead = 0;
            while (offset < bytes.length
                   && (byteRead=is.read(bytes, offset, bytes.length-offset))
                       >= 0) {
                offset += byteRead;
            }
            bytes[bytes.length-1] = 4;

            // --- создаем объект ServerSocket
            server = new ServerSocket(port);
        } catch (Exception e) {
            e.printStackTrace( );
            System.exit(1);
        } finally {
            if (is != null) {
                try {
                    is.close( );
                } catch (Exception e) {
                    e.printStackTrace( );
                    System.exit(1);
                }
            }
        }

        // --- запускаем объект Thread, который будет принимать подключения
        thisThread = new Thread(this);
        thisThread.start( );
    }

    public void run( ) {
        // --- пока сервер активен...
        while (thisThread != null) {
            BufferedOutputStream ps = null;
            Socket socket = null;
            try {
                // --- ...принимаем сокетные соединения
                //     (выполнение потока блокируется, пока не будет
                //      установлено соединение)
                socket = server.accept( );

                // --- создаем выходной поток
                ps = new BufferedOutputStream(socket.getOutputStream( ));

                // --- записываем байты и закрываем соединение
                ps.write(bytes);
                ps.close( );
```

```
            ps = null;
            socket.close( );
            socket = null;
        } catch(Exception e) {
            thisThread = null;
            e.printStackTrace( );
        } finally {
            if (ps != null) {
                try {
                    ps.close( );
                } catch (IOException e) {
                    e.printStackTrace( );
                    System.exit(1);
                }
            }

            if (socket != null) {
                try {
                    socket.close( );
                } catch (IOException e) {
                    e.printStackTrace( );
                    System.exit(1);
                }
            }
        }
    }

    // --- освобождаем ресурсы, занимаемые сервером
    if (server != null) {
        try {
            server.close( );
        } catch (IOException e) {
            e.printStackTrace( );
            System.exit(1);
        }
    }
}

public final static void main(String [] args) {
    // --- проверяем, все ли аргументы указаны
    if (args.length != 2) {
        System.out.println("usage: java FileSender [port] [file]");
        System.exit(1);
    }

    try {
        // --- преобразуем аргументы к их соответствующему типу
        int port = Integer.parseInt(args[0]);
        String filename = args[1];


        // --- создаем и запускаем объект FileSender
```

```
            //     (который будет выполняться
            //     в своем собственном потоке)
            FileSender fs = new FileSender(port, filename);
            fs.start( );
        } catch (Exception e) {
            e.printStackTrace( );
            System.exit(1);
        }
    }
}
```

Исходный код для сервера FileSender можно загрузить по адресу http://moock.org/eas3/examples.

Чтобы запустить сервер FileSender, мы вводим следующую команду для среды выполнения Java:

```
java FileSender порт имяФайла
```

Здесь *порт* — это порт, на котором сервер будет принимать соединения, а *имяФайла* — имя файла, передаваемого сервером FileSender любому подключившемуся клиенту. Например, чтобы запустить сервер на порте 3000 и сконфигурировать его на отправку файла с именем `photo.jpg`, мы вводим следующую команду для среды выполнения Java:

```
java FileSender 3000 photo.jpg
```

Информацию об ограничениях безопасности, касающихся сокетных соединений, можно найти в гл. 19.

Клиентская часть: получение элемента

Мы только что рассмотрели код для пользовательского сервера, написанного на языке Java, который автоматически отправляет указанный файл любому подключившемуся клиенту. Теперь создадим соответствующего клиента на языке ActionScript, подключаемого к серверу и получающего файл.

В отличие от многих традиционных языков программирования, система сокетных взаимодействий языка ActionScript построена полностью на событиях.

В ActionScript невозможно приостановить выполнение программы, ожидая появления данных в сокете. Иными словами, сокетные операции языка ActionScript являются асинхронными, а не синхронными.

В языке ActionScript данные могут быть прочитаны из сокета только после того, как возникнет событие `ProgressEvent.SOCKET_DATA`. Данное событие сообщает о том, что клиенту доступен некий произвольный объем новых данных для чтения. Однако как только операция чтения новых данных будет завершена, клиент должен снова дожидаться возникновения следующего события `ProgressEvent.SOCKET_DATA`, чтобы прочитать дополнительные данные из сокета. Рассмотрим общее описание процесса.

1. Клиент подключается к сокету.
2. Сокет получает некоторые данные.
3. Возникает событие `ProgressEvent.SOCKET_DATA`.
4. Клиент считывает все доступные данные.
5. Сокет получает дополнительные данные.
6. Возникает событие `ProgressEvent.SOCKET_DATA`.
7. Клиент считывает все доступные данные.
8. Повторение шагов 5–7 до тех пор, пока не будет закрыт сокет.

Объем данных, которые появляются в сокете с возникновением каждого события `ProgressEvent.SOCKET_DATA`, полностью произволен. Зачастую данные, доступные клиенту при возникновении события `ProgressEvent.SOCKET_DATA`, составляют лишь часть большого целого. Таким образом, вы должны проявлять особую осторожность, выполняя ручную сборку всех необходимых данных перед их обработкой. Клиент может получить данные некоторого изображения, скажем, в трех сегментах, каждый из которых вызывает событие `ProgressEvent.SOCKET_DATA`. Перед тем как обработать все изображение целиком, клиентский код должен собрать эти три сегмента вместе.

Чтобы собрать небольшие сегменты данных в одно целое, клиент должен вручную добавлять каждый сегмент во временный массив байтов (то есть в «буфер байтов») по мере загрузки этого целого. «Целым» может быть файл, объект, готовая инструкция, почтовое сообщение, сообщение чата или любая другая логическая структура данных, которая должна обрабатываться как единый элемент. Всякий раз, когда появляется новый сегмент, клиент проверяет, не завершена ли загрузка всего целого. Если да, то клиент приступает к его обработке. Если нет, то клиент ожидает появления дополнительных данных.

Стоит отметить, однако, что не существует официального способа проверить, было ли полностью получено некоторое логическое тело данных. Каждый бинарный сокетный протокол предоставляет свои собственные механизмы, позволяющие определить момент завершения текущей передачи данных. Например, сервер может сообщить своему клиенту, сколько данных будет передано до начала следующей передачи, или клиент может проверять загружаемую последовательность байтов на наличие маркеров начала и конца файла. В нашем примере клиент будет приступать к обработке загруженного отображаемого элемента сразу после закрытия сокетного соединения сервером. Закрытие сокетного соединения — это (очень простой) способ сообщить нашему клиенту о завершении отправки данных некоторого элемента.

Посмотрим, как все это будет выглядеть в коде. Наш простой клиент, написанный на ActionScript, состоит из одного класса `DisplayAssetLoader`. Чтобы получить и отобразить элемент, отправляемый сервером FileSender, класс `DisplayAssetLoader` должен выполнить такую последовательность действий.

1. Создать объект `Socket`.
2. Зарегистрировать объект `DisplayAssetLoader` для событий объекта `Socket`.
3. Использовать объект `Socket` для подключения к серверу.
4. Когда через сокет будут получены новые бинарные данные, поместить эти данные во временный буфер.

5. Когда сокет отсоединится, использовать метод экземпляра `loadBytes( )` класса `Loader`, для того чтобы загрузить бинарные данные из временного буфера в объект `Loader`.
6. Отобразить объект загруженного элемента на экране.

В листинге 28.13 представлен весь код целиком для класса `DisplayAssetLoader`. Ключевые возможности класса `DisplayAssetLoader` обсуждаются после листинга; незначительные детали описываются в виде комментариев к коду.

Листинг 28.13. Класс DisplayAssetLoader

```
package {
  import flash.display.*;
  import flash.events.*;
  import flash.net.*;
  import flash.text.*;
  import flash.utils.*;

  public class DisplayAssetLoader extends Sprite {
    // Константа, представляющая ASCII-символ
    // «завершение передачи»
    public static const EOT:int = 4;
    // Объект TextField, отображаемый на экране,
    // в который выводятся
    // статусные сообщения
    private var statusField:TextField;
    // Объект сокета, через который будет устанавливаться соединение
    private var socket:Socket;
    // Буфер байтов, в который по мере загрузки будут помещаться бинарные
    // данные элемента
    private var buffer:ByteArray = new ByteArray( );
    // Объект Loader, используемый для генерации элемента из загруженных
    // бинарных данных
    private var loader:Loader;

    // Конструктор класса
    public function DisplayAssetLoader ( ) {
      // Создаем объект TextField для отображения статусных сообщений
      statusField = new TextField( );
      statusField.border     = true;
      statusField.background = true;
      statusField.width = statusField.height = 350;
      addChild(statusField);

      // Создаем объект сокета
      socket = new Socket( );

      // Регистрируем приемники для событий сокета
      socket.addEventListener(Event.CONNECT, connectListener);
      socket.addEventListener(Event.CLOSE, closeListener);
      socket.addEventListener(ProgressEvent.SOCKET_DATA,
                              socketDataListener);
      socket.addEventListener(IOErrorEvent.IO_ERROR, ioErrorListener);
```

```
    // Сообщаем пользователю, что сейчас мы попытаемся подключиться
    // к сокету
    out("Attempting connection...");

    // Пытаемся подключиться к сокету
    try {
      socket.connect("localhost", 3000);
    } catch (e:Error) {
      out("Connection problem!\n");
      out(e.message);
    }
  }

  // Обрабатывает события подключения к сокету
  private function connectListener (e:Event):void {
    out("Connected! Waiting for data...");
  }

  // Обрабатывает вновь полученные данные
  private function socketDataListener (e:ProgressEvent):void {
    out("New socket data arrived.");

    // Когда появляются новые байты, помещаем их в буфер для дальнейшей
    // обработки
    socket.readBytes(buffer, buffer.length, socket.bytesAvailable);
  }

  // Обрабатывает события отключения сокета. Когда происходит отключение,
  // пытаемся сгенерировать отображаемый элемент из загруженных байтов.
  private function closeListener (e:Event):void {
    // Сначала проверяем, был ли получен весь элемент целиком...
    // Обращаемся к последнему байту в буфере
    buffer.position = buffer.length - 1;
    var lastByte:int = buffer.readUnsignedByte( );
    // Если байт «завершение передачи» отсутствует,
    // значит, бинарные данные элемента загружены не полностью,
    // поэтому элемент не создаем
    if (lastByte != DisplayAssetLoader.EOT) {
      return;
    }

    // Все в порядке, мы можем благополучно сгенерировать элемент из
    // загруженных байтов. Последний байт в буфере не является частью
    // элемента, поэтому отбрасываем его.
    buffer.length = buffer.length - 1;

    // Теперь создаем объект Loader, который сгенерирует элемент
    // из загруженных байтов
    loader = new Loader( );

    // Генерируем элемент из загруженных байтов
    loader.loadBytes(buffer);
```

```
      // Ожидаем завершения процесса инициализации элемента
      loader.contentLoaderInfo.addEventListener(Event.INIT,
                                                assetInitListener);
    }

    // Помещает элемент на экран после завершения процесса его инициализации
    private function assetInitListener (e:Event):void {
      addChild(loader.content);
      out("Asset initialized.");
    }

    // Обрабатывает ошибки ввода/вывода
    private function ioErrorListener (e:IOErrorEvent):void {
      out("I/O Error: " + e.text);
    }

    // Выводит статусные сообщения на экран и на отладочную консоль
    private function out (msg:*):void {
      trace(msg);
      statusField.appendText(msg + "\n");
    }
  }
}
```

Рассмотрим три основные части класса `DisplayAssetLoader`: создание и подключение к сокету, помещение байтов в буфер и создание отображаемого элемента из загруженных байтов.

Создание и подключение к сокету

Для установки сокетного соединения и управления им мы используем экземпляр класса `Socket`, который присваиваем закрытой переменной `socket`:

```
socket = new Socket( );
```

Чтобы подключиться к сокету, мы используем метод экземпляра `connect( )` класса `Socket`. Однако поскольку сокетные соединения потенциально могут генерировать ошибки безопасности и ошибки ввода/вывода, мы помещаем вызов метода `connect( )` в блок `try/catch`:

```
try {
  socket.connect("localhost", 3000);
} catch (e:Error) {
  out("Connection problem!\n");
  out(e.message);
}
```

Помещение байтов в буфер

Как мы уже знаем, при получении новых данных через сокет среда Flash выполняет диспетчеризацию события `ProgressEvent.SOCKET_DATA`. Получателем данного события является объект `Socket`, получивший эти данные. Таким образом, чтобы получать уведомления о появлении новых данных, мы регистрируем в объекте `socket` приемник для события `ProgressEvent.SOCKET_DATA`, как показано в следующем коде:

```
socket.addEventListener(ProgressEvent.SOCKET_DATA, socketDataListener);
```

Всякий раз, когда возникает событие `ProgressEvent.SOCKET_DATA`, вызывается метод экземпляра `socketDataListener( )` класса `DisplayAssetLoader`. Внутри метода `socketDataListener( )` переменная экземпляра `bytesAvailable` класса `Socket` обозначает количество байтов в сокете, доступных для чтения на настоящий момент. Метод `socketDataListener( )` добавляет новые полученные данные в объект `ByteArray`, на который ссылается переменная `buffer`. Чтобы прочитать новые данные и сохранить их в объекте `buffer`, мы используем метод экземпляра `readBytes( )` класса `Socket`, который принимает следующий общий вид:

объектSocket`.readBytes(`*байты*`, `*смещение*`, `*длина*`)`

Здесь *объектSocket* — это объект `Socket`, из которого будут прочитаны байты, *байты* — объект `ByteArray`, в который будут записаны байты, *смещение* — позиция внутри объекта *байты*, с которой начнется запись, а *длина* — количество байтов, которое будет прочитано из сокета. В нашем случае мы хотим прочитать все байты из сокета и сохранить их в объекте `buffer`, начиная с конца объекта `buffer`. Следовательно, мы используем такой код:

```
socket.readBytes(buffer, buffer.length, socket.bytesAvailable);
```

Рассмотрим полный листинг кода для метода `socketDataListener( )`:

```
private function socketDataListener (e:ProgressEvent):void {
  out("New socket data arrived.");
  socket.readBytes(buffer, buffer.length, socket.bytesAvailable);
}
```

Создание отображаемого элемента из загруженных байтов

Как только все байты для элемента будут получены, мы можем использовать метод экземпляра `loadBytes( )` класса `Loader` для создания из этих байтов экземпляра класса `DisplayObject` языка ActionScript. Напомним, что сервер FileSender сообщает о завершении передачи данных в полном объеме путем простого закрытия сокетного соединения. Поэтому мы помещаем код, создающий наш экземпляр класса `DisplayObject` («объект элемента»), в приемник события `Event.CLOSE`. Чтобы зарегистрировать приемник для события `Event.CLOSE` в объекте `socket`, мы используем следующий код:

```
socket.addEventListener(Event.CLOSE, closeListener);
```

В функции `closeListener( )` перед созданием экземпляра класса `DisplayObject` мы сначала проверяем, был ли получен байт «завершение передачи» (ASCII-код 4) от сервера. Напомним, что сервер отправляет байт «завершение передачи» в качестве последнего байта потока данных. Чтобы получить этот байт, мы используем следующий код:

```
buffer.position = buffer.length - 1;
var lastByte:int = buffer.readUnsignedByte( );
```

Если байт «завершение передачи» отсутствует, бинарные данные элемента были получены не полностью, и поэтому функция-приемник завершает свое выполнение:

```
if (lastByte != DisplayAssetLoader.EOT) {
  return;
}
```

Если байт «завершение передачи» *был* получен, мы создаем объект элемента с помощью метода `loadBytes( )`. Тем не менее перед передачей байтов элемента в метод `loadBytes( )` мы должны сначала удалить байт «завершение передачи» из буфера, как показано в следующем коде:

```
buffer.length = buffer.length - 1;
```

Теперь, имея на руках байты элемента, мы можем создать объект элемента:

```
loader = new Loader( );
loader.loadBytes(buffer);
```

После завершения выполнения предыдущего кода запускается процесс генерации объекта элемента, однако сам объект элемента пока недоступен. Как и в случае с методом экземпляра `load( )` класса `Loader`, если попытаться обратиться к объекту элемента через переменную `loader.content` сразу после вызова метода `loadBytes( )`, будет возвращено значение `null`. Таким образом, перед тем как обратиться к объекту элемента, мы должны дождаться возникновения события `Event.INIT`:

```
loader.contentLoaderInfo.addEventListener(Event.INIT,
                                          assetInitListener);
```

Когда элемент станет доступен, мы добавляем его на экран:

```
private function assetInitListener (e:Event):void {
  addChild(loader.content);
  out("Asset initialzed.");
}
```

Напомним, что при возникновении события `Event.INIT` становятся доступными элементы и объекты, созданные в конструкторе основного класса SWF-файла или в коде, который размещается в первом кадре временной шкалы SWF-файла, но при этом элементы и объекты, создаваемые во втором и последующих кадрах, недоступны. Дополнительные сведения об обращении к визуальным элементам и объектам, создаваемым во втором и последующих кадрах, можно найти ранее в разд. «Обращение к элементам в многокадровых SWF-файлах».

Удаление SWF-элементов, загруженных на этапе выполнения

Чтобы удалить из приложения SWF-элемент, загруженный на этапе выполнения, мы должны сначала обнулить все ссылки на него в приложении, а затем либо обнулить все ссылки на объект `Loader`, загрузивший этот элемент, либо вызвать метод `unload( )` над данным объектом `Loader`.

Однако перед тем, как удалить элемент из приложения, мы должны деактивировать сам элемент и всех его отображаемых детей, если таковые имеются, чтобы после удаления он не продолжал потреблять системные и сетевые ресурсы.

Обычные задачи, необходимые для деактивации загруженного SWF-файла, распределяются между приложением, загрузившим этот элемент, и самим элементом.

Приложение, загрузившее элемент, перед удалением этого элемента должно выполнить следующее.

- Если загрузка элемента еще не завершена, приложение должно прекратить операцию загрузки. Например:

```
try {
  theAssetLoader.close( );
}
catch (e:*) {}
```

- При обнулении всех ссылок на элемент приложение должно удалить элемент из всех родительских экземпляров объекта `DisplayObjectContainer`. Например:

```
container.removeChild(theAsset);
```

Сам элемент перед удалением должен выполнить следующие задачи.

- Сообщить любым загруженным дочерним SWF-элементам о необходимости деактивации.
- Остановить воспроизведение любых звуковых файлов.
- Остановить воспроизведение основной временной шкалы, если ее воспроизведение происходит в настоящий момент.
- Остановить воспроизведение любых клипов, проигрываемых в настоящий момент.
- Закрыть любые подключенные сетевые объекты, например экземпляры классов `Loader`, `URLLoader`, `Socket`, `XMLSocket`, `LocalConnection`, `NetConnections` и `NetStream`.
- Обнулить все ссылки на объекты `Camera` или `Microphone`.
- Отменить регистрацию всех приемников событий (особенно это касается события `Event.ENTER_FRAME`, а также приемников событий мыши и клавиатуры).
- Остановить все интервалы, выполняющиеся в настоящий момент (с помощью метода `clearInterval( )`).
- Остановить все объекты `Timer` (с помощью метода экземпляра `stop( )` класса `Timer`).

Элемент должен выполнять перечисленные задачи в собственном методе, освобождающем потребляемые ресурсы, который вызывается приложением перед удалением данного элемента.

За дополнительной информацией о деактивации элементов перед их удалением обратитесь к разд. «Деактивация объектов» гл. 14. Кроме того, прочитайте серию интернет-статей Гранта Скиннера (Grant Skinner) под названием «ActionScript 3.0: Resource Management», опубликованных по адресу http://www.gskinner.com/blog/archives/2006/06/as3_resource_ma.html.

Мы познакомились со всеми методиками добавления внешнего элемента в приложение на этапе выполнения. Далее рассмотрим, как добавлять внешний отображаемый элемент в приложение на этапе компиляции.

Встраивание отображаемых элементов на этапе компиляции

Чтобы включить внешний отображаемый элемент в приложение, написанное на языке ActionScript, на этапе компиляции, мы используем тег метаданных `[Embed]`. Этот тег добавляет указанный внешний элемент в SWF-файл и делает этот элемент доступным для программы в виде определенного пользователем или автоматически сгенерированного класса. Экземпляры встроенного элемента создаются из этого класса с помощью стандартного синтаксиса оператора `new` языка ActionScript.

Тег метаданных [Embed] поддерживается приложением Flex Builder 2 и консольным компилятором mxmlc. Однако тег метаданных [Embed] не поддерживается приложением Adobe Flash CS3. Возможно, поддержка тега [Embed] появится в будущих версиях среды разработки Flash.

Для использования тега `[Embed]` мы должны предоставить компилятору доступ к библиотеке `flex.swc`, поддерживающей компилятор Flex. По умолчанию все проекты, создаваемые в приложении Flex Builder 2, автоматически включают библиотеку `flex.swc` в пути библиотек языка ActionScript, поэтому в приложении Flex Builder 2 методики, рассматриваемые в данном разделе, будут работать без каких-либо специальных настроек компилятора.

Замечание по размеру файла и потреблению памяти

В отличие от методик загрузки элементов на этапе выполнения, рассмотренных в этой главе, встраивание отображаемого элемента на этапе компиляции с помощью тега метаданных `[Embed]` увеличивает размер SWF-файла, загружающего этот элемент, а также увеличивает объем используемой средой выполнения Flash памяти. По этой причине встраивать элемент на этапе компиляции необходимо только в том случае, когда вы абсолютно уверены в том, что приложению обязательно понадобится этот элемент. В противном случае загружать данный элемент следует на этапе выполнения по мере необходимости.

Например, представьте приложение, которое является электронным каталогом продукции с изображениями тысячи товаров и одним изображением экрана приветствия. Изображение экрана приветствия выводится при каждом запуске приложения, и, следовательно, его целесообразно встроить с помощью тега метаданных `[Embed]`. В отличие от этого, изображение товара необходимо только в том случае, когда пользователь просматривает нужный товар, и, следовательно, оно должно загружаться на этапе выполнения и удаляться, когда пользователь завершит просмотр этого товара.

Теперь, когда мы узнали, в каких случаях применяется тег `[Embed]`, посмотрим, как использовать его на практике. В следующем разделе рассматривается обобщенный код, необходимый для использования тега `[Embed]`. В последующих разделах приведены конкретные примеры использования обобщенного кода, рассматриваемого в следующем разделе.

Обобщенный синтаксис тега [Embed]

Тег метаданных [Embed] может применяться либо на уровне определения переменной экземпляра, либо на уровне определения класса.

Следующий код демонстрирует типовое использование тега [Embed] на уровне определения переменной:

```
[Embed(source="путьКФайлу")]
private var ИмяКласса:Class;
```

При компиляции данного кода компилятор языка ActionScript автоматически генерирует новый класс, представляющий внешний элемент, местоположение которого определяется значением *путьКФайлу*, и присваивает этот класс закрытой переменной с именем *ИмяКласса*. Суперклассом нового класса является один из классов в пространстве имен mx.core, предназначенных для встраивания элементов. Как мы увидим в последующих разделах, выбор конкретного класса-прослойки зависит от типа встраиваемого элемента. Значение *путьКФайлу* должно определять местоположение файла элемента с помощью одного из следующих способов:

- абсолютной ссылки, предоставляющей компилятору доступ к элементу на локальном компьютере (например, c:/assets/photo.jpg);
- относительной ссылки, сформированной по отношению к исходному файлу на языке ActionScript, из которого происходит встраивание элемента (например, ../images/photo.jpg).

Стоит отметить, что, поскольку переменная *ИмяКласса* ссылается на класс, ее типом данных является Class. Когда переменная ссылается на объект Class, первая буква в имени этой переменной обычно записывается в верхнем регистре (в соответствии с общепринятым стилем именования классов).

Установив связь между встраиваемым элементом и переменной *ИмяКласса*, мы используем следующий знакомый нам код для создания нового экземпляра элемента.

```
new ИмяКласса( );
```

Теперь вернемся к обобщенному синтаксису для использования тега метаданных [Embed] на уровне определения класса. Следующий код демонстрирует общий подход:

```
[Embed(source="путьКФайлу")]
public class ИмяКласса extends ТипЭлемента {
}
```

При выполнении предыдущего кода внешний элемент, путь к которому определяется значением *путьКФайлу*, связывается с классом *ИмяКласса*. Класс *ИмяКласса* должен быть определен с использованием модификатора управления доступом public и должен расширять класс *ТипЭлемента*, который является одним из классов-прослоек пространства имен mx.core платформы разработки Flex, предназначенных для встраивания элементов. В последующих разделах будет рассмотрено, какие конкретные классы-прослойки должны применяться для различных типов элементов, встраиваемых на уровне класса.

Чтобы создать новый экземпляр элемента, встроенного на уровне класса, мы снова используем следующий стандартный код:

```
new ИмяКласса( );
```

Поддерживаемые типы элементов

При использовании тега метаданных `[Embed]` либо на уровне переменной, либо на уровне класса, поддерживаются следующие типы отображаемых элементов:

- растровые изображения в формате GIF, JPEG или PNG;
- файлы в формате SVG;
- символы из SWF-файлов предыдущих версий (то есть в формате приложения Flash Player 8 или более старых версий);
- любой файл, представляющий бинарные данные.

Кроме того, встраивание SWF-файлов целиком, независимо от их версии, поддерживается тегом `[Embed]` только на уровне определения переменной.

Стоит отметить, что с помощью тега метаданных `[Embed]` нельзя встроить отдельные символы и классы из SWF-файла в формате приложения Flash Player 9 (или более поздних версий). Вместо этого для обращения к отдельным символам и классам следует использовать один из следующих способов.

- Встроить SWF-файл символа или класса на уровне переменной (дополнительные сведения можно получить в подразд. «Встраивание SWF-файлов целиком»), а затем использовать метод экземпляра `getDefinition( )` класса `ApplicationDomain` для обращения к этому символу или классу (дополнительную информацию можно найти в подразд. «Использование метода getDefinition( ) для обращения к классу во встроенном SWF-файле»).
- Встроить SWF-файл символа или класса в виде бинарных данных (дополнительную информацию можно найти в подразд. «Встраивание файлов в виде бинарных данных»), а затем использовать метод экземпляра `getDefinition( )` класса `ApplicationDomain` для обращения к этому символу или классу (дополнительные сведения можно получить в подразд. «Использование метода getDefinition( ) для обращения к классу во встроенном SWF-файле»).
- Создать ссылку на SWC-файл, содержащий класс или символ (дополнительную информацию можно найти в подразд. «Предоставление компилятору доступа к загружаемому классу» разд. «Проверка типов на этапе компиляции для динамически загружаемых элементов»).

Помимо отображаемых элементов, рассматриваемых в данном разделе, тег метаданных [Embed] может применяться для встраивания звуковых файлов и шрифтов.

Информацию по встраиванию звуковых файлов можно найти в разделе Flex Programming Topics ▸ Embedding Assets руководства разработчика на платформе Flex 2 корпорации Adobe.

Информацию о встраивании шрифтов можно найти в разд. «Шрифты и отображение текста» гл. 27. Дополнительную информацию также можно найти в разделе

Customizing the User Interface ▸ Using Fonts ▸ Using Embedded fonts ▸ Embedded font syntax ▸ Embedding Fonts in ActionScript руководства разработчика на платформе Flex 2 корпорации Adobe.

Теперь, когда мы познакомились с основным кодом, используемым для встраивания внешнего элемента на этапе компиляции, рассмотрим несколько конкретных примеров.

Встраивание растровых изображений

Следующий код демонстрирует, как встроить изображение с именем `photo.jpg` на уровне переменной. Код предполагает, что файл класса, из которого происходит встраивание изображения, и файл изображения находятся в одной директории. При выполнении этого кода среда Flash автоматически генерирует класс, представляющий элемент `photo.jpg`, и присваивает этот класс переменной `Photo`, позволяя создавать экземпляры данного элемента на этапе выполнения. Этот автоматически сгенерированный класс расширяет класс `mx.core.BitmapAsset`.

```
[Embed(source="photo.jpg")]
private var Photo:Class;
```

Следующий код показывает, как встраивать изображение `photo.jpg` на уровне класса. Снова предполагается, что файл класса и файл элемента находятся в одной директории. Обратите внимание, что по необходимости класс расширяет класс `mx.core.BitmapAsset`.

```
package {
  import mx.core.BitmapAsset;

  [Embed(source="photo.jpg")]
  public class Photo extends BitmapAsset {
  }
}
```

Чтобы создать новый экземпляр встроенного изображения, мы используем следующий код (независимо от того, было изображение встроено на уровне переменной или на уровне класса):

```
new Photo( )
```

Присваивая экземпляр встроенного изображения переменной, в качестве типа данных этой переменной мы выбираем либо класс `mx.core.BitmapAsset` (для элементов, встраиваемых на уровне переменной), либо класс `Photo` (для элементов, встраиваемых на уровне класса):

```
var photo:BitmapAsset = new Photo( );  // Уровень переменной
var photo:Photo = new Photo( );        // Уровень класса
```

Созданный экземпляр, как и любой другой отображаемый объект, может быть добавлен в список отображения:

```
addChild(photo);
```

Обратите внимание, что тег метаданных `[Embed]` поддерживает механизм форматирования scale-9 для встраиваемых растровых изображений. Если для встраивае-

мого растрового изображения применяется механизм форматирования scale-9, то автоматически сгенерированный класс расширяет класс `mx.core.SpriteAsset`, а не класс `mx.core.BitmapAsset`. Подробную информацию о механизме форматирования scale-9 и встраиваемых растровых изображениях можно найти в разделе Flex Programming Topics ▸ Embedding Assets ▸ Embedding asset types ▸ Using scale-9 formatting with embedded images руководства разработчика на платформе Flex 2 корпорации Adobe.

Встраивание файлов в формате SVG

Следующий код демонстрирует, как встроить изображение в формате SVG с именем `line.svg` на уровне переменной. Код предполагает, что файл класса, из которого происходит встраивание изображения в формате SVG, и файл изображения в формате SVG находятся в одной директории. При выполнении этого кода среда Flash автоматически генерирует класс, представляющий элемент `line.svg`, и присваивает этот класс переменной `SVGLine`, позволяя создавать экземпляры данного элемента на этапе выполнения. Этот автоматически сгенерированный класс расширяет класс `mx.core.SpriteAsset`.

```
[Embed(source="line.svg")]
private var SVGLine:Class;
```

Следующий код показывает, как встраивать изображение в формате SVG с именем `line.svg` на уровне класса. Снова предполагается, что файл класса и файл элемента находятся в одной директории. Обратите внимание, что по необходимости класс расширяет класс `mx.core.SpriteAsset`.

```
package {
  import mx.core.SpriteAsset;

  [Embed(source="line.svg")]
  public class SVGLine extends SpriteAsset {
  }
}
```

Чтобы создать новый экземпляр встроенного изображения в формате SVG, мы используем следующий код (независимо от того, было изображение в формате SVG встроено на уровне переменной или на уровне класса):

```
new SVGLine( )
```

Присваивая экземпляр встроенного изображения в формате SVG переменной, в качестве типа данных этой переменной мы выбираем либо класс `mx.core.SpriteAsset` (для элементов, встраиваемых на уровне переменной), либо класс `SVGLine` (для элементов, встраиваемых на уровне класса):

```
var line:SpriteAsset = new SVGLine( );  // Уровень переменной
var line: SVGLine = new SVGLine( );     // Уровень класса
```

Созданный экземпляр, как и любой другой отображаемый объект, может быть добавлен в список отображения:

```
addChild(line);
```

Встраивание SWF-файлов целиком

Следующий код демонстрирует, как целиком встроить SWF-файл с именем `App.swf` на уровне переменной. Код предполагает, что файл класса, из которого происходит встраивание SWF-файла, и сам SWF-файл находятся в одной директории. При выполнении этого кода среда Flash автоматически генерирует класс, представляющий элемент `App.swf`, и присваивает этот класс переменной `App`, позволяя создавать экземпляры данного элемента на этапе выполнения. Этот автоматически сгенерированный класс расширяет класс `mx.core.MovieClipLoaderAsset`.

```
[Embed(source="App.swf")]
private var App:Class;
```

Целиком SWF-файлы могут быть встроены только на уровне переменной.

Чтобы создать новый экземпляр встроенного SWF-файла, мы используем следующий код:

```
new App( )
```

Присваивая экземпляр встроенного SWF-файла переменной, в качестве типа данных этой переменной мы выбираем класс `mx.core.MovieClipLoaderAsset`:

```
var app:MovieClipLoaderAsset = new App( );
```

Созданный экземпляр, как и любой другой отображаемый объект, может быть добавлен в список отображения:

```
addChild(app);
```

Встраивание символов из SWF-файлов предыдущих версий

Следующий код демонстрирует, как на уровне переменной встроить отдельный символ с именем `Ball` из SWF-файла с именем `fp8app.swf` в формате приложения Flash Player 8 или более ранней версии. Код предполагает, что файл класса, из которого происходит встраивание символа, и SWF-файл, содержащий этот символ, находятся в одной директории. При компиляции этого кода компилятор языка ActionScript автоматически генерирует класс, представляющий символ `Ball`, и присваивает этот класс переменной `FP8Ball`, позволяя создавать экземпляры данного символа на этапе выполнения. Автоматически сгенерированный класс расширяет класс элемента из пространства имен `mx.core`, соответствующий типу символа `Ball` (то есть классы `MovieClipAsset`, `TextFieldAsset`, `ButtonAsset` или класс `SpriteAsset` для клипов, состоящих из одного кадра). В коде обратите внимание на использование дополнительного параметра `symbol` тега `[Embed]`:

```
[Embed(source="fp8app.swf", symbol="Ball")]
private var FP8Ball:Class;
```

Следующий код демонстрирует, как на уровне класса встроить отдельный символ с именем `Ball` из SWF-файла с именем `fp8app.swf` в формате приложения Flash Player 8 или более ранней версии. Вновь предполагается, что файл класса и файл

элемента находятся в одной директории. В этом примере мы считаем, что символ `Ball` является клипом, поэтому класс элемента по необходимости расширяет класс `mx.core.MovieClipAsset`.

```
package {
  import mx.core.MovieClipAsset;

  [Embed(source="fp8app.swf", symbol="Ball")]
  public class FP8Ball extends MovieClipAsset {
  }
}
```

Чтобы создать новый экземпляр встроенного символа, мы используем следующий код (независимо от того, был символ встроен на уровне переменной или на уровне класса):

```
new FP8Ball( )
```

Если символ был встроен на уровне переменной, то при присваивании экземпляра этого символа переменной в качестве ее типа данных мы выбираем класс элемента из пространства имен `mx.core`, соответствующий типу символа (то есть один из классов `MovieClipAsset`, `TextFieldAsset` или `ButtonAsset`). Например, наш символ `Ball` является клипом, поэтому экземпляры класса `FP8Ball` должны присваиваться переменным типа `MovieClipAsset`.

```
var fp8ball:MovieClipAsset = new FP8Ball( );
```

Если символ был встроен на уровне класса, то в качестве типа данных этой переменной мы выбираем класс, который использовался для встраивания символа. Например:

```
var fp8ball:FP8Ball = new FP8Ball( );
```

Созданный экземпляр, как и любой другой отображаемый объект, может быть добавлен в список отображения:

```
addChild(fp8ball);
```

Встраивание файлов в виде бинарных данных

Тег метаданных `[Embed]` может применяться для встраивания в приложение бинарных данных (байтов) из любого файла в виде массива байтов. В дальнейшем приложение может работать с этими байтами. Например, если встраиваемые бинарные данные представляют файл в формате GIF, JPEG, PNG или SWF, то приложение может использовать класс `Loader` для преобразования этих данных в отображаемый элемент.

В следующем коде мы встраиваем SWF-файл с именем `fp9app.swf`, имеющий формат приложения Flash Player 9, в виде бинарных данных на уровне переменной. Этот код предполагает, что файл класса, из которого происходит встраивание бинарных данных, и файл, содержащий эти данные, находятся в одной директории. При выполнении этого кода среда Flash автоматически генерирует класс, представляющий бинарные данные, и присваивает этот класс переменной `FP9BinaryData`, позволяя использовать данные на этапе выполнения. Автоматически сгенерированный класс

расширяет класс `mx.core.ByteArrayAsset`. В приведенном коде обратите внимание на использование дополнительного параметра `mimeType` тега `[Embed]`:

```
[Embed(source="fp9app.swf", mimeType="application/octet-stream")]
private var FP9BinaryData:Class;
```

В следующем коде мы встраиваем SWF-файл с именем `fp9app.swf`, имеющий формат приложения Flash Player 9, в виде бинарных данных на уровне класса. Снова предполагается, что файл класса и файл элемента находятся в одной директории. Обратите внимание, что по необходимости этот класс расширяет класс `mx.core.ByteArrayAsset`.

```
package {
  import mx.core.ByteArrayAsset;

  [Embed(source="fp9app.swf", mimeType="application/octet-stream")]
  public class FP9BinaryData extends ByteArrayAsset {
  }
}
```

Чтобы создать новый экземпляр встроенных бинарных данных, мы используем следующий код (независимо от того, были данные встроены на уровне переменной или на уровне класса):

```
new FP9BinaryData( )
```

Присваивая экземпляр встроенных бинарных данных переменной, в качестве типа данных этой переменной мы выбираем либо класс `mx.core.ByteArrayAsset` (для элементов, встраиваемых на уровне переменной), либо класс, который применялся для встраивания символа (для элементов, встраиваемых на уровне класса):

```
var fp9binarydata:ByteArrayAsset = new FP9BinaryData( );  // Уровень
                                                          // переменной
var fp9binarydata:FP9BinaryData = new FP9BinaryData( );   // Уровень класса
```

Если встроенные бинарные данные представляют файл в формате GIF, JPEG, PNG или SWF, то после создания экземпляра этих данных мы можем использовать класс `Loader` для генерации отображаемого элемента, как показано в следующем коде:

```
var loader:Loader = new Loader( );
loader.loadBytes(fp9binarydata);
addChild(loader);
```

После инициализации элемент может быть добавлен в список отображения с помощью методик, рассмотренных ранее в подразд. «Отображение загруженного элемента на экране» разд. «Использование класса Loader для загрузки отображаемых элементов на этапе выполнения».

Методика встраивания элемента в виде бинарных данных может использоваться для встраивания в приложение файлов XML на этапе компиляции. Это демонстрирует листинг 28.14.

Листинг 28.14. Встраивание XML-файла на этапе компиляции

```
package {
  import flash.display.*;
```

```
  import flash.events.*;
  import flash.utils.ByteArray;

  public class EmbedXML extends Sprite {
    [Embed(source="embeds/data.xml", mimeType="application/octet-stream")]
    private var BinaryData:Class;

    public function EmbedXML ( ) {
      // Создаем новый экземпляр встроенных данных
      var byteArray:ByteArray = new BinaryData( );

      // Преобразуем экземпляр данных в XML-файл
      var data:XML = new XML(byteArray.readUTFBytes(byteArray.length));

      // Отображаем исходный код для встроенного XML-файла
      trace(data.toXMLString( ));
    }
  }
}
```

Использование метода getDefinition( ) для обращения к классу во встроенном SWF-файле

Как мы уже знаем, отдельные символы и классы из SWF-файла в формате приложения Flash Player 9 (или более поздних версий) не могут быть встроены с помощью тега метаданных `[Embed]`. Вместо этого, чтобы обратиться к классу или классу символа во встроенном SWF-файле, мы можем создать ссылку на SWC-файл, содержащий желаемый класс или класс символа, либо использовать метод экземпляра `getDefinition( )` класса `ApplicationDomain` для обращения к желаемому классу или классу символа на этапе выполнения.

При использовании метода `getDefinition( )` перед попыткой обращения к символам или классам SWF-файла мы должны убедиться, что его экземпляр был проинициализирован. Для этого мы регистрируем приемник в объекте `LoaderInfo` объекта встроенного элемента для событий `Event.INIT`. В зависимости от того, как происходит встраивание SWF-файла, мы обращаемся к объекту `LoaderInfo` различными способами. Если SWF-файл был встроен непосредственно на уровне переменной, то мы используем следующий код, чтобы зарегистрировать приемник для событий `Event.INIT`:

```
// Создаем экземпляр элемента
var экземплярВстроенногоЭлемента:MovieClipLoaderAsset = new ИмяКласса( );

// Регистрируем приемник для события Event.INIT
Loader(экземплярВстроенногоЭлемента.getChildAt(0)).contentLoaderInfo.
addEventListener(
          Event.INIT,
          приемникСобытияINIT);
```

В приведенном коде *ИмяКласса* — переменная, которая ссылается на класс, представляющий встроенный SWF-файл, *экземплярВстроенногоЭлемента* — экземпляр класса

ИмяКласса, а *приемникСобытияINIT* — ссылка на функцию, которая будет выполняться при инициализации экземпляра.

С другой стороны, если SWF-файл был встроен в виде бинарных данных, мы используем следующий код, чтобы зарегистрировать приемник для получения уведомлений о возникновении события `Event.INIT`:

```
// Создаем экземпляр элемента
var БинарныеДанные:ByteArrayAsset = new БинарныеДанные( );

// Генерируем отображаемый объект, представляющий SWF-файл
var loader:Loader = new Loader( );
loader.loadBytes(БинарныеДанные);
addChild(loader);

// Регистрируем приемник для события Event.INIT
loader.contentLoaderInfo.addEventListener(Event.INIT,
                                          приемникСобытияINIT);
```

Здесь *БинарныеДанные* — переменная, которая ссылается на класс, представляющий бинарные данные встроенного SWF-файла, *БинарныеДанные* — экземпляр класса *БинарныеДанные*, а *приемникСобытияINIT*, как и раньше, — ссылка на функцию, которая будет выполняться при инициализации экземпляра данного SWF-файла.

Следующий код демонстрирует пример приемника события `Event.INIT`, который получает ссылку на класс символа клипа с именем `Ball`. Этот код также создает экземпляр символа `Ball` и добавляет его в иерархию отображения класса метода `initListener( )`.

```
private function initListener (e:Event):void {
  // Получаем ссылку на символ Ball из встроенного SWF-файла
  var BallSymbol:Class =
  e.target.content.loaderInfo.applicationDomain.getDefinition("Ball");

  // Создаем новый экземпляр символа Ball
  var ball:MovieClip = MovieClip(new BallSymbol( ));

  // Помещаем экземпляр символа Ball на экран
  addChild(ball);
}
```

Пример использования тега [Embed]

Для справки в листинге 28.15 показан класс, который демонстрирует сценарии использования тега `[Embed]` на уровне переменной, рассмотренные в предыдущих разделах.

Листинг 28.15. Класс, демонстрирующий встраивание элементов на уровне переменной

```
package {
  import flash.display.*;
  import flash.events.*;
  import mx.core.MovieClipAsset;
  import mx.core.MovieClipLoaderAsset;
```

```
import mx.core.SpriteAsset;
import mx.core.BitmapAsset;
import mx.core.ByteArrayAsset;

public class VariableLevelEmbedDemo extends Sprite {
  [Embed(source="photo.jpg")]
  private var Photo:Class;

  [Embed(source="line.svg")]
  private var SVGLine:Class;

  [Embed(source="fp9app.swf")]
  private var FP9App:Class;

  [Embed(source="fp8app.swf", symbol="Ball")]
  private var FP8Ball:Class;

  [Embed(source="fp9app.swf", mimeType="application/octet-stream")]
  private var FP9BinaryData:Class;

  public function VariableLevelEmbedDemo ( ) {
    // Растровое изображение, встраиваемое
    // на уровне переменной
    var photo:BitmapAsset = new Photo( );
    addChild(photo);

    // Файл в формате SVG, встраиваемый на уровне переменной
    var line:SpriteAsset = new SVGLine( );
    addChild(line);

    // Символ SWF-файла в формате приложения Flash Player 8, встраиваемый
    // на уровне переменной
    var fp8ball:MovieClipAsset = new FP8Ball( );
    addChild(fp8ball);

    // SWF-файл в формате приложения Flash Player 9, встраиваемый
    // на уровне переменной
    var fp9app:MovieClipLoaderAsset = new FP9App( );
    addChild(fp9app);

    // Чтобы обратиться к классу символа или к обычному классу
    // во встроенном SWF-файле, необходимо дождаться завершения
    // инициализации данного SWF-файла
    Loader(fp9app.getChildAt(0)).contentLoaderInfo.addEventListener(
                                                    Event.INIT,
                                                    fp9appInitListener);

    // Бинарные данные (SWF-файл в формате приложения Flash Player 9),
    // встраиваемые на уровне переменной
    var fp9binarydata:ByteArrayAsset = new FP9BinaryData( );
    var loader:Loader = new Loader( );
```

```
        loader.loadBytes(fp9binarydata);
        addChild(loader);

        // Чтобы обратиться к классу символа или к обычному классу
        // во встроенном SWF-файле, необходимо дождаться завершения
        // инициализации данного SWF-файла
        loader.contentLoaderInfo.addEventListener(Event.INIT,
                                                  fp9binarydataInitListener);
    }

    private function fp9appInitListener (e:Event):void {
        // Получаем ссылку на символ Ball из встроенного SWF-файла
        var BallSymbol:Class =
          e.target.content.loaderInfo.applicationDomain.getDefinition("Ball");
        // Создаем новый экземпляр символа Ball
        var ball:MovieClip = MovieClip(new BallSymbol( ));
        // Устанавливаем положение экземпляра символа Ball и помещаем его
        // на экран
        ball.x = 220;
        ball.y = 240;
        addChild(ball);
    }

    private function fp9binarydataInitListener (e:Event):void {
        // Получаем ссылку на символ Ball из встроенного SWF-файла
        var BallSymbol:Class =
          e.target.content.loaderInfo.applicationDomain.getDefinition("Ball");
        // Создаем новый экземпляр символа Ball
        var ball:MovieClip = MovieClip(new BallSymbol( ));
        // Устанавливаем положение экземпляра символа Ball и помещаем его
        // на экран
        ball.y = 200;
        addChild(ball);
    }
  }
}
```

Очищайте проект, чтобы увидеть изменения

Мы рассмотрели несколько способов, позволяющих встроить внешний отображаемый элемент в приложение. Обычно, когда файл встраиваемого элемента изменяется, эти изменения автоматически отражаются при следующей компиляции связанного приложения. Однако в приложении Flex Builder 2 при повторной компиляции приложения изменения в элементах иногда могут не отражаться. Чтобы гарантировать, что все изменения в элементе будут отражены на этапе компиляции, очистите проект приложения, как описано ниже.

1. Выберите команду меню **Project ▸ Clean** (Проект ▸ Очистить).
2. В окне **Clean** (Очистка) выберите проект, который вы желаете очистить.
3. Нажмите кнопку **OK**.

К части III

В предыдущих 28 главах мы рассмотрели огромное количество основополагающих вопросов, связанных с программированием на языке ActionScript, и наше путешествие почти подошло к концу! В последней части этой книги мы рассмотрим три прикладных вопроса программирования на языке ActionScript: программирование в среде разработки Flash, использование платформы разработки Flex «исключительно» через язык MXML и распространение группы классов для применения в родительском приложении.

Практическое применение ActionScript

В части III основное внимание уделяется вопросам применения кода, написанного на языке ActionScript. В этой части освещаются такие темы, как объединение кода на ActionScript с элементами, созданными вручную в среде разработки Flash, использование платформы разработки Flex в приложении Flex Builder 2 и создание библиотеки пользовательского кода.

Завершив изучение части III, вы приобретете практические навыки, необходимые для разработки приложений на языке ActionScript с помощью среды разработки Flash и приложения Flex Builder 2. Кроме того, вы узнаете о способах распространения кода между другими разработчиками или по всему миру.

- Глава 29 «Язык ActionScript и среда разработки Flash».
- Глава 30 «Минимальное приложение на языке MXML».
- Глава 31 «Распространение библиотеки классов».

Язык ActionScript и среда разработки Flash

В гл. 1 было рассказано, что среда разработки Flash может быть использована для связывания кода на языке ActionScript с изображениями, анимацией и мультимедийными элементами. Теперь, когда мы хорошо знакомы с основами языка ActionScript, рассмотрим важные связи между ActionScript и содержимым, создаваемым в среде разработки Flash.

Чтобы помочь программистам, приходящим из других языков и сред, некоторые концепции, представленные в этой главе, рассматриваются с точки зрения новичка в среде разработки Flash. Таким образом, часть представленного материала будет являться своего рода обзором для опытных пользователей среды разработки Flash.

Файлы примеров, рассматриваемых в этой главе, вы можете загрузить по адресу http://www.moock.org/eas3/examples.

Документ Flash

В процессе изучения данной книги мы создали множество SWF-файлов, используя «настоящий код» (то есть один или несколько классов языка ActionScript). В отличие от этого, в среде разработки Flash SWF-файлы создаются с помощью графического интерфейса, предназначенного для создания изображений, анимации и интерактивного мультимедийного содержимого.

Чтобы создать SWF-файл в среде разработки Flash, мы должны сначала сформировать *документ Flash*, или FLA-файл. Он описывает размещение тела мультимедийного содержимого во времени. Чтобы создать FLA-файл, пригодный для использования с кодом на языке ActionScript 3.0, выполните следующие шаги.

1. В среде разработки Flash выберите команду меню File ▸ New (Файл ▸ Создать).
2. На вкладке General (Общие) окна New Document (Новый документ) в списке Type (Тип) выберите значение Flash File (ActionScript 3.0) (Файл Flash (ActionScript 3.0)).
3. Нажмите кнопку OK.

Из FLA-файла мы можем скомпилировать (или экспортировать) соответствующий SWF-файл для дальнейшего воспроизведения в среде выполнения Flash.

Чтобы экспортировать SWF-файл для отладки в среде разработки Flash, мы используем команду меню Control ▸ Test Movie (Управление ▸ Проверка фильма). Чтобы

экспортировать SWF-файл для последующего распространения в Интернете, мы используем команду меню File ▸ Publish (Файл ▸ Опубликовать). SWF-файл, экспортированный из среды разработки Flash, может также распространяться в виде настольного приложения; дополнительную информацию можно найти в документации по приложению Adobe AIR.

Временные шкалы и кадры

Структурно FLA-файл представлен в виде иерархии анимаций, каждая из которых имеет собственную *временну́ю шкалу*. Временная шкала — это линейная последовательность *кадров*, похожая на обычный диафильм. Каждый кадр может содержать аудио, видео, текст, векторную и растровую графику, а также содержимое, создаваемое программным путем.

При экспортировании SWF-файла из FLA-файла компилятор Flash конвертирует временные шкалы FLA-файла в файловый формат Flash (SWF), подходящий для воспроизведения в клиентской среде выполнения Flash. При воспроизведении SWF-файла кадры из временных шкал FLA-файла отображаются на экране, быстро сменяя друг друга, создавая тем самым эффект анимации. Скорость, с которой в среде выполнения Flash происходит смена кадров, обусловлена *скоростью кадров*, измеряемой в кадрах в секунду (дополнительную информацию по скорости кадров можно найти в гл. 23).

Хотя некоторые FLA-файлы включают всего одну временную шкалу, большинство из них содержит несколько временных шкал, позволяя создавать содержимое отдельными частями и объединять их вместе, формируя более сложную анимацию. Например, сцена, изображающая движение автомобиля вниз по горной дороге, может включать три временные шкалы: одну для медленного перемещения гор, другую для быстрого перемещения дороги и третью для вращения колес автомобиля.

Первая временная шкала, создаваемая в новом FLA-файле, называется *основной временной шкалой*. Она формирует фундамент для всего содержимого, добавляемого в FLA-файл.

Для создания каждого кадра содержимого на временной шкале мы можем либо импортировать внешние элементы, либо создавать новую графику с помощью встроенных инструментов рисования среды разработки Flash (например, Pencil (Карандаш), Brush (Кисть) и Text (Текст)). Графика каждого кадра помещается в визуальное рабочее пространство, называемое *сценой* (Stage).

Не путайте сцену среды разработки Flash с классом Stage языка ActionScript: сцена среды разработки Flash является рабочим пространством для создания содержимого, а класс Stage языка ActionScript представляет корневой элемент в списке отображения среды выполнения Flash.

На рис. 29.1 изображен простой FLA-файл `hello.fla`, открытый в среде разработки Flash. В верхней половине рисунка показана основная временная шкала FLA-файла, которая в данном случае содержит два кадра. В нижней части рисунка показана

сцена, которая отображает содержимое выбранного кадра. В данном случае выбранный кадр является первым кадром основной временной шкалы, обозначаемым тонкой вертикальной линией, проходящей через кадр на временной шкале. Эта тонкая линия обозначает выбранный кадр и называется *головкой воспроизведения*.

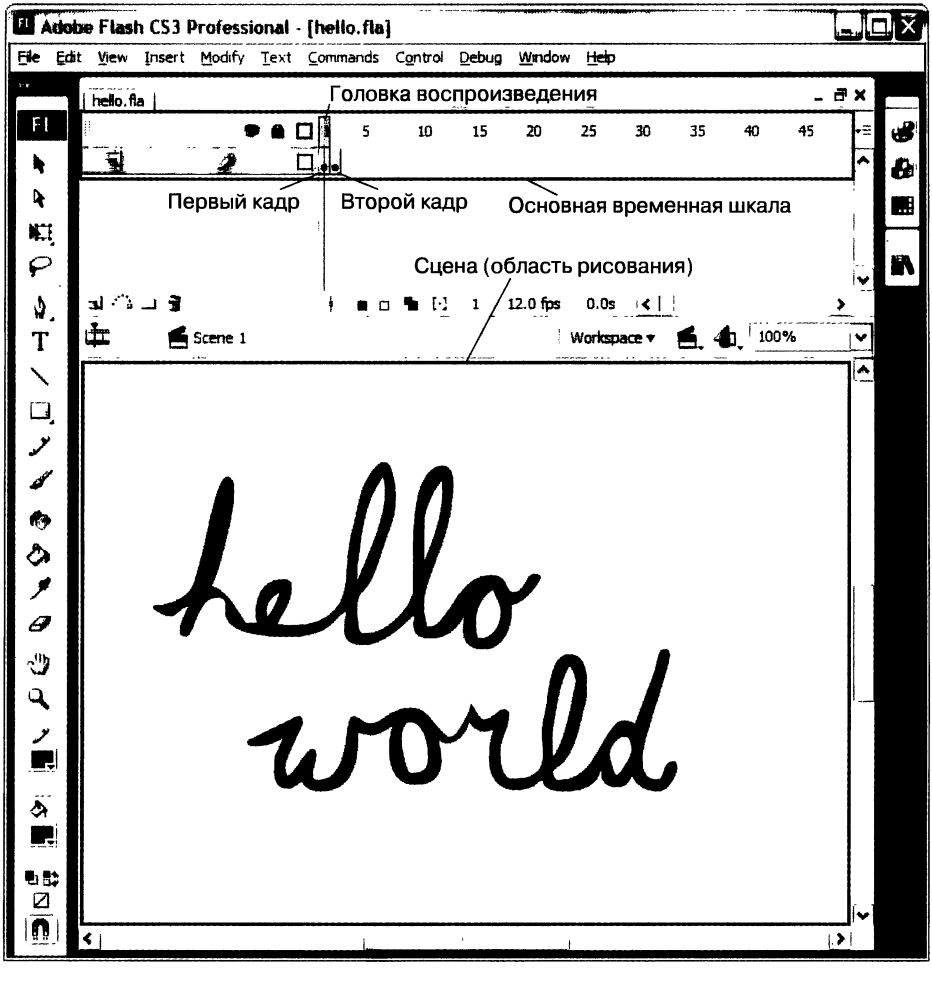

Рис. 29.1. Файл hello.fla, показан кадр 1 основной временной шкалы

На рис. 29.2 снова изображен файл `hello.fla`, но на этот раз на основной временной шкале выбран кадр 2. Обратите внимание, что положение головки воспроизведения, обозначающей выбранный кадр, сместилось вправо. Соответственно, на сцене сейчас отображается содержимое кадра 2.

Ключевые и обычные кадры. В среде разработки Flash определены два типа кадров: *ключевые* и *обычные*. Хотя рассмотрение анимации Flash выходит за рамки данной книги, понимание различий между ключевыми и обычными кадрами является неотъемлемым условием для разработки сценариев, рассматриваемых в следующем разделе.

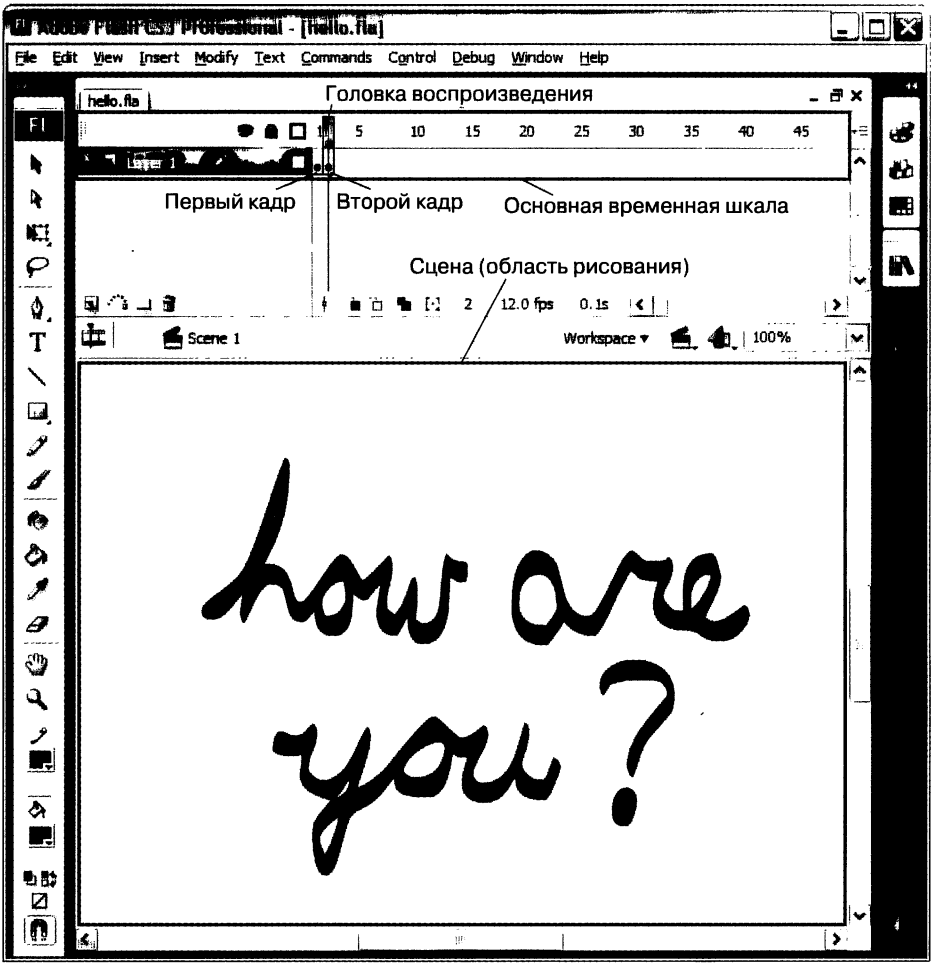

Рис. 29.2. Файл hello.fla, показан кадр 2 основной временной шкалы

Ключевой кадр — это кадр, у которого находящееся на сцене содержимое отличается от содержимого предыдущего кадра. Чтобы добавить новый ключевой кадр на временную шкалу, мы используем команду меню Insert ▸ Timeline ▸ Keyframe (Вставка ▸ Временная шкала ▸ Ключевой кадр).

В отличие от этого, *обычный кадр* — это кадр, у которого находящееся на сцене содержимое автоматически образуется (повторяется) из содержимого предыдущего ближайшего ключевого кадра. Чтобы добавить новый обычный кадр на временную шкалу, мы используем команду меню Insert ▸ Timeline ▸ Frame (Вставка ▸ Временная шкала ▸ Кадр).

Добавляя обычные кадры на временную шкалу, мы увеличиваем время, в течение которого изображение на экране остается неизменным. Добавляя ключевые кадры на временную шкалу, мы можем изменить содержимое, отображаемое на экране (обычно для создания эффекта анимации).

Предположим, что мы создаем анимацию, в которой человечек бежит к дому, на мгновение останавливается перед дверью, а затем убегает прочь. Когда человечек бежит к дому, каждый кадр анимации имеет различное содержимое, поэтому все соответствующие кадры должны быть ключевыми. Когда человечек находится перед дверью, каждый кадр анимации имеет одинаковое содержимое (поскольку человечек не двигается), поэтому все соответствующие кадры могут быть обычными. Когда человечек убегает прочь от дома, каждый кадр анимации снова имеет различное содержимое, поэтому все соответствующие кадры должны быть ключевыми.

На рис. 29.3 показаны кадры нашей анимации с бегущим человечком, которую пришлось сильно сократить, чтобы уместить в данной книге. Кадры 1, 2, 3, 8 и 9 являются ключевыми, каждый из которых имеет свое собственное содержимое. Кадры 4–7 являются обычными, содержимое которых переносится из кадра 3. На рисунке содержимое кадров 4–7 изображено серым цветом — это значит, что оно переносится из кадра 3. На рисунке ключевые кадры отмечены буквой K, а обычные кадры — буквой R.

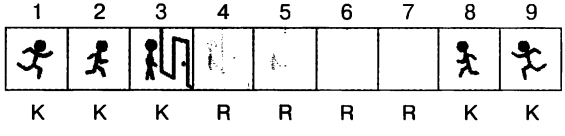

Рис. 29.3. Ключевые кадры в сравнении с обычными

В среде разработки Flash временная шкала для нашей анимации с выбранным кадром 1 выглядела бы так, как показано на рис. 29.4. Обратите внимание, что

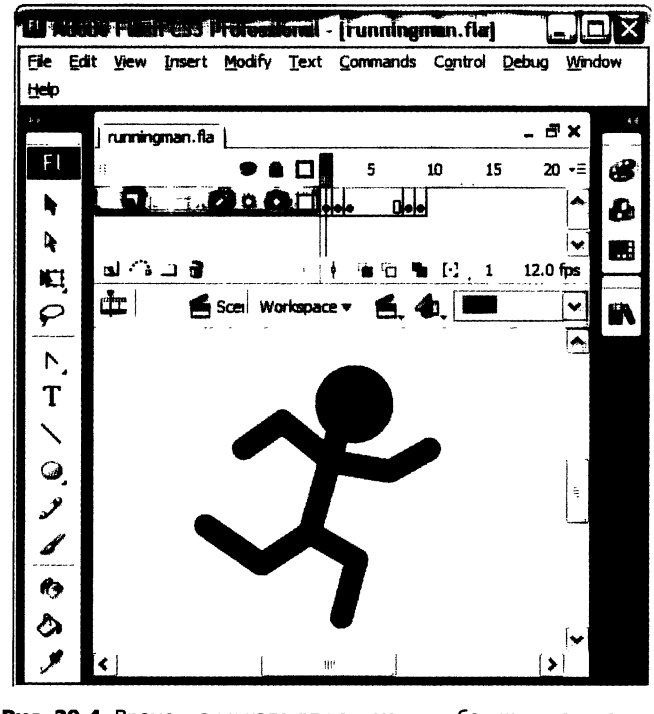

Рис. 29.4. Временная шкала для анимации с бегущим человечком

ключевые кадры с содержимым обозначаются значком заполненного круга. Ключевые кадры без содержимого, называемые *пустыми ключевыми кадрами*, обозначаются пустым кругом (наша анимация не имеет пустых ключевых кадров). Для обозначения обычных кадров значок круга не используется вообще. Последний обычный кадр, предшествующий ключевому кадру, обозначается значком прямоугольника.

Чтобы упростить изучение материала в последующих разделах, мы будем полагать, что наша анимация с бегущим человечком сохраняется в файле с именем `runningman.fla`.

Теперь рассмотрим процесс создания сценариев на временной шкале.

Создание сценариев на временной шкале

Чтобы выполнить код в определенной точке на временной шкале, мы используем *сценарий кадра*. Он представляет собой блок кода на языке ActionScript, добавленный к ключевому кадру на временной шкале FLA-файла.

Чтобы добавить сценарий к ключевому кадру на временной шкале, используйте следующую основную последовательность действий.

1. Щелкните кнопкой мыши на нужном ключевом кадре на временной шкале (выберите его).
2. Выберите команду меню Window ▸ Actions (Окно ▸ Действия), чтобы открыть панель Actions (Действия).
3. Введите желаемый код на палитре Actions (Действия).

Любой код, введенный на палитре Actions (Действия) при выбранном ключевом кадре, становится частью сценария этого кадра и будет выполняться непосредственно перед его отображением в среде выполнения Flash.

Сценарии обычно используются для управления содержимым кадра, в котором они находятся. Мы узнаем подробнее об управлении содержимым с помощью сценариев кадров далее, в разд. «Обращение к созданным вручную экземплярам символов» этой главы.

Сценарии кадров не могут содержать инструкции class, package или interface либо атрибуты public, internal, private, dynamic или static, но могут включать любой другой код на языке ActionScript.

Программирование с помощью сценариев кадров иногда называют *созданием сценариев на временной шкале*. Это старейший вид программирования на ActionScript, а до появления версии языка 2.0 и приложения Flash Player 7 это был *единственный* вид программирования на ActionScript. Возможно, часть наиболее новаторских разработок сообщества Flash появилась в результате соединения содержимого временной шкалы и сценариев кадров.

Чтобы привести пример сценария кадра, добавим блок кода в последний кадр анимации с бегущим человечком. В нашем примере сценария кадра будет открываться URL-адрес http://moock.org в новом окне браузера. Вот этот код:

```
import flash.net.*;
var request:URLRequest = new URLRequest("http://moock.org");
navigateToURL(request, "_blank");
```

На палитре **Actions** (Действия) среды разработки Flash наш сценарий кадра будет выглядеть так, как показано на рис. 29.5. Обратите внимание, что на временной шкале ключевые кадры со сценариями кадров обозначаются маленьким значком «a» (первая буква названия языка ActionScript).

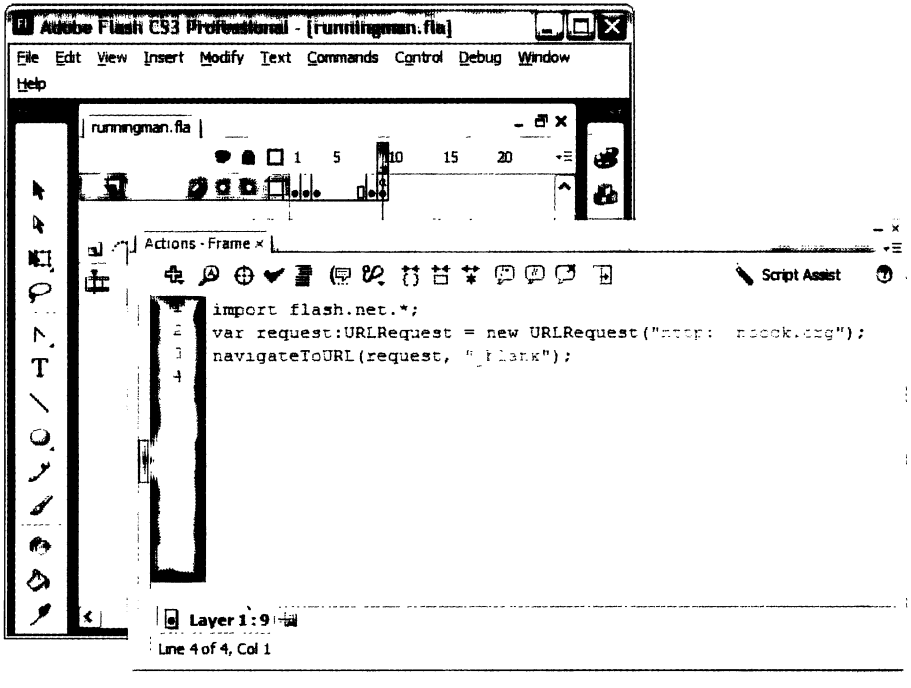

Рис. 29.5. Сценарий кадра

В существующем виде наш сценарий имеет одну проблему. По умолчанию среда выполнения Flash зацикливает анимации (то есть воспроизводит их повторно). Таким образом, наш сценарий будет выполняться всякий раз при завершении очередного цикла воспроизведения анимации с бегущим человечком, приводя к открытию множества окон браузера. Мы должны использовать метод экземпляра `stop( )` класса `MovieClip` для предотвращения повторного воспроизведения анимации, как показано в следующем коде. С методами класса `MovieClip`, предназначенными для управления воспроизведением, мы подробно познакомимся далее, в разд. «Программное управление временной шкалой».

```
import flash.net.*;
var request:URLRequest = new URLRequest("http://moock.org");
navigateToURL(request, "_blank");
stop( );
```

В предыдущем коде метод `stop( )` вызывается над объектом языка ActionScript, представляющим основную временную шкалу. В следующем разделе подробно

описывается представление основной временной шкалы в виде объекта в языке ActionScript.

Стоит отметить, что сценарии кадров могут ссылаться на любой пользовательский класс (или другое определение), доступный через путь к исходным классам документа (устанавливаемый для каждого FLA-файла) или через глобальный путь к классам (применяемый ко всем FLA-файлам). Чтобы задать путь к классам документа, выберите команду меню File ▸ Publish Settings (Файл ▸ Настройки публикации), нажмите кнопку Settings (Параметры), расположенную возле параметра ActionScript version (Версия ActionScript) с установленным значением ActionScript 3.0 на вкладке Flash, и укажите необходимые значения в списке Classpath (Путь к классам). Чтобы задать глобальный путь к классам, выберите команду меню Edit ▸ Preferences (Правка ▸ Предпочтения), затем выберите категорию ActionScript, нажмите кнопку ActionScript 3.0 Settings (Параметры ActionScript 3.0) в области Language (Язык) и укажите необходимые значения в списке Classpath (Путь к классам).

Класс документа

С точки зрения ActionScript основная временная шкала FLA-файла считается экземпляром *класса документа*, указать который можно следующим образом: выберите команду меню File ▸ Publish Settings (Файл ▸ Настройки публикации), нажмите кнопку Settings (Параметры), расположенную возле параметра ActionScript version (Версия ActionScript) на вкладке Flash, и введите необходимое значение в поле ввода Document class (Класс документа).

Этот класс документа должен наследоваться от класса `flash.display.MovieClip`, если выполняются следующие условия.

- Основная временная шкала содержит сценарии кадров.
- Класс документа желает управлять основной временной шкалой программным путем с помощью методов класса `MovieClip`.
- Сцена основной временной шкалы содержит компоненты с настраиваемыми параметрами и справедливо любое из следующих утверждений.
  - Значения настраиваемых параметров не являются одинаковыми во всех кадрах временной шкалы. Например, символ Button (Кнопка) в кадре 1 имеет название `"OK"`, а в кадре 2 — `"Submit"`.
  - Компонент присутствует не во всех кадрах временной шкалы. Например, символ List (Список) с настраиваемым поставщиком данных присутствует в кадре 1, но отсутствует в кадре 2.
- Сцена основной временной шкалы содержит компоненты с настраиваемыми свойствами доступности или значения с палитры Strings (Строки).

В противном случае класс документа должен наследоваться только от класса `flash.display.Sprite`.

При указании класса документа для FLA-файла используйте только полностью уточненное имя класса; не включайте расширение файла (AS).

Если класс, указанный в качестве класса документа, не будет найден, то компилятор Flash автоматически сгенерирует класс документа с указанным именем. Автоматически генерируемый класс расширяет класс `MovieClip`.

Если для FLA-файла не был указан класс документа, то он устанавливается автоматически. Если выполняются все следующие условия, то автоматически устанавливаемым классом документа становится класс `flash.display.MovieClip`.

- Сцена основной временной шкалы не содержит именованных экземпляров (дополнительную информацию можно найти в разд. «Обращение к созданным вручную экземплярам символов»).
- Основная временная шкала не содержит сценариев кадров.
- Сцена основной временной шкалы не содержит компонентов с настраиваемыми параметрами, значения которых изменяются от кадра к кадру.
- Сцена основной временной шкалы не содержит компонентов с настраиваемыми параметрами доступности или значений с палитры **Strings** (Строки).

В противном случае автоматически устанавливаемым классом документа будет автоматически генерируемый подкласс класса `MovieClip`.

Любой сценарий кадра основной временной шкалы можно считать далеким аналогом метода экземпляра класса документа. Код сценария кадра основной временной шкалы выполняется в своей собственной области действия и использует такой же набор правил доступа, который применяется к методам экземпляра класса документа. Иными словами, сценарий кадра основной временной шкалы может обратиться к любому определению (переменная, метод, класс, интерфейс или пространство имен), доступному из любого метода экземпляра класса документа. Подобным образом сценарий кадра основной временной шкалы может использовать ключевое слово `this`, чтобы обратиться к текущему объекту (то есть к экземпляру класса документа).

Для примера создадим класс документа `RunningMan` для файла `runningman.fla` из предыдущего раздела. Поскольку основная временная шкала файла `runningman.fla` включает сценарий кадра, класс `RunningMan` должен наследоваться от класса `MovieClip`. В классе `RunningMan` мы определим простой метод для открытия URL-адреса в браузере.

```
package {
  import flash.display.MovieClip;
  import flash.net.*;

  public class RunningMan extends MovieClip {
    public function goToSite (url:String):void {
      var request:URLRequest = new URLRequest(url);
      navigateToURL(request, "_blank");
    }
  }
}
```

Чтобы присоединить класс `RunningMan` к нашей анимации с бегущим человечком, мы сохраняем исходный файл этого класса в той же папке, где находится файл `runningman.fla`, а затем выполняем следующие шаги.

1. Выбираем команду меню File ▸ Publish Settings (Файл ▸ Настройки публикации) и нажимаем кнопку Settings (Параметры), расположенную возле параметра ActionScript version (Версия ActionScript) с установленным значением ActionScript 3.0 на вкладке Flash.
2. В поле Document class (Класс документа) вводим значение `RunningMan` (обратите внимание: `RunningMan`, а не `RunningMan.as`).
3. В окне ActionScript 3.0 Settings (Параметры ActionScript 3.0) нажимаем кнопку OK.
4. В окне Publish Settings (Настройки публикации) нажимаем кнопку OK.

Теперь, когда в качестве класса документа для файла `runningman.fla` установлен класс `RunningMan`, мы можем обновить сценарий для кадра 9 из предыдущего раздела. Предыдущий код сценария кадра выглядел так:

```
import flash.net.*;
var request:URLRequest = new URLRequest("http://moock.org");
navigateToURL(request, "_blank");
stop( );
```

Следующий код демонстрирует новый сценарий для кадра 9. Стоит отметить, что в этом сценарии происходит непосредственное обращение к методу экземпляра `goToSite( )` класса `RunningMan`.

```
goToSite("http://moock.org");
stop( );
```

В предыдущем коде выражение `stop( )` ссылается на метод экземпляра `stop( )` класса `MovieClip`, от которого наследуется класс `RunningMan`. Являясь потомком класса `MovieClip`, `RunningMan` также имеет доступ ко всем незакрытым методам и переменным, определенным в классах `EventDispatcher`, `DisplayObject`, `InteractiveObject`, `DisplayObjectContainer` и `Sprite`. Например, следующий код использует метод экземпляра `addChild( )` класса `DisplayObjectContainer`, чтобы добавить новое текстовое поле в иерархию отображения основной временной шкалы:

```
// Код сценария для кадра 9 файла runningman.fla
import flash.text.*;

// Добавляем новое текстовое поле в иерархию
// отображения основной временной шкалы
var msg:TextField = new TextField( );
msg.text = "I guess no one was home...";
msg.autoSize = TextFieldAutoSize.LEFT;
msg.border     = true;
msg.background = true;
msg.selectable = false;
addChild(msg);

stop( );
```

В результате выполнения предыдущего кода объект `TextField`, на который ссылается переменная `msg`, добавляется в иерархию отображения основной временной шкалы. При экспортировании файла `runningman.swf` из файла `runningman.fla`

и его последующей загрузке в среду выполнения Flash текст «I guess no one was home...» появится на экране, когда головка воспроизведения достигнет кадра 9. Отображение текста на экране происходит благодаря тому, что экземпляр класса документа первого SWF-файла, загруженного в среду выполнения, автоматически добавляется в список отображения. Дополнительную информацию можно найти в гл. 20.

Стоит отметить, что предыдущий код, создающий текстовое поле, в качестве альтернативы (и этот вариант является более предпочтительным) мог быть добавлен в метод экземпляра класса `RunningMan`. С технической точки зрения любой подход является допустимым, однако лучшая практика заключается в сохранении кода во внешних файлах классов, а не в сценариях кадров.

Определения переменных и функций в сценариях кадров. Если определение переменной размещается в сценарии кадра на основной временной шкале FLA-файла, то в классе документа этого FLA-файла создается соответствующая переменная экземпляра. Подобным образом, если определение функции размещается в сценарии кадра на основной временной шкале FLA-файла, в классе документа этого FLA-файла создается соответствующий метод экземпляра.

Аналогично, если определение переменной размещается в сценарии кадра на временной шкале символа Movie Clip (Клип), в классе, связанном с этим символом, создается соответствующая переменная экземпляра. И если в сценарии кадра на временной шкале символа Movie Clip (Клип) размещается определение функции, то в классе, связанном с этим символом, создается соответствующий метод экземпляра (о символах Movie Clip (Клип) и связываемых классах будет рассказано в двух следующих разделах).

Тем не менее обратите внимание, что переменная экземпляра, созданная в сценарии кадра, будет проинициализирована только в момент выполнения этого сценария (то есть когда головка воспроизведения достигнет кадра, содержащего данный сценарий). Например, рассмотрим следующий код, демонстрирующий два сценария кадра:

```
// Сценарий кадра 1
trace(n);  // Выводит: 0

// Сценарий кадра 2
var n:int = 10;
trace(n);  // Выводит: 10
```

Когда выполняется первый из двух предыдущих сценариев кадра, переменная экземпляра `n` уже определена, но еще не проинициализирована (часть кода `= 10` пока не выполнялась). В результате код `trace(n)` выведет значение `0` (значение по умолчанию для переменных типа `int`) в окне Output (Вывод). Когда выполняется второй сценарий, переменная экземпляра `n` *уже* проинициализирована (присвоено значение `10`), поэтому код `trace(n)` выведет значение `10` в окне Output (Вывод).

В отличие от этого, когда выполнение сценария кадра завершено, любые переменные экземпляра, определенные в этом сценарии, могут использоваться вплоть до

завершения программы. Предположим, что мы добавили третий сценарий кадра на гипотетическую временную шкалу из предыдущего кода:

```
// Сценарий кадра 3
trace(n);  // Выводит: 10
gotoAndStop(1);
```

Когда выполняется третий сценарий, переменная экземпляра n по-прежнему имеет значение 10, поэтому код trace(n) выводит значение 10 в окне Output (Вывод). Затем код gotoAndStop(1) перемещает головку воспроизведения на кадр 1, приводя к выполнению сценария кадра 1 во второй раз. На этот раз переменная экземпляра n имеет значение 10, поэтому код trace(n) выводит значение 10 (а не 0) в окне Output (Вывод).

Метод экземпляра, созданный в сценарии кадра, может быть использован даже до того, как головка воспроизведения достигнет кадра, содержащего определение этого метода экземпляра.

Символы и экземпляры

Ранее из разд. «Временные шкалы и кадры» мы узнали, что FLA-файл представляет собой иерархию анимаций, каждая из которых имеет собственную *временную шкалу*. Теперь, когда мы знаем, как создавать основную анимацию в документе Flash, рассмотрим вложенные анимации. Для их создания мы должны разобраться с *символами*.

В среде разработки Flash символ — это определенная пользователем анимация, кнопка или изображение с возможностью повторного использования. Символы создаются за пределами сцены в специальном режиме редактирования. Каждый FLA-файл сохраняет связанные с ним символы в хранилище элементов, называемом *библиотекой* (Library). Из одного символа можно создать произвольное количество его копий, или *экземпляров*.

Например, чтобы создать небо, заполненное анимированными мерцающими звездами, мы могли бы создать один символ звезды и затем добавить множество его экземпляров в сцену. На рис. 29.6 показано, как могли бы выглядеть символ звезды и его экземпляры в FLA-файле. Обратите внимание, что позиция, масштаб и угол поворота каждого экземпляра звезды устанавливаются независимо.

По умолчанию, когда SWF-файл компилируется из FLA-файла, в SWF-файл включаются только те символы, экземпляры которых действительно используются в документе (то есть появляются в сцене) или экспортируются для языка ActionScript (обратитесь к разд. «Создание экземпляров символов среды разработки Flash из кода на языке ActionScript»). Следовательно, содержимое каждого символа включается всего один раз и затем на этапе выполнения он дублируется для каждого экземпляра по мере необходимости. В связи с этим использование символов и экземпляров (вместо обычных изображений) может в значительной степени уменьшить размер SWF-файла.

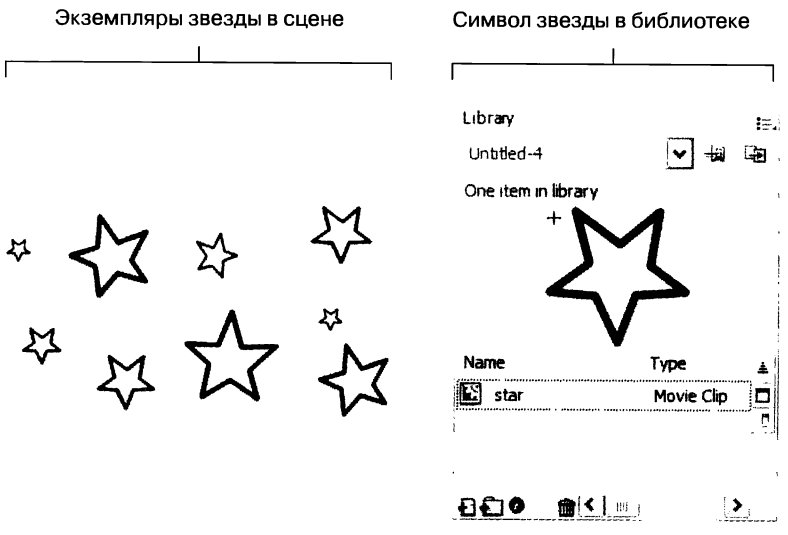

Рис. 29.6. Символ звезды и его экземпляры

Типы символов

В среде разработки Flash определены три основных типа символов:

- Movie Clip (Клип) (для анимаций с содержимым, создаваемым программным *путем);*
- Graphic (Графика) (для анимаций без содержимого, создаваемого программным путем, которые могут просматриваться непосредственно в среде разработки Flash);
- Button (Кнопка) (для реализации простой интерактивности).

К символам Graphic (Графика) нельзя обращаться из кода на языке ActionScript, а символы Button (Кнопка) предоставляют простейшие интерактивные возможности, которые также можно реализовать с помощью символа Movie Clip (Клип) или компонента `Button`. Таким образом, в этой главе мы опустим рассмотрение символов Graphic (Графика) и Button (Кнопка) и сосредоточимся исключительно на символах Movie Clip (Клип).

Символы Movie Clip (Клип)

Символ Movie Clip (Клип) — это самодостаточная анимация с возможностью повторного использования. Любой символ Movie Clip (Клип) имеет свою собственную временную шкалу и сцену, подобно основной временной шкале. Более того, экземпляры символов Movie Clip (Клип) могут быть вложены друг в друга для создания графики *или анимаций, имеющих иерархическую структуру. Например, символ Movie Clip* (Клип), представляющий нарисованное лицо, может содержать отдельный символ Movie Clip (Клип), представляющий анимированные моргающие глаза.

На рис. 29.7 изображен символ Movie Clip (Клип), созданный в среде разработки Flash. На сцене показано содержимое кадра 2 временной шкалы этого символа

Movie Clip (Клип). Обратите внимание, что временная шкала на рисунке не является основной временной шкалой FLA-файла, а представляет собой отдельную, независимую временную шкалу символа Movie Clip (Клип) звезды.

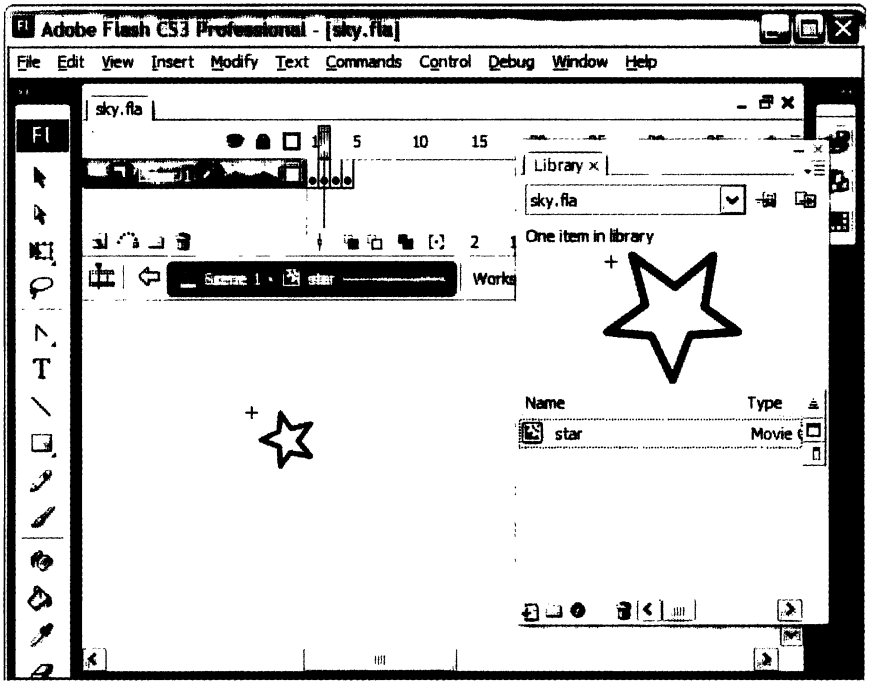

Рис. 29.7. Редактирование символа Movie Clip (Клип)

Связанные классы для символов Movie Clip (Клип)

С точки зрения языка ActionScript на этапе выполнения каждый экземпляр символа Movie Clip (Клип) в FLA-файле представляется экземпляром класса `Sprite` или одного из его подклассов. Класс, используемый для представления экземпляров определенного символа Movie Clip (Клип), называется *связанным классом* этого символа. Связанный класс символа может быть указан вручную или сгенерирован автоматически.

Чтобы указать связанный класс для символа Movie Clip (Клип), мы используем окно Linkage Properties (Свойства связывания). Обратите внимание, что если любое из следующих утверждений верно, то указываемый связанный класс должен наследоваться от класса `flash.display.MovieClip`.

- Временная шкала символа содержит сценарии кадров.
- Связанный класс желает управлять экземплярами символа программным путем, используя методы класса `MovieClip`.

- Сцена символа содержит компоненты с настраиваемыми параметрами, и справедливо любое из следующих утверждений.
  - Значения настраиваемых параметров не являются одинаковыми во всех кадрах временной шкалы. Например, символ Button (Кнопка) в кадре 1 имеет название "OK", а в кадре 2 — "Submit".
  - Компонент присутствует не во всех кадрах временной шкалы. Например, символ List (Список) с настраиваемым поставщиком данных присутствует в кадре 1, но отсутствует в кадре 2.
- Сцена основной временной шкалы содержит компоненты с настраиваемыми свойствами доступности или значения с палитры Strings (Строки).

В противном случае указываемый связанный класс должен наследоваться только от класса flash.display.Sprite.

Чтобы указать связанный класс для символа Movie Clip (Клип), выполняйте следующие шаги.

1. Выберите символ в библиотеке FLA-файла.
2. Откройте меню Options (Свойства) палитры Library (Библиотека), щелкнув кнопкой мыши на значке в правом верхнем углу палитры, и выберите команду Linkage (Связывание).
3. В области Linkage (Связывание) окна Linkage Properties (Свойства связывания) следует установить флажок Export for ActionScript (Экспорт для ActionScript). Стоит отметить, что при этом выбранный символ включается в скомпилированный SWF-файл независимо от того, используются его экземпляры в документе или нет.
4. В поле Class (Класс) окна Linkage Properties (Свойства связывания) введите полностью уточненное имя класса (то есть имя класса, объединенное с именем пакета класса, если класс размещается в пакете). Связываемый класс должен быть доступен либо через глобальный путь к классам, либо через путь к классам FLA-файла, содержащего этот символ. Указывая имя связанного класса в окне Linkage Properties (Свойства связывания), никогда не изменяйте значения по умолчанию для параметра Base class (Базовый класс), кроме тех случаев, когда с одним суперклассом связывается более одного символа, — эта ситуация рассматривается в разд. «Связывание нескольких символов с одним суперклассом».
5. Нажмите кнопку OK.

Если класс, указанный на шаге 4 описанной процедуры, не будет найден, то компилятор Flash автоматически сгенерирует связанный класс с указанным именем. Автоматически генерируемый класс расширяет указанный базовый класс.

Если для некоторого символа не был указан связанный класс, то компилятор устанавливает его автоматически. Если выполняются все следующие условия, то автоматически устанавливаемым связанным классом является класс MovieClip.

- Сцена символа не содержит именованных экземпляров.
- Временная шкала символа не содержит сценариев кадров.
- Сцена символа не содержит компонентов с настраиваемыми параметрами, значения которых изменяются от кадра к кадру.

- Сцена символа не содержит компонентов с настраиваемыми параметрами доступности или значений из палитры Strings (Строки).

В противном случае автоматически устанавливаемым связанным классом будет автоматически генерируемый подкласс класса `MovieClip`.

После завершения операции связывания символа с классом создаваемые вручную экземпляры этого символа будут перенимать программные поведения, определяемые связанным классом. С другой стороны, экземпляры, создаваемые программным путем, будут перенимать аудиовизуальное содержимое связанного символа. Таким образом, символ и класс являются одним целым: символ определяет графическое содержимое, а класс — программное поведение.

В качестве примера свяжем наш символ Movie Clip (Клип) звезды из предыдущего раздела с подклассом `Star` класса `MovieClip`. Класс `Star` будет случайным образом изменять прозрачность (значение канала Alpha) каждого экземпляра звезды через каждые 100 мс.

```
package {
  import flash.display.MovieClip;
  import flash.utils.Timer;
  import flash.events.TimerEvent;

  public class Star extends MovieClip {
    private var timer:Timer;

    public function Star ( ) {
      timer = new Timer(100, 0);
      timer.addEventListener(TimerEvent.TIMER, timerListener);
      timer.start( );
    }

    private function timerListener (e:TimerEvent):void {
      randomFade( );
    }

    private function randomFade ( ):void {
      // Присваиваем переменной alpha случайное значение с плавающей запятой
      // в диапазоне от 0 до (но не включая) 1. Переменная экземпляра alpha
      // наследуется от класса DisplayObject (являющегося предком класса
      // MovieClip).
      alpha = Math.random( );
    }

    // Предоставляет способ остановить таймер. Как рассматривалось
    // в разд. «Деактивация объектов» гл. 14, внешний код должен
    // использовать этот метод перед удалением экземпляра звезды
    // из программы.
    public function dispose ( ):void {
      timer.stop( );
    }
  }
```

Чтобы связать класс `Star` с символом Movie Clip (Клип) звезды, выполните следующие действия.

1. Сохраните файл, содержащий символ звезды, под именем `sky.fla`.
2. Сохраните класс `Star` в той же папке, где находится файл `sky.fla`, в текстовом файле под именем `Star.as`.
3. Выберите символ звезды в библиотеке файла `sky.fla`.
4. Откройте меню Options (Свойства) палитры Library (Библиотека), щелкнув кнопкой мыши на значке в правом верхнем углу палитры, и выберите команду Linkage (Связывание).
5. В области Linkage (Связывание) окна Linkage Properties (Свойства связывания) установите флажок Export for ActionScript (Экспорт для ActionScript).
6. В поле Class (Класс) окна Linkage Properties (Свойства связывания) введите значение `Star`.
7. Нажмите кнопку OK.

На рис. 29.8 показано окно Linkage Properties (Свойства связывания) в том виде, в котором оно находится после выполнения шага 6 предыдущей процедуры.

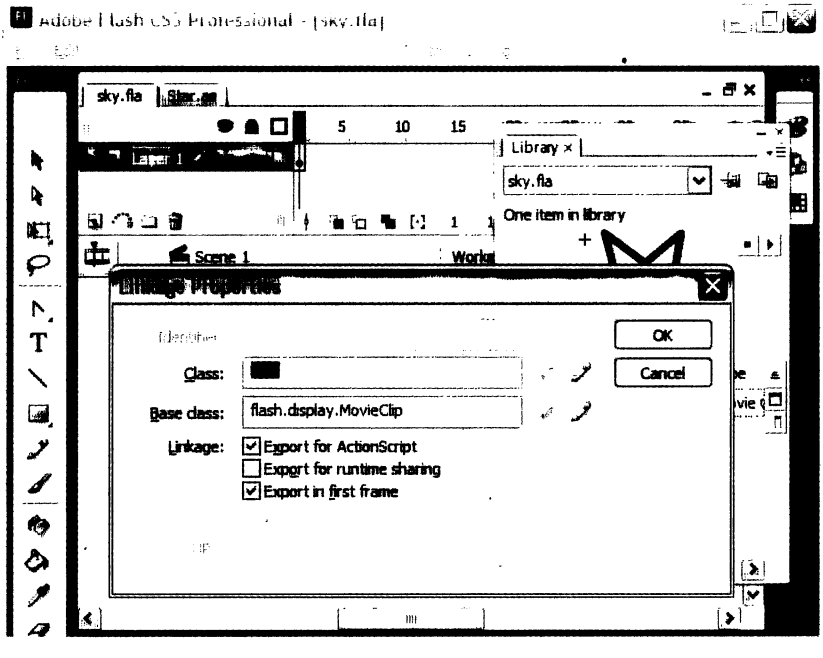

Рис. 29.8. Связывание символа Star с классом Star

Чтобы создать экземпляры символа звезды в среде разработки Flash, мы перетаскиваем название символа из библиотеки на сцену основной временной шкалы (или на сцену любого другого символа, но в данном случае у нас нет других символов в документе). Эта процедура проиллюстрирована на рис. 29.9, где изображены пять экземпляров символа звезды, перетаскиваемых на сцену первого кадра основной временной шкалы файла `sky.fla`.

Поместив экземпляры символа звезды на основную временную шкалу, мы можем экспортировать файл `sky.swf` из файла `sky.fla` и понаблюдать за эффектом мерцания (определяемым символом звезды) и эффектом затухания (определяемым классом `Star`).

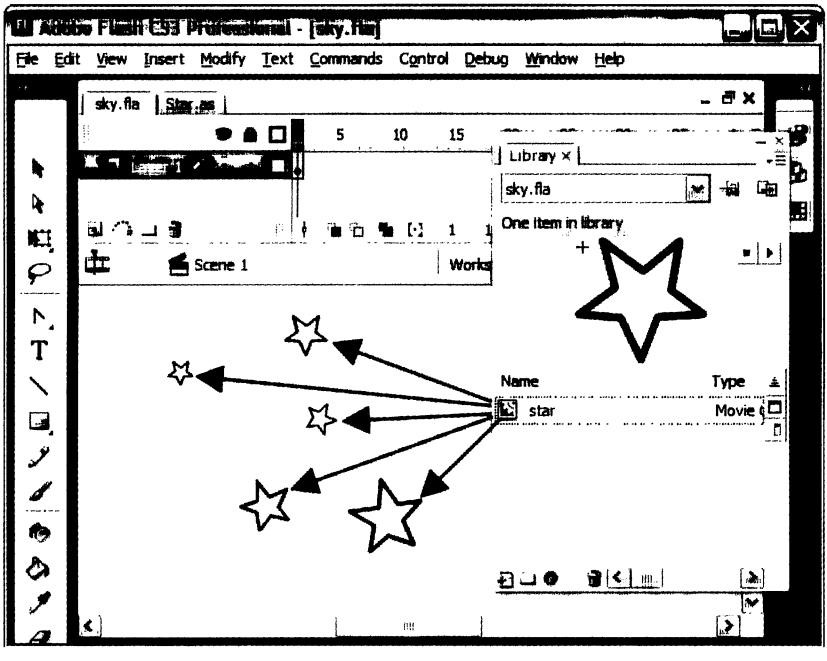

Рис. 29.9. Создание экземпляров символа звезды

Теперь, когда мы знаем, как вручную создавать экземпляры символов Movie Clip (Клип), рассмотрим, как обращаться к этим экземплярам и управлять ими.

Обращение к созданным вручную экземплярам символов

На этапе выполнения каждый экземпляр символа, который помещен на любую временную шкалу в FLA-файле, автоматически становится отображаемым ребенком объекта языка ActionScript, представляющего данную временную шкалу.

Например, в предыдущем разделе мы добавили пять экземпляров символа звезды на основную временную шкалу файла `sky.fla`. В результате на этапе выполнения эти пять экземпляров становятся отображаемыми объектами-детьми экземпляра класса документа файла `sky.fla`. Чтобы доказать это, создадим класс документа для файла `sky.fla` и воспользуемся методом `getChildAt( )` для получения списка всех детей основной временной шкалы.

```
package {
  import flash.display.MovieClip;

  public class Sky extends MovieClip {
    public function Sky ( ) {
      for (var i:int=0; i < numChildren; i++) {
        trace("Found child: " + getChildAt(i));
      }
    }
  }
}
```

В случае с предыдущим классом документа, когда приложение `sky.swf` будет экспортировано и воспроизведено в режиме **Test Movie** (Проверка фильма) среды разработки Flash, в окне **Output** (Вывод) появятся следующие строки:

```
Found child: [object Star]
Found child: [object Star]
Found child: [object Star]
Found child: [object Star]
Found child: [object Star]
```

Код в методе-конструкторе класса документа SWF-файла может обращаться к любым дочерним элементам, размещенным вручную в первом кадре SWF-файла, но не во втором или последующих кадрах (поскольку элементы, размещенные в последующих кадрах, не добавляются в список отображаемых детей до тех пор, пока головка воспроизведения не достигнет этих кадров).

С точки зрения языка ActionScript пять экземпляров символа звезды на основной временной шкале являются экземплярами класса `Star`, который наследуется от класса `MovieClip`, поэтому мы можем управлять звездами с помощью переменных и методов класса `DisplayObject` (или любого другого предка класса `MovieClip`). Например, следующий код использует переменную экземпляра `x` класса `DisplayObject` для выравнивания звезд по вертикальной линии, проходящей через координату 100 по оси X:

```
for (var i:int=0; i < numChildren; i++) {
  getChildAt(i).x = 100;
}
```

Тем не менее метод `getChildAt( )`, применяемый в предыдущем коде, имеет одно ограничение. Поскольку в среде разработки Flash глубины отдельных экземпляров символов не совсем очевидны, использование метода `getChildAt( )` для выборочного управления конкретным экземпляром символа оказывается весьма затруднительным. Чтобы без труда идентифицировать конкретный экземпляр и управлять им из кода, мы присваиваем ему имя экземпляра.

Имена экземпляров

Имя экземпляра — это простое строковое имя, присваиваемое экземпляру символа. Чтобы присвоить имя созданному вручную экземпляру символа, мы используем палитру **Properties** (Свойства). Например, чтобы присвоить имена экземплярам

символа звезды из предыдущего раздела, выполните такую последовательность действий.

1. Откройте палитру Properties (Свойства) (команда меню Window ▸ Properties (Окно ▸ Свойства)).
2. Выберите экземпляр звезды на сцене.
3. Вместо значения <Instance Name> (Имя экземпляра) введите значение `star1`.
4. Повторите шаги 2–3, чтобы присвоить имена оставшимся экземплярам звезды, начиная со значения `star2` и заканчивая значением `star5`.

При выполнении шага 3 предыдущей процедуры палитра Properties (Свойства) будет выглядеть так, как показано на рис. 29.10 (в нижней его части).

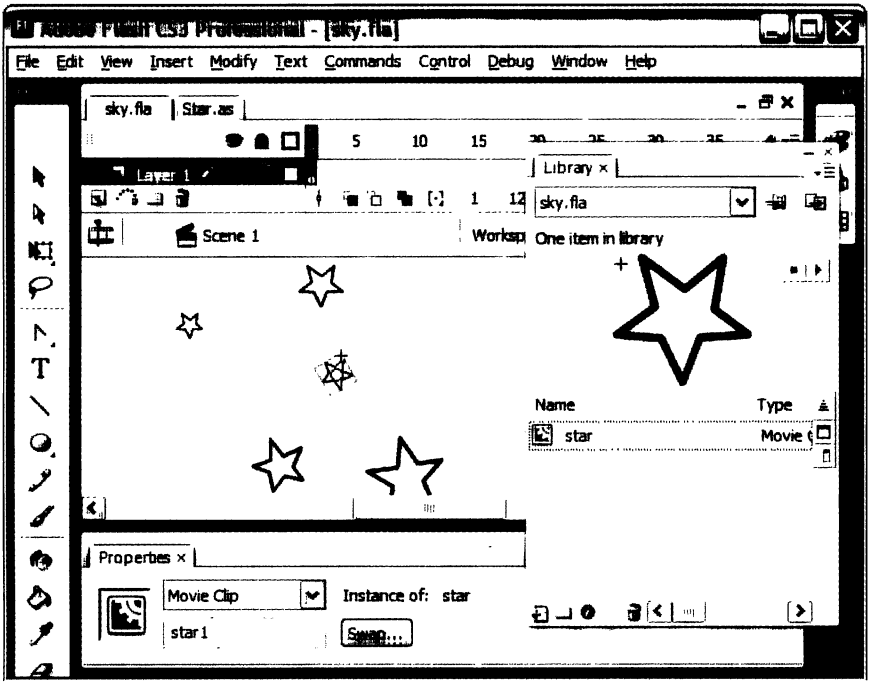

Рис. 29.10. Присваивание имени экземпляру символа звезды

Обратиться по имени к созданному вручную экземпляру символа из кода на языке ActionScript можно с помощью метода экземпляра `getChildByName( )` класса `DisplayObjectContainer`. Например, следующий код использует метод `getChildByName( )`, чтобы переместить экземпляр `"star3"` (звезда с именем экземпляра `"star3"`) в точку с координатой (0; 0):

```
package {
  import flash.display.MovieClip;

  public class Sky extends MovieClip {
    public function Sky ( ) {
      getChildByName("star3").x = 0;
```

```
      getChildByName("star3").y = 0;
    }
  }
}
```

На этапе выполнения к имени экземпляра можно обратиться через переменную экземпляра `name` класса `DisplayObject`. Например, следующий код отображает имена всех экземпляров, размещенных в первом кадре основной временной шкалы файла `sky.fla`:

```
for (var i:int=0; i < numChildren; i++) {
  trace(getChildAt(i).name);
}
```

Кроме того, на этапе выполнения имя экземпляра объекта можно изменить, присвоив новое значение его переменной `name`, как показано в следующем коде:

```
некийОбъектDisplayObject.name = "некоеИмя";
```

Тем не менее изменение имен экземпляров на этапе выполнения обычно усложняет отладку кода, поэтому вы должны избегать использования подобной практики в своих программах.

Соответствие переменных именам экземпляров

Метод `getChildByName( )` из предыдущего раздела позволяет успешно обращаться к конкретному экземпляру символа, однако он не совсем удобен. Чтобы упростить обращение к создаваемым вручную экземплярам символов из кода на языке ActionScript, компилятор Flash предоставляет два автоматических сервиса. Во-первых, когда компилятор встречает именованный экземпляр на временной шкале, он автоматически присваивает его переменной экземпляра класса документа временной шкалы или связанного класса, имеющей такое же имя, как у данного экземпляра. Во-вторых, в некоторых случаях, когда соответствующая переменная экземпляра еще не существует в классе документа временной шкалы или связанном классе, компилятор автоматически создает ее. Рассмотрим эти два автоматических сервиса на примере.

Возвращаясь к примеру с файлом `sky.fla` из предыдущего раздела, вспомним, что классом документа файла `sky.fla` является `Sky` и что основная временная шкала файла `sky.fla` содержит пять экземпляров символа звезды с именами `"star1"` — `"star5"`. Когда компилятор Flash осуществляет компиляцию файла `sky.swf`, он автоматически добавляет в класс `Sky` код, который присваивает эти пять экземпляров звезды пяти переменным экземпляра с именами `star1` — `star5`. Результат эквивалентен добавлению следующего кода в начало конструктора класса `Sky`:

```
package {
  import flash.display.MovieClip;

  public class Sky extends MovieClip {
    public function Sky ( ) {
      star1 = getChildByName("star1");
      star2 = getChildByName("star2");
      star3 = getChildByName("star3");
      star4 = getChildByName("star4");
```

```
      star5 = getChildByName("star5");
    }
  }
}
```

Конечно, в данном случае никаких переменных с именами `star1` — `star5` в классе `Sky` на самом деле не определено. Следовательно, предыдущее автоматическое присваивание экземпляров переменным потенциально опасно.

Возникнет ли ошибка, зависит от параметра компиляции под названием **Automatically declare stage instances** (Автоматически объявлять экземпляры сцены), устанавливаемого для каждого FLA-файла. Данный параметр находится в области **Stage** (Сцена) окна **ActionScript Settings** (Параметры ActionScript), открыть которое можно, выбрав команду меню **File** ▸ **Publish Settings** (Файл ▸ Настройки публикации) и нажав кнопку **Settings** (Параметры), расположенную возле параметра **ActionScript version** (Версия ActionScript) на вкладке **Flash**. Когда флажок **Automatically declare stage instances** (Автоматически объявлять экземпляры сцены) установлен (по умолчанию), компилятор не только присваивает экземпляры сцены соответствующим переменным, но и автоматически объявляет эти переменные. Например, если мы установим флажок **Automatically declare stage instances** (Автоматически объявлять экземпляры сцены) для файла `sky.fla`, то компилятор автоматически объявит переменные экземпляра с именами `star1` — `star5` в классе `Sky`. Результат эквивалентен добавлению следующего кода в класс `Sky` (обратите внимание, что для автоматически объявляемых переменных используется модификатор управления доступом `public`):

```
package {
  import flash.display.MovieClip;

  public class Sky extends MovieClip {
    public var star1:Star;
    public var star2:Star;
    public var star3:Star;
    public var star4:Star;
    public var star5:Star;

    public function Sky ( ) {
      star1 = getChildByName("star1");
      star2 = getChildByName("star2");
      star3 = getChildByName("star3");
      star4 = getChildByName("star4");
      star5 = getChildByName("star5");
    }
  }
}
```

Благодаря предыдущему автоматически сгенерированному коду внутри класса `Sky` мы можем ссылаться на экземпляры звезд из основной временной шкалы файла `sky.fla` непосредственно по имени экземпляра. Например, следующий код перемещает экземпляр `"star3"` (звезда с именем экземпляра `"star3"`) в точку с координатой (0; 0):

```
package {
  import flash.display.MovieClip;

  public class Sky extends MovieClip {
    public function Sky ( ) {
      star3.x = 0;
      star3.y = 0;
    }
  }
}
```

Подобным образом обращаться к экземплярам звезд непосредственно по имени экземпляра можно также из любого сценария кадра на основной временной шкале файла `sky.fla`. Например, если поместить следующий код в сценарий первого кадра файла `sky.fla`, то экземпляр `"star5"` будет повернут на 30°:

```
star5.rotation = 30;
```

Ловко, да?

Однако с учетом вышеописанного автоматического поведения, если флажок **Automatically declare stage instances** (Автоматически объявлять экземпляры сцены) установлен, программист должен проявлять осторожность, чтобы не определить переменные экземпляра, имена которых совпадают с именами экземпляров символов. Например, следующий код определяет переменную экземпляра `star1`, имя которой совпадает с именем экземпляра символа с основной временной шкалы файла `sky.fla`:

```
package {
  import flash.display.MovieClip;

  public class Sky extends MovieClip {
    public var star1:Star;

    public function Sky ( ) {
    }
  }
}
```

Определение переменной экземпляра в предыдущем коде вызывает следующую ошибку на этапе компиляции:

```
A conflict exists with definition star1 in namespace internal.
```

На русском языке она будет выглядеть так: **Существует конфликт с определением star1 в пространстве имен internal.**

В отличие от этого, когда флажок **Automatically declare stage instances** (Автоматически объявлять экземпляры сцены) снят, компилятор не объявляет соответствующие переменные для экземпляров сцены. Например, если мы снимем данный флажок для файла `sky.fla`, то компилятор не объявит переменные экземпляров с именами `star1` — `star5` в классе `Sky`. Тем не менее он по-прежнему присваивает тем переменным ссылки на пять экземпляров символа звезды, размещенных на основной временной шкале. В итоге мы должны добавить объявления соответствующих

переменных в класс `Sky`, как показано в следующем коде. Обратите внимание, что по необходимости переменные объявлены с использованием модификатора управления доступом `public`.

```
package {
  import flash.display.MovieClip;

  public class Sky extends MovieClip {
    public var star1:Star;
    public var star2:Star;
    public var star3:Star;
    public var star4:Star;
    public var star5:Star;

    public function Sky ( ) {
    }
  }
}
```

Если убрать объявления переменных из предыдущего кода, то возможны два результата в зависимости от того, объявлен класс с атрибутом `dynamic` или нет (обратитесь к разд. «Динамические переменные экземпляра» гл. 15). Если класс объявлен без использования атрибута `dynamic` (как в случае с классом `Sky`), на этапе выполнения возникнет следующая ошибка, поскольку инструкции присваивания, автоматически добавленные компилятором, ссылаются на несуществующие переменные:

```
ReferenceError: Error #1056: Cannot create property star5 on Sky.
```

По-русски она будет выглядеть следующим образом: **Ошибка обращения: невозможно создать свойство star5 в классе Sky.**

Если, с другой стороны, класс объявлен *с* использованием атрибута `dynamic`, то автоматически добавляемые инструкции присваивания на этапе выполнения просто создают новые динамические переменные экземпляра (с именами `star1` — `star5`) для каждого экземпляра класса.

Несмотря на то что по умолчанию флажок Automatically declare stage instances (Автоматически объявлять экземпляры сцены) установлен, многие программисты предпочитают не устанавливать его. Как мы только что узнали, если он не установлен, то нединамические классы, связанные с временными шкалами, должны объявлять переменные, которые соответствуют всем именованным экземплярам сцены. Подобные явные объявления проливают свет на природу появления переменных в использующем их классе и исключают возникновение ошибок компиляции при редактировании данного класса в других средах разработки (например, в приложении Flex Builder 2).

Стоит отметить, что автоматически генерируемые классы документа и автоматически генерируемые связанные классы всегда компилируются с установленным флажком Automatically declare stage instances (Автоматически объявлять экземпляры сцены). Следовательно, обратиться к именованным экземплярам сцены любой временной шкалы, связанной с автоматически генерируемым классом документа или автоматически генерируемым связанным классом, можно через соответствующие переменные экземпляра.

Обращение к созданному вручную тексту

В предыдущих разделах мы познакомились с тремя инструментами, которые позволяют обращаться к созданным вручную экземплярам символов из кода на языке ActionScript, а именно:

- метод `getChildAt( );`
- метод `getChildByName( );`
- автоматически присваиваемые переменные экземпляра.

Те же три инструмента могут быть использованы и для обращения к созданному вручную тексту.

В среде разработки Flash текстовые поля создаются с помощью инструмента Text (Текст). Созданные вручную текстовые поля типа Dynamic Text (Динамический текст) или Input Text (Вводимый текст) на этапе выполнения представляются экземплярами класса `flash.text.TextField`. Как и в случае с экземплярами символов, поля типа Dynamic Text (Динамический текст) и Input Text (Вводимый текст) становятся отображаемыми детьми объекта, представляющего временную шкалу, на которой они размещаются. Кроме того, как и в случае с экземплярами символов, текстовым полям описанных типов могут присваиваться имена экземпляров, которые соответствуют именам автоматически присваиваемых переменных экземпляра.

Например, приведенная ниже последовательность действий описывает, как переместить созданное вручную текстовое поле типа Dynamic Text (Динамический текст) в точку с координатой (200; 300), используя код на языке ActionScript.

1. В среде разработки Flash создайте новый FLA-файл и назовите его `message.fla`.
2. На палитре Tools (Инструменты) выберите инструмент Text (Текст).
3. Щелкните кнопкой мыши на сцене первого кадра основной временной шкалы файла `message.fla`.
4. В появившемся поле введите слово `hello`.
5. На палитре Properties (Свойства) (команда меню Window ▸ Properties (Окно ▸ Свойства)) измените значение Static Text (Статический текст) на Dynamic Text (Динамический текст).
6. На палитре Properties (Свойства) вместо значения <Instance Name> (Имя экземпляра) введите `msg`.
7. Щелкните кнопкой мыши на первом кадре основной временной шкалы файла `message.fla` (выделите его).
8. Откройте палитру Actions (Действия) (команда меню Window ▸ Actions (Окно ▸ Действия)).
9. Введите следующий код на палитре Actions (Действия):

```
msg.x = 200;
msg.y = 300;
```

Программное управление временной шкалой

Теперь, когда мы знаем, как обращаться к созданным вручную экземплярам символов Movie Clip (Клип), кратко рассмотрим инструменты, предназначенные для управления воспроизведением этих экземпляров. В следующем списке представлены наиболее важные методы и переменные класса `MovieClip` для управления временной шкалой. Дополнительную информацию о классе `MovieClip` можно найти в справочнике по языку ActionScript корпорации Adobe.

- `play( )` — инициирует последовательное отображение кадров временной шкалы клипа. Кадры отображаются со скоростью, установленной через переменную экземпляра `frameRate` класса `Stage` среды выполнения Flash.
- `stop( )` — останавливает воспроизведение клипа. Однако обратите внимание, что, несмотря на остановку клипа, среда выполнения Flash продолжает воспроизводить звуки, реагировать на действия пользователя и обрабатывать события, включая события `Event.ENTER_FRAME`. Таким образом, визуальные изменения, происходящие в обработчиках событий, в результате использования метода `setInterval( )` или события `Timer`, отображаются даже в том случае, когда головка воспроизведения остановлена. Метод `stop( )` просто предотвращает дальнейшее перемещение головки воспроизведения по временной шкале клипа.
- `gotoAndPlay( )` — перемещает головку воспроизведения временной шкалы клипа на кадр с указанным номером или меткой, после чего начинает воспроизведение временной шкалы с этой точки. Состояние других клипов не изменяется. Чтобы начать воспроизведение других клипов, следует вызвать метод `play( )` или `gotoAndPlay( )` для каждого клипа по отдельности.
- `gotoAndStop( )` — перемещает головку воспроизведения временной шкалы клипа на кадр с указанным номером или меткой, после чего останавливает воспроизведение клипа.
- `nextFrame( )` — перемещает головку воспроизведения временной шкалы клипа вперед на один кадр и останавливает воспроизведение на этом кадре.
- `prevFrame( )` — перемещает головку воспроизведения временной шкалы клипа назад на один кадр и останавливает воспроизведение на этом кадре.
- `currentFrame` — возвращает номер кадра, на котором в настоящий момент находится головка воспроизведения клипа. Стоит отметить, что номером первого кадра является 1, а не 0; таким образом, значение переменной `currentFrame` находится в диапазоне от 1 до значения переменной `totalFrames`.
- `currentLabel` — возвращает строку, представляющую метку текущего кадра, которая задается в среде разработки Flash.
- `currentLabels` — возвращает массив, содержащий все метки текущей временной шкалы.
- `totalFrames` — возвращает количество кадров на временной шкале клипа.

Применим некоторые из перечисленных методов и переменных к нашему документу `sky.fla`. Все следующие примеры могут быть помещены либо в метод класса `Sky`, либо на основную временную шкалу файла `sky.fla`.

Следующий код отображает четвертый кадр экземпляра "star1":

```
star1.gotoAndStop(4);
```

Приведенный ниже код останавливает воспроизведение временной шкалы экземпляра "star3":

```
star3.stop( );
```

Следующий код перемещает головку воспроизведения экземпляра "star5" на два кадра вперед:

```
star5.gotoAndStop(star5.currentFrame + 2);
```

Приведенный далее код останавливает воспроизведение временной шкалы всех экземпляров звезды, находящихся на основной временной шкале:

```
for (var i:int=0; i < numChildren; i++) {
  getChildAt(i).stop( );
}
```

Создание экземпляров символов среды разработки Flash из кода на языке ActionScript

Ранее из разд. «Связанные классы для символов Movie Clip (Клип)» мы узнали, что экземпляры символа могут создаваться вручную в среде разработки Flash путем перетаскивания имени символа из библиотеки в сцену временной шкалы. Экземпляры символов, экспортируемые для языка ActionScript, могут также создаваться непосредственно из кода с помощью стандартного оператора new.

Например, ранее мы связали символ звезды с классом Star. Чтобы создать экземпляр символа звезды из кода, мы используем следующее выражение:

```
new Star( )
```

Чтобы добавить экземпляр символа звезды на основную временную шкалу приложения sky.swf на этапе выполнения, мы бы включили следующий код либо в сценарий кадра на основной временной шкале, либо в метод класса Sky:

```
var star:Star = new Star( );
addChild(star);
```

Следующий код создает 50 экземпляров звезды и размещает их на экране в произвольных точках:

```
var sky:Array = new Array( );
for (var i:int = 0; i < 50; i++) {
  sky.push(new Star( ));
  sky[i].x = Math.floor(Math.random( )*550);
  sky[i].y = Math.floor(Math.random( )*400);
  addChild(sky[i]);
}
```

Стоит отметить, что код на языке ActionScript может быть использован для создания экземпляров любого символа, экспортируемого для языка ActionScript, неза-

висимо от того, имеет этот символ собственный связанный класс или нет. Иными словами, класс, указанный в окне Linkage Properties (Свойства связывания), не обязательно должен быть существующим пользовательским классом (как, например, класс `Star`). Как мы узнали ранее, если класс, указанный в окне Linkage Properties (Свойства связывания), не может быть найден с использованием пути к классам, то компилятор Flash создает его автоматически. Автоматически генерируемый класс может применяться для создания экземпляров данного символа из кода.

Предположим, что у нас есть символ Movie Clip (Клип) с именем `box_symbol` и мы хотим создать его экземпляры из кода на языке ActionScript. Рассмотрим необходимую последовательность действий.

1. Выберите символ `box_symbol` в библиотеке.
2. Откройте меню Options (Свойства) палитры Library (Библиотека), щелкнув кнопкой мыши на значке в правом верхнем углу палитры, и выберите команду Linkage (Связывание).
3. В области Linkage (Связывание) окна Linkage Properties (Свойства связывания) установите флажок Export for ActionScript (Экспорт для ActionScript).
4. В поле Class (Класс) окна Linkage Properties (Свойства связывания) введите значение `Box`.
5. Нажмите кнопку OK.
6. По умолчанию в результате выполнения предыдущих шагов среда разработки Flash выведет окно с предупреждением следующего содержания: A definition for this class could not be found in the classpath, so one will be automatically generated in the SWF file on export (Определение для данного класса не может быть найдено в пути к классам, поэтому он будет автоматически сгенерирован в SWF-файле при экспорте). Нажмите кнопку OK, чтобы закрыть это предупреждение.

Чтобы создать экземпляр символа `box_symbol` после выполнения предыдущих шагов, мы используем выражение `new Box( )` на любой временной шкале или в любом классе SWF-файла, содержащего символ `box_symbol`.

Имена экземпляров для отображаемых объектов, создаваемых программным путем

Между прочим, как и в случае с экземплярами, создаваемыми вручную, отображаемым объектам, которые создаются программным путем, можно *тоже* присваивать имя экземпляра через переменную `name`. Например, следующий код создает экземпляр класса `TextField` и присваивает ему имя экземпляра `"price"`, добавляет его в контейнер, а затем по имени получает на него ссылку.

```
var t:TextField = new TextField( );
t.text = "$99.99";
t.name = "price"
var detailsPage:Sprite = new Sprite( );
detailsPage.addChild(t);
trace(detailsPage.getChildByName("price")); // Выводит: [object TextField]
```

Приведенный код может показаться удобным, поскольку он предоставляет способ для обращения к объекту в списке отображения с использованием некоторой определенной программистом метки, а не ссылки на объект или позиции глубины. Тем не менее подобное использование имен экземпляров может служить причиной возникновения ошибок — язык ActionScript не требует, чтобы имена экземпляров были уникальными, и не генерирует исключение при попытке обращения к несуществующим именам экземпляров. Таким образом, следует избегать использования имен экземпляров для отображаемых объектов, создаваемых программным путем.

Имена экземпляров в основном должны использоваться только при обращении к экземплярам текстовых полей или символов библиотеки, созданным вручную в среде разработки Flash.

К отображаемым объектам, создаваемым программным путем, следует всегда обращаться по ссылке. Например, следующий код демонстрирует две версии метода `displayPrice( )` (это гипотетический метод, который отображает стоимость товара). В обоих случаях стоимость отображается в объекте `TextField`. В первой версии (рекомендуемой) объект `TextField`, который будет отображать стоимость, передается в метод как объектная ссылка. Во второй версии в метод передается объект `DisplayObjectContainer`, который содержит объект `TextField`, и метод получает ссылку на объект `TextField` по имени экземпляра.

```
// Рекомендуется
public function displayPrice (priceField:TextField, price:Number):void {
  priceField.text = "$" + price;
}

// Не рекомендуется
public function displayPrice (orderForm:Sprite, price:Number):void {
  TextField(orderForm.getChildByName("price")).text = "$" + price;
}
```

Всегда, когда это возможно, отображаемые объекты должны быть доступны для зависимых частей программы по ссылке.

В языке MXML имя экземпляра отображаемого объекта может быть установлено через атрибут `id`, а обращаться к именованным отображаемым объектам можно также с помощью метода `getChildByName( )`. Тем не менее, как и в случае с приложениями, разработанными на чистом ActionScript, вместо имен экземпляров предпочтительнее использовать ссылки.

Связывание нескольких символов с одним суперклассом

Ранее из разд. «Связанные классы для символов Movie Clip (Клип)» мы узнали, как связать класс с символом Movie Clip (Клип). Теперь рассмотрим, как обеспечить

одинаковым программным поведением несколько различных символов Movie Clip (Клип), связав их с одним суперклассом.

В качестве примера создадим простую форму регистрации с двумя различными графическими стилями интерфейса. Подобная форма может потребоваться в приложении, которое предоставляет пользователям возможность выбора дизайна интерфейса, или *обложки*. Наши формы будут выглядеть по-разному, однако их поведение будет одинаковым.

Мы начнем с создания FLA-файла `LoginApp.fla` в среде разработки Flash. В нем мы создадим два символа — по одному для каждого графического стиля формы регистрации. Назовем первый символ именем `LoginForm_Style1`, а второй символ — `LoginForm_Style2`. В каждый символ формы регистрации добавим два созданных вручную текстовых поля (с именами `username` и `password`) и кнопку отправки данных (с именем `submitBtn`). Сама по себе кнопка является нарисованным вручную экземпляром символа Movie Clip (Клип). На рис. 29.11 изображены два символа формы регистрации.

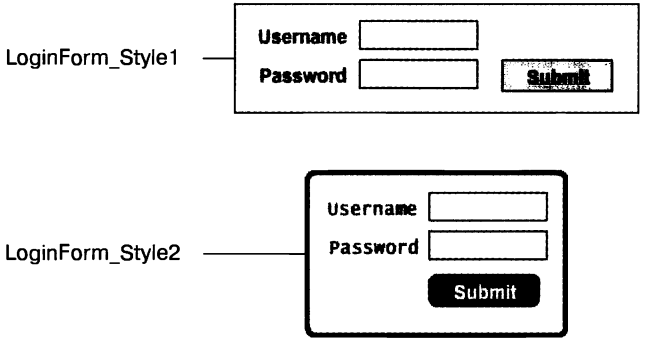

Рис. 29.11. Символы формы регистрации

Далее создадим класс `LoginForm`, который управляет поведением символов формы регистрации. Он реагирует на нажатия кнопки отправки данных и передает полученную информацию на сервер. В этом примере снимем флажок компиляции Automatically declare stage instances (Автоматически объявлять экземпляры сцены). Соответственно внутри класса `LoginForm` объявим созданные вручную элементы из символов формы регистрации в качестве переменных экземпляра. Имена переменных экземпляра — `username`, `password` и `submitBtn` — соответствуют именам экземпляров в символах формы регистрации.

Рассмотрим код для класса `LoginForm`:

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;

  public class LoginForm extends MovieClip {
    public var username:TextField;
```

```
    public var password:TextField;
    public var submitBtn:SimpleButton;

    public function LoginForm ( ) {
      submitBtn.addEventListener(MouseEvent.CLICK, submitListener);
    }

    private function submitListener (e:MouseEvent):void {
      submit(username.text, password.text);
    }

    public function submit (name:String, pass:String):void {
      trace("Now submitting user: " + name + " with password: " + pass);

      // Теперь передаем полученную информацию
      // на сервер (код не показан).
      // Обычно для отправки данных на сервер
      // используется класс flash.net.URLLoader.
    }
  }
}
```

Наконец, связываем поведение класса `LoginForm` с символами формы регистрации. Однако поскольку один класс не может быть связан более чем с одним символом, мы не сможем связать класс `LoginForm` непосредственно с символами формы регистрации. Вместо этого мы должны связать каждый символ формы регистрации с *подклассом* класса `LoginForm`. Этот процесс заключается в выполнении следующих шагов.

1. Выберите символ `LoginForm_Style1` в библиотеке файла `LoginApp.fla`.
2. Откройте меню **Options** (Свойства) палитры **Library** (Библиотека), щелкнув кнопкой мыши на значке в правом верхнем углу палитры, и выберите команду **Linkage** (Связывание).
3. В области **Linkage** (Связывание) окна **Linkage Properties** (Свойства связывания) установите флажок **Export for ActionScript** (Экспорт для ActionScript).
4. В поле **Base class** (Базовый класс) окна **Linkage Properties** (Свойства связывания) введите `LoginForm`.
5. В поле **Class** (Класс) окна **Linkage Properties** (Свойства связывания) введите `LoginForm_Style1`.
6. Нажмите кнопку **OK**.
7. По умолчанию в результате выполнения предыдущих шагов среда разработки Flash выведет окно с предупреждением следующего содержания: **A definition for this class could not be found in the classpath, so one will be automatically generated in the SWF file on export** (Определение для данного класса не может быть найдено в пути к классам, поэтому он будет автоматически сгенерирован в SWF-файле при экспорте). Нажмите кнопку **OK** этого окна.
8. Выберите символ `LoginForm_Style2` в библиотеке файла `LoginApp.fla`, после чего повторите шаги 2–6, заменив значение `LoginForm_Style1` значением `LoginForm_Style2` на шаге 6.

В результате выполнения предыдущих шагов компилятор Flash на этапе компиляции автоматически создаст два класса — `LoginForm_Style1` и `LoginForm_Style2`, каждый из которых расширяет класс `LoginForm`. Затем компилятор свяжет символ `LoginForm_Style1` с классом `LoginForm_Style1`, а символ `LoginForm_Style2` — с классом `LoginForm_Style2`. Таким образом, оба символа наследуют поведение класса `LoginForm`.

Композиционный подход как альтернатива связанным классам

В этом разделе мы научились обеспечивать поведением символы Movie Clip (Клип), связывая их с пользовательскими классами. В качестве альтернативного подхода, чтобы обеспечить символ программным поведением, мы можем просто создать экземпляр любого пользовательского класса на временной шкале символа и затем использовать этот экземпляр для управления данным символом.

Например, в предыдущем разделе мы связали символы формы регистрации с классом `LoginForm`, чтобы предоставить этим символам возможность отправлять информацию на сервер программным путем. Однако можно утверждать, что класс `LoginForm` не является подтипом класса `MovieClip`. Скорее это простая коммуникационная утилита, которая помогает получить введенные данные из пользовательского интерфейса. Как таковая, она может (и, возможно, должна) быть определена в виде отдельного класса для последующего использования любым символом, который соглашается предоставлять соответствующие данные.

Для простоты сравнения, следующий код демонстрирует новую версию класса `LoginForm`, переработанного для использования в качестве вспомогательного класса на временной шкале символа. Новый класс имеет новое имя `LoginManager`, отражающее его новую роль — класс коммуникационной утилиты. Обратите внимание, что конструктор нового класса принимает ссылки на объекты пользовательского интерфейса, поддерживающие пользовательский ввод.

```
package {
  import flash.display.*;
  import flash.text.*;
  import flash.events.*;

  public class LoginManager {
    private var username:TextField;
    private var password:TextField;

    public function LoginManager (username:TextField,
                                  password:TextField,
                                  submitBtn:SimpleButton) {
      this.username = username;
      this.password = password;
      submitBtn.addEventListener(MouseEvent.CLICK, submitListener);
    }
```

```
    private function submitListener (e:MouseEvent):void {
      submit(username.text, password.text);
    }

    public function submit (name:String, pass:String):void {
      trace("Now submitting user: " + name + " with password: " + pass);

      // Теперь передаем полученную информацию на сервер (код не показан).
      // Обычно для отправки данных на сервер используется класс
      // flash.net.URLLoader.
    }
  }
}
```

Чтобы использовать класс `LoginManager`, в каждом из символов формы регистрации из предыдущего раздела должен быть определен сценарий кадра со следующим кодом (предположительно, в кадре 1, но в случае с анимированной формой, возможно, далее на временной шкале). В следующем коде аргументы `username`, `password` и `submitBtn` являются именами экземпляров текстовых полей и кнопки в символе формы регистрации:

```
var loginManager:LoginManager = new LoginManager(username,
                                                  password,
                                                  submitBtn);
```

Различие между подходом с использованием класса `LoginManager` и подходом с использованием класса `LoginForm` из предыдущего раздела, по сути дела, аналогично различию между композицией и наследованием (дополнительные сведения можно найти в подразд. «Наследование в сравнении с композицией» разд. «Теория наследования» гл. 6). Потенциальными преимуществами подхода с использованием композиции являются следующие.

- Класс `LoginManager` может наследоваться от любого произвольного класса, в отличие от класса `LoginForm`, который должен наследоваться от класса `MovieClip`.
- Разработчик символа формы регистрации может изменять имена экземпляров текстовых полей и кнопки отправки данных, не оказывая влияния на функциональность класса `LoginManager`.

Теперь перейдем к рассмотрению нашей последней темы по среде разработки Flash, касающейся предварительной загрузки классов.

Предварительная загрузка классов

При компиляции SWF-файла среда разработки Flash по умолчанию экспортирует все классы из первого кадра SWF-файла. Как результат, загрузка всех классов должна быть завершена до того, как на экране отобразится первый кадр SWF-файла. В зависимости от общего размера классов, включенных в SWF-файл, этот процесс может привести к заметной задержке перед началом воспроизведения SWF-файла.

Чтобы избежать этого, мы можем экспортировать классы SWF-файла *после* кадра 1 на основной временной шкале, а затем создать простой сценарий временной шкалы, который отображает информацию о ходе загрузки классов.

В качестве примера изменим наш файл `sky.fla` таким образом, чтобы классы, которые он использует, не загружались до кадра 15. Стоит отметить, однако, что класс документа SWF-файла — и любой класс, на который прямо или косвенно ссылается класс документа, — всегда загружается в кадре 1. Таким образом, чтобы предотвратить загрузку класса `Star` до кадра 1, мы должны сначала удалить имена всех созданных вручную экземпляров класса `Star` и определения переменных `star1` — `star5` из кода класса `Sky`. Чтобы удалить имена экземпляров, мы выбираем каждый экземпляр символа `Star` на сцене и удаляем его имя на палитре **Properties** (Свойства).

Далее описывается, как загрузить классы файла `sky.fla` в кадре 15 и отобразить соответствующее сообщение в процессе загрузки классов.

Сначала выполним следующие шаги, чтобы указать компилятору Flash на необходимость экспортирования классов файла `sky.fla` в кадре 15.

1. Откройте файл `sky.fla` в среде разработки Flash.
2. Выберите команду меню **File** ▸ **Publish Settings** (Файл ▸ Настройки публикации).
3. В окне **Publish Settings** (Настройки публикации) нажмите кнопку **Settings** (Параметры), расположенную возле параметра **ActionScript version** (Версия ActionScript) с установленным значением **ActionScript 3.0** на вкладке **Flash**.
4. В поле **Export Classes in Frame** (Экспортировать классы в кадре) окна **ActionScript Settings** (Параметры ActionScript) введите `15`.
5. Нажмите кнопку **OK**, для того чтобы закрыть окно **ActionScript Settings** (Параметры ActionScript).
6. Снова нажмите кнопку **OK**, чтобы закрыть **Publish Settings** (Настройки публикации).

Далее на временную шкалу файла `sky.fla` мы добавляем очень простой предварительный загрузчик, который отображает сообщение **Loading** (Загрузка) в процессе загрузки классов.

Следующие процедуры подразумевают предварительное знакомство со слоями и метками кадров временной шкалы, но, даже если вы никогда не сталкивались с этими аспектами среды разработки Flash, вы сможете выполнить процедуры в соответствии с приведенным описанием.

Сначала на основной временной шкале файла `sky.fla` мы дважды щелкаем кнопкой мыши на имени слоя **Layer 1** (Слой 1) и изменяем его на **sky**. Затем мы увеличиваем размер временной шкалы до 15 кадров следующим образом.

1. На основной временной шкале щелкните на кадре 1 слоя **sky**, чтобы выделить этот кадр.
2. Щелкните кнопкой мыши на ключевом кадре 1 и, удерживая кнопку мыши, перетащите его на кадр 15 временной шкалы.

В результате увеличения временной шкалы, описанного в предыдущей процедуре, экземпляры символа звезды, которые до этого отображались в кадре 1, теперь будут отображаться в кадре 15 (где загружается класс `Star`).

Далее мы добавим новый слой для кода и назовем его scripts.

1. Выберите команду меню Insert ▸ Timeline ▸ Layer (Вставка ▸ Временная шкала ▸ Слой).
2. Дважды щелкните кнопкой мыши на имени нового слоя и измените его на scripts.

После этого мы добавим слой labels с двумя метками кадров — `loading` и `main`. Они обозначают состояние загрузки и стартовую точку для приложения соответственно.

1. Выберите команду меню Insert ▸ Timeline ▸ Layer (Вставка ▸ Временная шкала ▸ Слой).
2. Дважды щелкните кнопкой мыши на имени нового слоя и измените его на labels.
3. В кадрах 4 и 15 слоя labels добавьте новые ключевые кадры (используя команду меню Insert ▸ Timeline ▸ Keyframe (Вставка ▸ Временная шкала ▸ Ключевой кадр)).
4. Выбрав кадр 4 на слое labels, в поле Frame (Кадр) палитры Properties (Свойства) вместо <Frame Label> (Метка кадра) введите `loading`.
5. Выбрав кадр 15 на слое labels, в поле Frame (Кадр) палитры Properties (Свойства) вместо <Frame Label> (Метка кадра) введите `main`.

Теперь добавим сценарий предварительного загрузчика на слой scripts.

1. В кадре 5 слоя scripts добавьте новый ключевой кадр (используя команду меню Insert ▸ Timeline ▸ Keyframe (Вставка ▸ Временная шкала ▸ Ключевой кадр)).
2. Выбрав кадр 5 слоя scripts, введите следующий код на палитре Actions (Действия):

```
if (framesLoaded == totalFrames) {
  gotoAndStop("main");
} else {
  gotoAndPlay("loading");
}
```

Наконец, добавим сообщение о загрузке, отображаемое в процессе загрузки классов файла `star.fla`:

1. Выбрав кадр 1 слоя scripts, введите следующий код на палитре Actions (Действия):

```
import flash.text.*;

var loadMsg:TextField = new TextField( );
loadMsg.text = "Loading...Please wait.";
loadMsg.autoSize = TextFieldAutoSize.LEFT;
loadMsg.border     = true;
loadMsg.background = true;
loadMsg.selectable = false;
addChild(loadMsg);
```

2. В кадре 15 слоя scripts добавьте новый ключевой кадр (используя команду меню Insert ▸ Timeline ▸ Keyframe (Вставка ▸ Временная шкала ▸ Ключевой кадр)).

3. Выбрав кадр 15 слоя scripts, введите следующий код на палитре Actions (Действия):

```
removeChild(loadMsg);
```

Вот и все! Вы можете протестировать приложение `sky.swf`, выбрав команду меню Control ▸ Test Movie (Управление ▸ Проверка фильма). Находясь в режиме Text Movie (Проверка фильма), можно имитировать процесс загрузки SWF-файла, открыв палитру Bandwidth Profiler (Профайлер полосы пропускания) (команда меню View ▸ Bandwidth Profiler (Вид ▸ Профайлер полосы пропускания)) и выбрав команду меню View ▸ Simulate Download (Вид ▸ Имитировать загрузку). Поскольку наш класс `Star` имеет совсем небольшой размер, нужно установить очень низкую скорость загрузки (например, 200 байт в секунду), чтобы увидеть сообщение о предварительной загрузке. Чтобы изменить скорость загрузки, выберите команду меню View ▸ Download Settings (Вид ▸ Параметры загрузки).

Вы можете загрузить предыдущий пример файла `sky.fla` по адресу http://www.moock.org/eas3/examples.

При задании значения в поле Exports Classes in Frame (Экспортировать классы в кадре) примите во внимание, что:

- класс документа FLA-файла всегда экспортируется в кадре 1, независимо от значения, указанного в поле Exports Classes in Frame (Экспортировать классы в кадре);
- все классы, на которые ссылается класс документа или сценарий основной временной шкалы, экспортируются в кадре 1, независимо от значения, указанного в поле Exports Classes in Frame (Экспортировать классы в кадре);
- если установлен флажок Automatically declare stage instances (Автоматически объявлять экземпляры сцены) и сцена основной временной шкалы содержит именованные экземпляры символа, связанного с классом, то этот связанный класс будет экспортирован в кадре 1 (поскольку на этот класс ссылаются переменные, объявляемые автоматически для экземпляров на сцене).

Далее в программе: использование платформы разработки Flex

В этой главе были рассмотрены многие прикладные методики работы с кодом на языке ActionScript в среде разработки Flash. В следующей главе будет рассказано, как применять компоненты пользовательского интерфейса платформы разработки Flex в проекте приложения Flex Builder 2 на языке ActionScript. Затем из гл. 31 вы узнаете, как использовать готовый код совместно с другими разработчиками.

Минимальное приложение на языке MXML

В гл. 20 рассказывалось, что платформа разработки Flex включает развитый набор настраиваемых компонентов для создания пользовательских интерфейсов. Компоненты пользовательского интерфейса платформы разработки Flex обычно применяются в приложениях на языке MXML, но также могут применяться в приложениях, написанных в основном на языке ActionScript. Для тех читателей, которые не желают использовать язык MXML, в этой главе рассматривается абсолютный минимум шагов, необходимый для применения компонентов пользовательского интерфейса платформы разработки Flex в проекте приложения Flex Builder 2 с помощью минимально возможного количества инструкций языка MXML.

Во избежание недоразумений, эта глава не имеет ничего против языка MXML. В целом, язык MXML — это отличный инструмент для создания стандартизованных интерфейсов, развертываемых на платформе Flash. В этой главе просто рассматриваются ситуации, в которых либо макет приложения создается полностью программным путем, либо разработчик не имеет времени или желания изучать язык MXML.

Полную информацию о языке MXML и платформе разработки Flex можно найти в документации корпорации Adobe и книге «Programming Flex 2» (Kazoun and Lott, 2007) издательства O'Reilly.

Пользователи среды разработки Flash должны помнить, что приложение Flash CS3 включает собственный набор компонентов пользовательского интерфейса в пакете fl. Компоненты приложения Flash CS3 могут также быть использованы (как с технической точки зрения, так и с точки зрения допустимости) в программах на языке ActionScript, компилируемых с помощью приложения Flex Builder 2 или компилятора mxmlc.

Общий подход

Рассмотрим минимальный набор шагов, необходимых для создания приложения, которое использует компоненты пользовательского интерфейса платформы разработки Flex, на языке ActionScript.

1. В приложении Flex Builder 2 создайте проект Flex.
2. В новом проекте определите класс со статическим методом, который будет служить стартовой точкой для приложения.
3. В основном MXML-файле добавьте свойство события MXML, получающее уведомления о возникновении события `FlexEvent.APPLICATION_COMPLETE`

экземпляра класса Application верхнего уровня и в ответ вызывает статический метод, определенный на шаге 2.

4. В статическом методе, который был определен на шаге 2, создайте желаемые компоненты пользовательского интерфейса и добавьте их в экземпляр класса Application.

Перечисленные шаги подробно рассматриваются в следующих разделах.

Создание проекта Flex

Чтобы создать проект для приложения, выполните следующие шаги.

1. В приложении Flex Builder 2 выберите команду меню File ▸ New ▸ Flex Project (Файл ▸ Создать ▸ Проект Flex).
2. В окне New Flex Project (Новый проект Flex) установите переключатель How will your Flex application access data? (Как ваше приложение Flex будет обращаться к данным?) в положение Basic (Основной) и затем нажмите кнопку Next (Далее).
3. В поле Project name (Имя проекта) введите желаемое имя проекта и нажмите кнопку Next (Далее).
4. В поле Main source folder (Основная папка исходных файлов) введите src.
5. В поле Main application file (Основной файл приложения) введите желаемое имя файла с расширением MXML. Например, MinimalMXML.mxml.
6. Нажмите кнопку Finish (Готово).

В результате выполнения предыдущих шагов приложение Flex Builder 2 создаст новый проект, в пути библиотек которого будет автоматически включен файл framework.swc, содержащий компоненты пользовательского интерфейса.

После создания проекта мы добавим стартовую точку для приложения, как описано в следующем разделе.

Создание стартовой точки для приложения

Стартовая точка для нашего приложения представляет собой статический метод, определенный в пользовательском классе. В нашем примере мы назовем этот статический метод именем name( ), а пользовательский класс — EntryClass. Метод main( ) создает экземпляры компонентов пользовательского интерфейса и добавляет их в иерархию отображения экземпляра класса Application верхнего уровня. Экземпляр класса Application верхнего уровня создается автоматически, служит основой для всех приложений на языке MXML и предоставляет доступ к списку отображения. Обратиться к экземпляру класса Application верхнего уровня из нашей программы можно через статическую переменную application класса mx.core.Application.

В листинге 30.1 представлен код для класса EntryClass.

Листинг 30.1. Класс на языке ActionScript для минимального приложения MXML

```
package {
  import mx.controls.*;
  import mx.core.*;
```

```
  public class EntryClass {
    // Стартовая точка для приложения
    public static function main ( ):void {

      // Создаем компоненты пользовательского интерфейса
      // платформы разработки Flex.
      // Например:
      var button:Button = new Button( );

      // Добавляем компоненты
      // пользовательского интерфейса на экран.
      // Например:
      var mxmlApp:Application = Application(Application.application);
      mxmlApp.addChild(button);
    }
  }
}
```

Создав метод `EntryClass.main( )`, мы можем вызывать его в ответ на возникновение события `FlexEvent.APPLICATION_COMPLETE` экземпляра класса `Application` верхнего уровня, как описывается в следующем разделе.

Вызов стартовой точки для приложения

Ранее в подразд. «Создание проекта Flex» мы определили, что основным файлом приложения для нашего примера проекта является `MinimalMXML.mxml`. В итоге приложение Flex Builder 2 автоматически помещает в него следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">
</mx:Application>
```

Придется внести лишь одно незначительное изменение в приведенный код на языке MXML: мы должны добавить свойство события, которое вызывает метод `EntryClass.main( )` при получении экземпляром класса `Application` верхнего уровня уведомления о возникновении события `FlexEvent.APPLICATION_COMPLETE`. Этот подход продемонстрирован в следующем коде (соответствующие изменения выделены полужирным шрифтом):

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute" applicationComplete="EntryClass.main( )">
</mx:Application>
```

В результате выполнения предыдущего кода приложение, завершив процесс инициализации, автоматически вызовет метод `EntryClass.main( )`, который, в свою очередь, создаст экземпляры желаемых компонентов пользовательского интерфейса (смотрите, никакого MXML!).

Применим общий подход, рассмотренный в предыдущих разделах, на реальном примере.

Реальный пример применения компонентов пользовательского интерфейса

Чтобы увидеть, как применять код на языке ActionScript для создания и управления компонентами пользовательского интерфейса платформы разработки Flex, создадим простое приложение, которое содержит всего два экземпляра компонентов: экземпляр компонента `Button` и экземпляр компонента `DataGrid`. Приложение просто считает, сколько раз пользователь щелкнул кнопкой мыши на экземпляре компонента `Button`. Экземпляр компонента `DataGrid` отображает общее количество щелчков и интервал времени между ними.

Назовем основной файл MXML нашего приложения `MinimalMXML.mxml`, а класс, определяющий стартовую точку для нашего приложения, — `Clickometer`.

Рассмотрим код для файла `MinimalMXML.mxml`:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="vertical" applicationComplete="Clickometer.main( )">
</mx:Application>
```

В коде обратите внимание, что значение свойства события `applicationComplete` определяет метод (`Clickometer.main( )`), который будет вызван после того, как приложение завершит процесс инициализации. Стоит также отметить, что приложение использует «вертикальную» схему размещения. Информацию по вариантам размещения можно найти в разделе, посвященном описанию переменной экземпляра `layout` класса `Application`, справочника по языку ActionScript корпорации Adobe.

Теперь рассмотрим код для класса `Clickometer`, в котором создаются компоненты пользовательского интерфейса:

```
package {
  import mx.controls.*;
  import mx.core.*;
  import flash.events.*;
  import flash.utils.*;

  public class Clickometer {
    private static var lastClickTime:int = 0;
    private static var numClicks:int = 0;
    private static var grid:DataGrid;
    private static var button:Button;

    // Точка входа программы
    public static function main ( ):void {
      // Создаем кнопку
      button = new Button( );
      button.label = "Click Quickly!";
      button.addEventListener(MouseEvent.CLICK, clickListener);
```

```
        // Создаем таблицу данных
        grid = new DataGrid( );
        grid.dataProvider = new Array( );

        // Добавляем отображаемые элементы на экран. Переменная
        // Application.application ссылается на приложение Flex верхнего
        // уровня – основной контейнер для компонентов пользовательского
        // интерфейса и отображаемых элементов.
        var mxmlApp:Application = Application(Application.application);
        mxmlApp.addChild(button);
        mxmlApp.addChild(grid);
    }

    // Этот метод вызывается всякий раз, когда пользователь
    // нажимает кнопку
    private static function clickListener (e:MouseEvent):void {
      var now:int = getTimer( );
      var elapsed:int = now - lastClickTime;
      lastClickTime = now;
      numClicks++;
      grid.dataProvider.addItem({Clicks: numClicks, "Time (ms)": elapsed});
    }
  }
}
```

Приведенный пример приложения можно загрузить по адресу http://www.moock.org/eas3/examples.

Передача кода вашим друзьям

Итак, в этой книге осталась всего одна глава. Что же является последним основополагающим вопросом языка ActionScript? Совместное использование вашего гениального кода с другими программистами. Для совместного использования кода на языке ActionScript можно просто отправить этот особый код в виде одного или нескольких файлов ваших классов. Или, если вы более амбициозны, можете создать целую *библиотеку классов* (то есть группу классов) для включения в приложение на этапе компиляции или на этапе выполнения. В последней главе этой книги рассматриваются способы создания и распространения кода с использованием библиотек классов.

Распространение библиотеки классов

В этой главе рассматриваются три конкретные методики совместного использования группы классов *(библиотеки классов)* в нескольких проектах и несколькими разработчиками. Несомненно, самый легкий способ совместного использования классов заключается в простом распространении исходного кода. Мы рассмотрим этот простейший случай перед тем, как перейти к обсуждению методов совместного использования классов без распространения исходного кода, что может потребоваться при продаже профессиональной библиотеки классов.

Термин «библиотека классов» на жаргоне программистов обозначает произвольную группу классов, распространяемую среди команды разработчиков или по всему миру. Не путайте этот термин с библиотекой FLA-файла или палитрой Library (Библиотека) среды разработки Flash. Эти термины относятся исключительно к среде разработки Flash и не являются частью текущего обсуждения.

В ActionScript распространять библиотеку классов между другими разработчиками можно либо просто в виде набора исходных AS-файлов, либо в виде SWF-файла, либо в виде SWC-файла. В этой главе мы рассмотрим все три подхода. Стоит отметить, однако, что ActionScript предоставляет широкий набор возможностей по распространению библиотек классов; в этой главе описаны три конкретные ситуации, однако представленный материал не является исчерпывающим. Дополнительную информацию по распространению библиотек классов можно найти в следующих разделах документации корпорации Adobe:

- Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Client System Environment ▸ ApplicationDomain class (http://livedocs.macromedia.com/flex/201/html/18_Client_System_Environment_175_4.html);
- Building and Deploying Flex 2 Applications ▸ Building Flex Applications ▸ Using Runtime Shared Libraries (http://livedocs.macromedia.com/flex/201/html/rsl_124_1.html).

Файлы примеров, которые будут рассматриваться в этой главе, доступны по адресу http://www.moock.org/eas3/examples.

Совместное использование исходных файлов классов

Начнем с простейшего способа распространения библиотеки классов: совместного использования исходных файлов классов.

Представьте, что вы работаете в небольшой компании Beaver Code, занимающейся разработкой веб-приложений и имеющей свой сайт http://www.beavercode.com. Вы создали класс `com.beavercode.effects.TextAnimation`, который реализует различные текстовые эффекты. Вы хотите использовать класс `TextAnimation` на двух других сайтах, над которыми работаете в настоящий момент, — Barky Pet Supplies и Mega Bridal Depot. Вместо того чтобы помещать копию файла класса (то есть файл `TextAnimation.as`) в папку каждого проекта, вы сохраняете этот файл класса в центральном хранилище и просто обращаетесь к нему из каждого проекта. Например, для сохранения файла `TextAnimation.as` в операционной системе Windows вы используете следующее местоположение: `c:\data\actionscript\com\beavercode\effects\TextAnimation.as`. Чтобы класс `TextAnimation` был доступен для обоих проектов, вы включаете директорию `c:\data\actionscript` в путь к классам каждого проекта (дополнительную информацию по пути к классам можно найти в гл. 7).

Следуя той же логике, если бы ваша команда состояла из нескольких разработчиков, вы могли бы предположить, что удобно разместить классы на центральном сервере, при этом каждый разработчик мог бы воспользоваться ими, добавив папку сервера в путь к классам своих проектов. Например, вы могли бы сохранить все совместно используемые классы на сервере с именем `codecentral`: `\\codecentral\com\beavercode\effects\TextAnimation.as`. Однако работать непосредственно с сервером крайне опасно и не рекомендуется.

Если вы сохраняете свои классы на центральном сервере и позволяете разработчикам непосредственно изменять их, то могут возникать ситуации, когда один разработчик перезаписывает изменения другого разработчика. Более того, если часы сервера и часы персонального компьютера программиста не синхронизированы, последняя версия класса может быть не включена в программу на этапе компиляции. Чтобы избежать подобных проблем, вы должны всегда использовать программное обеспечение для контроля версий, позволяющее управлять файлами ваших классов при работе в команде. Двумя популярными (и бесплатными!) вариантами такого программного обеспечения являются CVS (http://www.cvshome.org) и Subversion (http://subversion.tigris.org).

Кроме того, при работе с большими проектами вы также можете захотеть автоматизировать процесс экспорта SWF-файла, используя такой инструмент компиляции, как, например, Apache Ant (http://ant.apache.org).

Информацию об использовании инструмента Ant в приложении Flex Builder 2 можно найти в разделе Using Flex Builder 2 ▸ Programming Flex Applications ▸ Building Projects ▸ Advanced build options ▸ Customizing builds with Apache Ant (http://livedocs.macromedia.com/flex/201/html/build_044_12.html).

Автоматизация экспорта SWF-файла в среде разработки Flash заключается в выполнении консольного JSFL-сценария, который сообщает компилятору Flash о необходимости создания SWF-файла для каждого FLA-файла в вашем проекте. Исчерпывающее рассмотрение консольной компиляции в среде разработки Flash выходит за рамки этой книги, однако ниже представлен небольшой пример, который дает общее представление о содержимом этого сценария в операционной системе Windows:

```
// Код в файле exportPetSupplies.jsfl:
// ===============================
// Открываем FLA-файл.
var doc = fl.openDocument("file:///c|/data/projects/pet/petsupplies.fla");
// Экспортируем SWF-файл.
doc.exportSWF("file:///c|/data/projects/pet/petsupplies.swf", true);
// Закрываем среду разработки Flash (не обязательно).
fl.quit(false);

// Команда, вызываемая из командной строки в директории /pet/:
// ================================================
"[папка_среды_разработки_Flash]\flash.exe" exportPetSupplies.jsfl
```

Чтобы предыдущий пример команды был выполнен, среда разработки Flash не должна быть запущена. После вызова команды в директории `c:\data\projects\pet` появится скомпилированный клип `petsupplies.swf`.

Распространение библиотеки классов в виде SWC-файла

Когда над проектом работает команда удаленных разработчиков или библиотека классов публикуется для всего мира, подход с предоставлением непосредственного доступа к файлам классов может оказаться обременительным. Для удобства инструменты языка ActionScript корпорации Adobe предоставляют возможность включения библиотеки классов в один файл, имеющий формат SWC.

В следующих разделах сначала описывается, как создать SWC-файл, содержащий библиотеку классов, а затем рассматривается, как использовать классы из этой библиотеки в приложении.

Создание библиотеки классов в формате SWC в приложении Flex Builder 2

Чтобы продемонстрировать процесс создания SWC-файла, содержащего библиотеку классов, в приложении Flex Builder 2, мы вернемся к примеру с компанией Beaver Code из предыдущего раздела. Наша библиотека классов будет называться `beavercore` и будет помещена в пакет **com.beavercore**. Название пакета соответствует названию фиктивного сайта **http://www.beavercore.com**, который был создан разработчиками компании Beaver Code для размещения библиотеки классов `beavercore`.

Следующие шаги описывают, как создать файл `beavercore.swc`, содержащий библиотеку классов `beavercore`. Для простоты библиотека содержит всего один класс — `com.beavercore.effects.TextAnimation`.

1. В приложении Flex Builder выберите команду меню **File ▸ New ▸ Flex Library Project** (Файл ▸ Создать ▸ Проект библиотеки Flex).

2. В поле Project name (Название проекта) окна New Flex Library Project (Новый проект библиотеки Flex) введите `beavercore`, после чего нажмите кнопку Next (Далее).
3. В поле Main source folder (Основная папка исходных файлов) введите `src` и нажмите кнопку Finish (Готово).
4. Указав в проекте beavercore папку src, выберите команду меню File ▸ New ▸ Folder (Файл ▸ Создать ▸ Папка). В поле Folder name (Имя папки) введите `com`. Повторите этот процесс, чтобы создать следующую структуру папок: `src/com/beavercore/effects`.
5. Указав в проекте beavercore папку effects, выберите команду меню File ▸ New ▸ ActionScript Class (Файл ▸ Создать ▸ Класс ActionScript).
6. В поле Name (Имя) окна New ActionScript Class (Новый класс ActionScript) введите `TextAnimation`, после чего нажмите кнопку Finish (Готово).
7. В файле `TextAnimation.as` введите следующий код:

```
package com.beavercore.effects {
  public class TextAnimation {
    public function TextAnimation ( ) {
      trace("Imagine a text effect with great majesty.");
    }

    public function start ( ):void {
      trace("Effect now starting.");
    }
  }
}
```

8. На палитре Navigator (Навигатор) выберите папку проекта beavercore, а затем меню Project ▸ Build Project (Проект ▸ Скомпилировать проект).

   Обратите внимание, что команда Build Project (Скомпилировать проект) доступна только в том случае, если флажок Project ▸ Build Automatically (Проект ▸ Автоматическая компиляция) снят. Если он установлен, пропустите шаг 8.

В результате выполнения предыдущих шагов приложение Flex Builder 2 создаст файл `beavercore.swc` и поместит его в папку /bin/. Файл содержит классы проекта в скомпилированном виде. В нашем простом примере приложение Flex Builder добавляет все классы из проекта beavercore в файл `beavercore.swc`. В более сложной ситуации мы могли бы указать явно, какие классы должны быть включены в этот файл или исключены из него, выбрав команду меню Project ▸ Properties (Проект ▸ Свойства) и определив список классов на вкладке Classes (Классы) свойства Flex Library Build Path (Путь компиляции библиотеки Flex).

Использование библиотеки классов в формате SWC в приложении Flex Builder 2

Теперь, когда мы создали библиотеку классов в формате SWC (`beavercore.swc`), посмотрим, как использовать ее в проекте.

Предположим, что мы создаем сайт с применением технологии Flash в приложении Flex Builder 2 для компании Barky's Pet Supplies. При разработке сайта мы хотим

использовать класс TextAnimation из библиотеки классов beavercore.swc. Рассмотрим, как это сделать.

1. В приложении Flex Builder выберите команду меню File ▸ New ▸ ActionScript Project (Файл ▸ Создать ▸ Проект ActionScript).
2. В поле Project name (Название проекта) окна New ActionScript Project (Новый проект ActionScript) введите beaver_barkys, после чего нажмите кнопку Next (Далее).
3. В поле Main source folder (Основная папка исходных файлов) введите src.
4. В поле Main application file (Основной файл приложения) введите Barkys.
5. На вкладке Library path (Пути библиотек) нажмите кнопку Add SWC (Добавить SWC).
6. Найдите и выберите файл beavercore.swc из предыдущего раздела, после чего нажмите кнопку Finish (Готово).
7. В файле Barkys.as (который откроется автоматически) введите следующий код:

```
package {
  import flash.display.Sprite;
  import com.beavercore.effects.TextAnimation;

  public class Barkys extends Sprite {
    public function Barkys ( ) {
      var textAni:TextAnimation = new TextAnimation( );
      textAni.start( );
    }
  }
}
```

8. На палитре Navigator (Навигатор) выберите папку проекта beaver_barkys, а затем выберите команду меню Run ▸ Debug Barkys (Выполнить ▸ Отладка Barkys).

В результате выполнения предыдущих шагов компилятор сгенерирует SWF-файл (Barkys.swf), включающий класс TextAnimation, и выполнит этот файл. В окне Console (Консоль) появятся следующие сообщения:

```
Imagine a text effect with great majesty.
Effect now starting.
```

Как вы видите, класс Barkys создает прямую ссылку на класс TextAnimation, будто последний на самом деле является частью проекта beaver_barkys.

Теперь, когда известно, как создать и распространить библиотеку классов в виде SWC-файла с помощью приложения Flex Builder 2, рассмотрим, как проделать то же самое в среде разработки Flash.

Создание библиотеки классов в формате SWC в среде разработки Flash

Следующие шаги описывают, как использовать среду разработки Flash для создания библиотеки классов с именем beavercore.swc, содержащей единственный класс TextAnimation.

1. Создайте новую папку с именем **beavercore** в файловой системе. Папка будет содержать исходные файлы для библиотеки классов.
2. В ней создайте структуру вложенных папок: `com/beavercore/effects`.
3. В папке **effects** создайте новый текстовый файл с именем `TextAnimation.as`.
4. В файле `TextAnimation.as` введите следующий код:

```
package com.beavercore.effects {
  public class TextAnimation {
    public function TextAnimation ( ) {
      trace("Imagine a text effect with great majesty.");
    }
    public function start ( ):void {
      trace("Effect now starting.");
    }
  }
}
```

5. В папке **beavercore** создайте новый текстовый файл с именем `BeaverCore.as`.
6. В файле `BeaverCore.as` введите следующий код. Обратите внимание, что класс `BeaverCore` содержит ссылки на классы (и другие определения), которые мы хотим включить в библиотеку классов.

```
package {
  import com.beavercore.effects.*;
  import flash.display.Sprite;

  public class BeaverCore extends Sprite {
    com.beavercore.effects.TextAnimation;
  }
}
```

7. В среде разработки Flash создайте новый FLA-файл и сохраните его под именем `beavercore.fla` в папке **beavercore**.
8. В поле **Document class** (Класс документа) палитры **Properties** (Свойства) (команда меню **Window** ▸ **Properties** (Окно ▸ Свойства)) введите `BeaverCore`.
9. Выберите команду меню **File** ▸ **Publish Settings** (Файл ▸ Настройки публикации).
10. На вкладке **Formats** (Форматы) в области **Type** (Тип) снимите флажок **HTML**.
11. На вкладке **Flash** в области **Options** (Параметры) установите флажок **Export SWC** (Экспорт SWC).
12. Нажмите кнопку **Publish** (Опубликовать), а затем — кнопку **OK**.

В результате выполнения предыдущих шагов среда разработки Flash сгенерирует файл `beavercore.swc` и поместит его в папку **beavercore**. Этот файл содержит классы в скомпилированном виде.

Использование библиотеки классов в формате SWC в среде разработки Flash

Следующие шаги описывают процесс, который позволит использовать класс `TextAnimation` из библиотеки классов `beavercore.swc` при разработке сайта для компании Barky's Pet Supplies.

1. Создайте новую папку с именем **barkys** в файловой системе. Папка будет содержать исходные файлы для сайта.
2. В папке создайте новый текстовый файл с именем `Barkys.as`.
3. В файле введите следующий код:

```
package {
  import flash.display.Sprite;
  import com.beavercore.effects.TextAnimation;

  public class Barkys extends Sprite {
    public function Barkys ( ) {
      var textAni:TextAnimation = new TextAnimation( );
      textAni.start( );
    }
  }
}
```

4. В среде разработки Flash создайте новый FLA-файл и сохраните его в папке **barkys** под именем `barkys.fla`.
5. В поле **Document class** (Класс документа) палитры **Properties** (Свойства) (команда меню **Window** ▸ **Properties** (Окно ▸ Свойства)) введите `Barkys`.
6. Во вложенной папке **Configuration\Components**, находящейся внутри папки с установленной средой разработки Flash, создайте новую папку с именем **BeaverCode**. В Windows XP местоположением вложенной папки **Configuration\Components** по умолчанию является `C:\Program Files\Adobe\Adobe Flash CS3\en\Configuration\Components`. В Mac OS X местоположением вложенной папки **Configuration\Components** по умолчанию является `Macintosh HD:Applications:Adobe Flash CS3:Configuration:Components`.
7. Скопируйте файл `beavercore.swc`, описанный в предыдущем разделе, в папку **BeaverCode**, которая создана на шаге 6. В результате копирования файла `beavercore.swc` в подпапку **Configuration\Components** на палитре **Components** (Компоненты) среды разработки Flash появится соответствующая запись.
8. В среде разработки Flash откройте палитру **Components** (Компоненты) (команда меню **Window** ▸ **Components** (Окно ▸ Компоненты)).
9. Откройте меню **Options** (Параметры) палитры **Components** (Компоненты), щелкнув кнопкой мыши на значке в правом верхнем углу палитры, и выберите команду **Reload** (Обновить). Папка **BeaverCode** появится на палитре **Components** (Компоненты).
10. На палитре **Components** (Компоненты) откройте папку **BeaverCode**.
11. Откройте библиотеку файла `barkys.fla` (команда меню **Window** ▸ **Library** (Окно ▸ Библиотека)).
12. Перетащите компонент `BeaverCore` с палитры **Components** (Компоненты) в библиотеку файла `barkys.fla`.
13. Выберите команду меню **Control** ▸ **Test Movie** (Управление ▸ Проверка фильма).

В результате выполнения предыдущих шагов компилятор сгенерирует SWF-файл (`Barkys.swf`), включающий класс `TextAnimation`, и выполнит этот SWF-файл. В окне **Output** (Вывод) появятся следующие сообщения:

```
Imagine a text effect with great majesty.
Effect now starting.
```

Обратите внимание на то, что класс `Barkys` создает прямую ссылку на класс `TextAnimation` — таким же образом класс `Barkys` может обращаться к любому классу, доступному в пути.

Распространение библиотеки классов в виде SWF-файла

При работе с несколькими SWF-файлами, которые используют один и тот же класс, компиляция этого класса в каждый SWF-файл будет приводить к потере пространства. Когда размер файла имеет существенное значение, мы можем избежать подобного дублирования, создав библиотеку классов в виде отдельного SWF-файла и загружая его на этапе выполнения. Как только библиотека будет загружена в первый раз, она окажется в кэше компьютера конечного пользователя и может быть использована другими SWF-файлами, не вызывая повторной загрузки.

Процесс создания и использования библиотеки классов в формате SWF гораздо сложнее процесса использования библиотеки классов в формате SWC. Следовательно, вы должны использовать такие библиотеки только в тех случаях, когда необходимо максимально уменьшить размер файла вашего приложения.

В следующих разделах сначала описывается, как создать SWF-файл, содержащий библиотеку классов, а затем рассматривается, как использовать классы из этой библиотеки в приложении.

Создание библиотеки классов в формате SWF в приложении Flex Builder 2

Чтобы продемонстрировать процесс создания SWF-файла, содержащего библиотеку классов, в приложении Flex Builder 2, мы снова вернемся к примеру с библиотекой `beavercore`. Следующие шаги описывают, как создать библиотеку классов с именем `beavercore.swf`, содержащую единственный класс `TextAnimation` (предположим, что мы начинаем с нуля, хотя некоторые из перечисленных шагов повторяются из подразд. «Создание библиотеки классов в формате SWC в приложении Flex Builder 2» предыдущего раздела).

1. В приложении Flex Builder выберите команду меню File ▸ New ▸ Flex Library Project (Файл ▸ Создать ▸ Проект библиотеки Flex).
2. В поле Project name (Название проекта) окна New Flex Library Project (Новый проект библиотеки Flex) введите `beavercore`, после чего нажмите кнопку Next (Далее).
3. В поле Main source folder (Основная папка исходных файлов) введите `src` и нажмите кнопку Finish (Готово).

4. Выбрав в проекте beavercore папку src, выберите команду меню File ▸ New ▸ Folder (Файл ▸ Создать ▸ Папка). В поле Folder name (Имя папки) введите значение com. Повторите этот процесс, чтобы создать следующую структуру папок: src/com/beavercore/effects.
5. Выбрав в проекте beavercore папку effects, выберите команду меню File ▸ New ▸ ActionScript Class (Файл ▸ Создать ▸ Класс ActionScript).
6. В поле Name (Имя) окна New ActionScript Class (Новый класс ActionScript) введите TextAnimation, после чего нажмите кнопку Finish (Готово).
7. В файле TextAnimation.as введите следующий код:

```
package com.beavercore.effects {
  public class TextAnimation {
    public function TextAnimation ( ) {
      trace("Imagine a text effect with great majesty.");
    }

    public function start ( ):void {
      trace("Effect now starting.");
    }
  }
}
```

В приложении Flex Builder 2 отсутствует возможность компиляции SWF-файла непосредственно из проекта библиотеки Flex. Таким образом, мы должны скомпилировать файл beavercore.swf с помощью консольного компилятора mxmlc. Чтобы скомпилировать наши классы в SWF-файл, мы должны создать для него основной класс. В этот основной класс мы поместим ссылки на классы (и другие определения), которые хотим включить в библиотеку классов. Следующие шаги описывают данный процесс для операционной системы Microsoft Windows.

Компиляция SWF-файла с помощью компилятора mxmlc. Указав папку src в проекте beavercore, выберите команду меню File ▸ New ▸ ActionScript Class (Файл ▸ Создать ▸ Класс ActionScript).

1. В поле Name (Имя) окна New ActionScript Class (Новый класс ActionScript) введите Main, после чего нажмите кнопку Finish (Готово).
2. В файле Main.as введите следующий код. В классе Main перечисляются имена всех классов (и определений), которые будут включены в библиотеку классов.

```
package {
  import com.beavercore.effects.*;
  import flash.display.Sprite;

  public class Main extends Sprite {
    com.beavercore.effects.TextAnimation;
  }
}
```

3. Выбрав команду Пуск ▸ Все программы ▸ Стандартные ▸ Командная строка, откройте командную строку операционной системы Windows.

4. Используя командную строку, перейдите в директорию `C:\Program Files\Adobe\Flex Builder 2\Flex SDK\bin`, выполнив следующую команду (стоит отметить, что местоположение компилятора зависит от версии и операционной системы; дополнительные сведения можно найти в документации корпорации Adobe):

   ```
   cd C:\Program Files\Adobe\Flex Builder 2\Flex SDK 2\bin
   ```

5. В командной строке введите следующую команду, после чего нажмите клавишу Enter:

   ```
   mxmlc путь_к_проекту\src\Main.as -output путь_к_проекту\bin\beavercore.swf
   ```

В результате выполнения предыдущих шагов приложение Flex Builder 2 сгенерирует файл `beavercore.swf` и поместит его в папку /bin/. Файл содержит нашу библиотеку классов, и теперь он может быть загружен и использован любым приложением на этапе выполнения. Тем не менее любому приложению, загружающему файл `beavercore.swf`, должен также предоставляться SWC-файл, используемый при проверке типов на этапе компиляции. Чтобы создать его, мы выбираем папку проекта beavercore на палитре Navigator (Навигатор), а затем выбираем команду меню Project ▸ Build Project (Проект ▸ Скомпилировать проект). В результате приложение Flex Builder 2 сгенерирует файл `beavercore.swc` и поместит его в папку /bin/.

Использование библиотеки классов в формате SWF в приложении Flex Builder 2

Теперь, когда мы создали библиотеку классов в формате SWF (`beavercore.swf`), посмотрим, как использовать эту библиотеку в проекте.

Предположим, что мы используем приложение Flex Builder 2 для создания сайта компании Mega Bridal Depot (упомянутого ранее в этой главе) и хотим воспользоваться классом `TextAnimation` из библиотеки классов `beavercore.swf`. Сначала мы создаем проект ActionScript для сайта Mega Bridal Depot и включаем файл `beavercore.swc` в пути внешних библиотек. Затем на этапе выполнения мы загружаем библиотеку классов `beavercore.swf`.

Следующие шаги описывают процесс, который позволит создать проект ActionScript для сайта компании Mega Bridal Depot и включить файл `beavercore.swc` в пути внешних библиотек.

1. В приложении Flex Builder выберите команду меню File ▸ New ▸ ActionScript Project (Файл ▸ Создать ▸ Проект ActionScript).
2. В поле Project name (Название проекта) окна New ActionScript Project (Новый проект ActionScript) введите `beaver_megabridaldepot`, после чего нажмите кнопку Next (Далее).
3. В поле Main source folder (Основная папка исходных файлов) введите `src`.
4. В поле Main application file (Основной файл приложения) введите название файла `MegaBridalDepot`.
5. На вкладке Library path (Пути библиотек) нажмите кнопку Add SWC (Добавить SWC).

6. Найдите и выберите файл `beavercore.swc` из предыдущего раздела, после чего нажмите кнопку **Finish** (Готово).
7. В списке **Build path libraries** (Пути библиотек для компиляции) выделите элемент **Link Type: Merged into code** (Тип связи: встроить в код), а затем нажмите кнопку **Edit** (Редактировать).
8. В списке **Link Type** (Тип связи) окна **Library Path Item Options** (Свойства элемента пути библиотек) выберите значение **External** (Внешняя) и нажмите кнопку **OK**.
9. В окне **New ActionScript Project** (Новый проект ActionScript) нажмите кнопку **Finish** (Готово).

В результате выполнения предыдущих шагов файл `beavercore.swc` будет добавлен в пути внешних библиотек проекта Mega Bridal Depot. По существу, классы и определения из файла `beavercore.swc` будут доступны для проверки типов на этапе компиляции, но не будут скомпилированы в приложение `MegaBridalDepot.swf`. Вместо этого мы должны загружать эти классы на этапе выполнения.

Чтобы загрузить библиотеку классов `beavercore.swf` на этапе выполнения, мы используем метод экземпляра `load( )` класса `Loader`, рассмотренный в гл. 28. При загрузке файла `beavercore.swf` мы импортируем его в домен приложения `MegaBridalDepot.swf`, что позволяет обращаться к классам из файла `beavercore.swf` напрямую, будто они являются частью приложения. Стоит отметить, что обращаться к классам библиотеки `beavercore.swf` в приложении `MegaBridalDepot.swf` мы должны только после завершения операции загрузки (то есть после того, как возникнет событие `Event.INIT` для операции загрузки файла `beavercore.swf`).

Следующий код демонстрирует класс `MegaBridalDepot`, который загружает библиотеку классов `beavercore.swf` на этапе выполнения. Предполагается, что файл `beavercore.swf` был скопирован в ту же папку, где находится приложение `MegaBridalDepot.swf`.

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.system.*;
  import flash.events.*;
  import com.beavercore.effects.TextAnimation;

  public class MegaBridalDepot extends Sprite {
    public function MegaBridalDepot ( ) {
      var libLoader:Loader = new Loader( );
      libLoader.contentLoaderInfo.addEventListener(
                                                  Event.INIT, initListener);
      libLoader.load(
            new URLRequest("beavercore.swf"),
            new LoaderContext(false, ApplicationDomain.currentDomain));
    }

    private function initListener (e:Event):void {
      var textAni:TextAnimation = new TextAnimation( );
```

```
            textAni.start( );
        }
    }
}
```

Стоит отметить, что при использовании описанной операции `libLoader.load( )` необходимо проявлять осторожность, чтобы не загрузить классы библиотеки `beavercore.swf` в собственный домен приложения `beavercore.swf`, как показано в следующем коде:

```
// НЕПРАВИЛЬНО! Этот код загружает классы библиотеки beavercore.swf
// в собственный домен приложения beavercore.swf, в качестве ребенка домена
// приложения системы. В результате к классам библиотеки beavercore.swf
// нельзя обращаться непосредственно из приложения MegaBridalDepot.swf.
libLoader.load(new URLRequest("beavercore.swf"));
```

Следующий код подобным образом по ошибке загружает классы библиотеки `beavercore.swf` в собственный домен приложения, но на этот раз в качестве ребенка домена приложения `MegaBridalDepot.swf`:

```
// НЕПРАВИЛЬНО! Классы загружаются в собственный домен приложения
// beavercore.swf. На этот раз, хотя домен приложения beavercore.swf
// и является ребенком родительского домена приложения MegaBridalDepot.swf,
// код в приложении MegaBridalDepot.swf все равно не может обращаться
// к классам в домене приложения beavercore.swf напрямую. Тот факт, что
// домен приложения MegaBridalDepot.swf является родителем для домена
// приложения beavercore.swf, просто указывает на использование приложением
// beavercore.swf версии любых классов приложения MegaBridalDepot.swf,
// которые определены в обоих файлах.
libLoader.load(new URLRequest("beavercore.swf"),
               new LoaderContext(false,
               new ApplicationDomain(ApplicationDomain.currentDomain)));
```

Дополнительную информацию по доменам приложений можно найти в разделе **Programming ActionScript 3.0 ▸ Flash Player APIs ▸ Client System Environment ▸ Application Domain class** документации корпорации Adobe.

Теперь, когда мы узнали, как создать и распространить библиотеку классов в виде SWF-файла в приложении Flex Builder 2, рассмотрим, как проделать то же самое в среде разработки Flash.

Создание библиотеки классов в формате SWF в среде разработки Flash

К счастью, процесс создания библиотеки классов в формате SWF в среде разработки Flash идентичен процессу создания библиотеки классов в формате SWC, который был рассмотрен ранее, в подразд. «Создание библиотеки классов в формате SWC в среде разработки Flash» предыдущего раздела. На самом деле в результате публикации SWC-файла создается SWC-файл, содержащий библиотеку классов, *и* SWF-файл, содержащий библиотеку классов. SWF-файл помещается в ту же папку, что и SWC-файл.

Например, при публикации нашей предыдущей библиотеки классов `beavercore` в виде SWC-файла среда разработки Flash также автоматически создала файл `beavercore.swf`. Как и SWC-файл, файл `beavercore.swf` был помещен в папку **beavercore**. Следующий раздел описывает, как в приложении использовать библиотеку классов `beavercore.swf`.

Использование библиотеки классов в формате SWF в приложении Flash CS3

Следующие шаги описывают процесс, который позволит использовать класс `TextAnimation` из библиотеки классов `beavercore.swf` на сайте компании Mega Bridal Depot.

1. Создайте новую папку с именем **megabridaldepot** в файловой системе. Папка будет содержать исходные файлы для сайта.
2. В папке **megabridaldepot** создайте новый текстовый файл `MegaBridalDepot.as`.
3. В файле введите следующий код (дополнительные сведения о методиках загрузки, применяемых в следующем коде, можно найти в подразд. «Использование библиотеки классов в формате SWF в приложении Flex Builder 2»):

```
package {
  import flash.display.*;
  import flash.net.*;
  import flash.system.*;
  import flash.events.*;
  import com.beavercore.effects.TextAnimation;

  public class MegaBridalDepot extends Sprite {
    public function MegaBridalDepot ( ) {
      var libLoader:Loader = new Loader( );
      libLoader.contentLoaderInfo.addEventListener(
                                          Event.INIT, initListener);
      libLoader.load(
         new URLRequest("beavercore.swf"),
         new LoaderContext(false, ApplicationDomain.currentDomain));
    }

    private function initListener (e:Event):void {
      var textAni:TextAnimation = new TextAnimation( );
      textAni.start( );
    }
  }
}
```

4. В среде разработки Flash создайте новый FLA-файл и сохраните его в папке **megabridaldepot** под именем `megabridaldepot.fla`.
5. В поле **Document class** (Класс документа) палитры **Properties** (Свойства) (команда меню **Window** ▸ **Properties** (Окно ▸ Свойства)) введите `MegaBridalDepot`.
6. Скопируйте файл `beavercore.swc` в папку **megabridaldepot**.

7. Теперь скопируйте файл `beavercore.swf` в папку megabridaldepot.
8. Выберите команду меню Control ▸ Test Movie (Управление ▸ Проверка фильма).

На шаге 6 описанной процедуры файл `beavercore.swc` включается в путь к классам проекта `megabridaldepot.fla`, обеспечивая доступность классов и определений из библиотеки `beavercore.swc` для проверки типов на этапе компиляции. Стоит отметить, однако, что классы и определения из SWC-файла применяются только для проверки типов и не включаются в экспортируемый файл `megabridaldepot.swf`. Чтобы включить определения из SWC-файла в экспортируемый SWF-файл, добавьте SWC-файл в библиотеку исходного FLA-файла, как было описано ранее, в подразд. «Использование библиотеки классов в формате SWC в среде разработки Flash» предыдущего раздела.

Включение SWC-файла в путь к классам FLA-файла обеспечивает доступность определений из этого SWC-файла для проверки типов на этапе компиляции, но не добавляет эти определения в SWF-файл, экспортируемый из данного FLA-файла. Результат эквивалентен включению SWC-файла в пути внешних библиотек при компиляции проекта с помощью приложения Flex Builder 2 или компилятора mxmlc. Разработчики на языке ActionScript 2.0 должны отметить, что в файле _exclude.xml больше нет необходимости; механизм использования файла _exclude.xml в приложении Flash CS3 не поддерживается.

В результате выполнения предыдущих шагов компилятор сгенерирует SWF-файл `MegaBridalDepot.swf` и выполнит его в режиме Test Movie (Проверка фильма). Файл не включает класс `TextAnimation`. Вместо этого данный класс загружается на этапе выполнения. Как только класс будет загружен, в окне Output (Вывод) появятся следующие сообщения:

```
Imagine a text effect with great majesty.
Effect now starting.
```

На этом наша благородная миссия по распространению кода завершена. И, между прочим, подошла к концу и книга...

Неужели на этом все закончится?

На протяжении 31 главы мы рассмотрели множество различных инструментов и методик программирования, и теперь настал *ваш* черед экспериментировать с ними. Возьмите то, что было изучено, и реализуйте свои собственные идеи и проекты. Не жалейте, если вы потратили большую часть своего времени на изучение языка ActionScript, и не думайте, что эти знания не пригодятся вам в будущем. Большинство концепций, представленных в этой книге, можно найти во многих других языках. Приобретенные знания по ActionScript помогут вам в таких языках, как, например, Java, C++, Perl и Visual Basic.

Программирование — это вид искусства. По существу, оно включает в себя все крушения надежд и энтузиазм, присущие скульптуре, прозе, рисованию или музыке. И оно подчиняется основному закону творчества: нет предела совершенству. Каждый день занимаясь программированием, вы создаете что-то новое и продол-

жаете обучаться. Этот процесс никогда не прекращается. Хотя эта книга подошла к концу, ваше путешествие в мир программирования будет продолжаться до тех пор, пока вы разрабатываете код.

Если попутно вы увидите или сделаете что-то интересное, отправьте мне сообщение на адрес colin@moock.org.

Благодарю вас за то, что вы разделили со мной часть вашего путешествия в мир программирования. В добрый путь!

Окончательная версия программы «Зоопарк»

В этом приложении представлена окончательная версия кода программы Зоопарк», которая рассматривалась в части I. Для добавления графики и интерактивности в программу применяются методики, рассмотренные в части II.

Код для программы «Зоопарк» можно загрузить по адресу http://www.moock.org/eas3/examples.

Обратите внимание, что код в этой версии виртуального зоопарка подвергся структурным изменениям, отражающим реальные шаблоны проектирования. В частности, добавлены два новых класса: `FoodButton`, представляющий простую интерактивную кнопку-текст, и `VirtualPetView`, обеспечивающий графическое представление экземпляров класса `VirtualPet`.

Класс `VirtualZoo` претерпел следующие значительные изменения:

- теперь он создает экземпляр класса `VirtualPetView`, используемый для отображения животного на экране;
- он ожидает, пока экземпляр класса `VirtualPetView` загрузит необходимые изображения, прежде чем приступить к имитации животного.

Класс `VirtualPet` претерпел следующие значительные изменения:

- константы `VirtualPet.PETSTATE_FULL`, `VirtualPet.PETSTATE_HUNGRY`, `VirtualPet.PETSTATE_STARVING` и `VirtualPet.PETSTATE_DEAD` представляют физическое состояние животного;
- переменная экземпляра `petState` хранит текущее физическое состояние животного;
- приемники событий уведомляются об изменениях в физическом состоянии животного посредством события `VirtualPet.STATE_CHANGE`;
- приемники событий уведомляются об изменениях в имени животного посредством события `VirtualPet.NAME_CHANGE`;
- чтобы изменить количество калорий в желудке животного, класс `VirtualPet` использует метод `setCalories( );` при необходимости метод `setCalories( )` изменяет состояние животного с помощью метода `setPetState( );`
- физическое состояние животного изменяется методом `setPetState( )`, который генерирует соответствующее событие `VirtualPet.STATE_CHANGE`;
- для вызова метода `digest( )` класс `VirtualPet` использует объект `Timer` вместо метода `setInterval( );`

- жизненный цикл (переваривание пищи) каждого объекта `VirtualPet` может начинаться и завершаться с помощью методов `start( )` и `stop( )`;
- метод `digest( )` больше не определяет, умрет ли животное, если оно не будет принимать пищу; он делегирует эту ответственность методу `setCalories( )`;
- формальный метод `die( )` деактивирует объекты `VirtualPet`.

Внимательно изучите следующие листинги с комментариями. Затем в качестве упражнения попробуйте добавить в зоопарк второе животное.

В листинге П.1 представлен код для класса `VirtualZoo`, который является основным классом программы.

Листинг П.1. Класс VirtualZoo

```
package {
  import flash.display.Sprite;
  import zoo.*;
  import flash.events.*;

  // Класс VirtualZoo является основным классом приложения. Он расширяет
  // класс Sprite, благодаря чему его экземпляр может быть создан и добавлен
  // в список отображения на этапе запуска программы.
  public class VirtualZoo extends Sprite {
    // Экземпляр класс VirtualPet
    private var pet:VirtualPet;
    // Объект, который будет отображать животное на экране
    private var petView:VirtualPetView;

    // Конструктор
    public function VirtualZoo ( ) {
      // Создаем новое животное и пытаемся присвоить ему имя
      try {
        pet = new VirtualPet("Bartholomew McGillicuddy");
      } catch (e:Error) {
        // Если попытка создать объект VirtualPet вызывает исключение,
        // данный объект не может быть создан. Таким образом, в этом месте
        // кода мы сообщаем о проблеме и создаем новый объект с заведомо
        // допустимым именем.
        trace("An error occurred: " + e.message);
        pet = new VirtualPet("Stan");
      }

      // Создаем объект, который будет отображать животное на экране
      petView = new VirtualPetView(pet);

      // Регистрируем данный объект VirtualZoo для получения уведомления
      // о завершении процесса инициализации
      // отображаемого объекта ("petView")
      petView.addEventListener(Event.COMPLETE, petViewCompleteListener);
    }

    // Обработчик события вызывается после завершения процесса
    // инициализации объекта VirtualPetView (petView)
```

```
    public function petViewCompleteListener (e:Event):void {
      // Добавляем представление в список отображения
      addChild(petView);
      // Начинаем жизненный цикл животного
      pet.start( );
      // Кормим животное
      pet.eat(new Sushi( ));
    }
  }
}
```

В листинге П.2 продемонстрирован код для класса `VirtualPet`, экземпляры которого представляют животных в зоопарке.

Листинг П.2. Класс VirtualPet

```
package zoo {
  import flash.utils.*;
  import flash.events.*;

  // Класс VirtualPet представляет животное в зоопарке. Он расширяет класс
  // EventDispatcher, благодаря чему может являться получателем
  // события в процессе диспетчеризации.
  public class VirtualPet extends EventDispatcher {
    // ==СТАТИЧЕСКИЕ КОНСТАНТЫ==
    // Типы событий, относящиеся к классу VirtualPet (обрабатываемые
    // объектом VirtualPetView, который отображает животное на экране)
    public static const NAME_CHANGE:String  = "NAME_CHANGE";
    public static const STATE_CHANGE:String = "STATE_CHANGE";

    // Состояния, представляющие текущее физическое состояние животного
    public static const PETSTATE_FULL:int     = 0;
    public static const PETSTATE_HUNGRY:int   = 1;
    public static const PETSTATE_STARVING:int = 2;
    public static const PETSTATE_DEAD:int     = 3;

    // ==СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ==
    // Максимальная длина имени животного
    private static var maxNameLength:int = 20;
    // Максимальное количество калорий, которое может иметь животное
    private static var maxCalories:int = 2000;
    // Скорость, с которой животное переваривает пищу
    private static var caloriesPerSecond:int = 100;
    // Имя для животного, используемое по умолчанию
    private static var defaultName:String = "Unnamed Pet";

    // ==ПЕРЕМЕННЫЕ ЭКЗЕМПЛЯРА ==
    // Имя животного
    private var petName:String;
    // Текущее количество калорий в «желудке» животного.
    private var currentCalories:int;
    // Текущее физическое состояние животного
    private var petState:int;
```

```
// Таймер для вызова метода digest( ) на регулярной основе
private var digestTimer:Timer;

// Конструктор
public function VirtualPet (name:String):void {
  // Присваиваем имя этому животному
  setName(name);
  // Даем этому животному половину от максимально возможного количества
  // калорий (полупустой «желудок»).
  setCalories(VirtualPet.maxCalories/2);
}

// Начинает жизненный цикл животного
public function start ( ):void {
  // Вызываем метод digestTimerListener( ) один раз в секунду
  digestTimer = new Timer(1000, 0);
  digestTimer.addEventListener(TimerEvent.TIMER, digestTimerListener);
  digestTimer.start( );
}

// Останавливает жизненный цикл животного
public function stop ( ):void {
  if (digestTimer != null) {
    digestTimer.stop( );
  }
}

// Присваивает имя животному и уведомляет приемники об изменении
public function setName (newName:String):void {
  // Генерируем исключение, если новое имя не является допустимым
  if (newName.indexOf(" ") == 0) {
    throw new VirtualPetNameException( );
  } else if (newName == "") {
    throw new VirtualPetInsufficientDataException( );
  } else if (newName.length > VirtualPet.maxNameLength) {
    throw new VirtualPetExcessDataException( );
  }

  // Присваиваем новое имя
  petName = newName;
  // Уведомляем приемники об изменении имени
  dispatchEvent(new Event(VirtualPet.NAME_CHANGE));
}

// Возвращает имя животного
public function getName ( ):String {
  // Если животному не было присвоено допустимое имя...
  if (petName == null) {
    // ...возвращаем имя, используемое по умолчанию
    return VirtualPet.defaultName;
  } else {
```

```
    // ...в противном случае возвращаем имя животного
    return petName;
  }
}

// Добавляет некоторое количество калорий в желудок животного,
// используя объект Food
public function eat (foodItem:Food):void {
  // Если животное умерло, ничего не делаем
  if (petState == VirtualPet.PETSTATE_DEAD) {
    trace(getName( ) + " is dead. You can't feed it.");
    return;
  }

  // Если пищей является яблоко, проверяем его на наличие червей.
  // Если яблоко червивое, не будем его есть.
  if (foodItem is Apple) {
    if (Apple(foodItem).hasWorm( )) {
      trace("The " + foodItem.getName( ) + " had a worm. " + getName( )
            + " didn't eat it.");
      return;
    }
  }

  // Отображаем отладочное сообщение с информацией об употребленной пище
  trace(getName( ) + " ate the " + foodItem.getName( )
        + " (" + foodItem.getCalories( ) + " calories).");
  // Добавляем калории из пищи в «желудок» животного
  setCalories(getCalories( ) + foodItem.getCalories( ));
}

// Присваивает животному новое количество калорий и при необходимости
// изменяет его состояние
private function setCalories (newCurrentCalories:int):void {
  // При необходимости корректирует значение newCurrentCalories
  // в соответствии с допустимым диапазоном
  if (newCurrentCalories > VirtualPet.maxCalories) {
    currentCalories = VirtualPet.maxCalories;
  } else if (newCurrentCalories < 0) {
    currentCalories = 0;
  } else {
    currentCalories = newCurrentCalories;
  }

  // Определяем количество калорий в желудке животного, в процентах
  // от максимально допустимого количества калорий
  var caloriePercentage:int = Math.floor(getHunger( )*100);

  // Отображаем отладочное сообщение, информирующее о текущем количестве
  // калорий у животного
```

```
    trace(getName( ) + " has " + currentCalories + " calories"
          + " (" + caloriePercentage + "% of its food) remaining.");

    // При необходимости устанавливаем состояние животного в зависимости
    // от изменения в количестве калорий
    if (caloriePercentage == 0) {
      // У животного не осталось пищи. Поэтому, если оно еще не умерло...
      if (getPetState( ) != VirtualPet.PETSTATE_DEAD) {
        // ...деактивируем его
        die( );
      }
    } else if (caloriePercentage < 20) {
      // Животному срочно требуется пища. Устанавливаем его состояние
      // в «ужасно голодный».
      if (getPetState( ) != VirtualPet.PETSTATE_STARVING) {
        setPetState(VirtualPet.PETSTATE_STARVING);
      }
    } else if (caloriePercentage < 50) {
      // Животному требуется пища. Устанавливаем его состояние
      // в «голодный».
      if (getPetState( ) != VirtualPet.PETSTATE_HUNGRY) {
        setPetState(VirtualPet.PETSTATE_HUNGRY);
      }
    } else {
      // Животному не требуется пища. Устанавливаем его состояние
      // в «сытый».
      if (getPetState( ) != VirtualPet.PETSTATE_FULL) {
        setPetState(VirtualPet.PETSTATE_FULL);
      }
    }
  }

  // Возвращает количество калорий в «желудке» животного
  public function getCalories ( ):int {
    return currentCalories;
  }

  // Возвращает количество с плавающей запятой, описывающее количество
  // пищи, оставшееся в «желудке» животного, в процентах
  public function getHunger ( ):Number {
    return currentCalories / VirtualPet.maxCalories;
  }

  // Деактивирует животное
  private function die ( ):void {
    // Останавливаем жизненный цикл животного
    stop( );
    // Переводим животное в состояние «мертвый»
    setPetState(VirtualPet.PETSTATE_DEAD);
    // Отображаем отладочное сообщение, информирующее о смерти животного
```

```
      trace(getName( ) + " has died.");
    }

    // Уменьшает количество калорий животного в соответствии со скоростью
    // переваривания пищи этого животного. Этот метод вызывается
    // автоматически объектом digestTimer.
    private function digest ( ):void {
      trace(getName( ) + " is digesting...");
      setCalories(getCalories( ) - VirtualPet.caloriesPerSecond);
    }

    // Присваивает целое число, представляющее текущее
    // физическое состояние животного
    private function setPetState (newState:int):void {
      // Если состояние животного не изменилось, выходим из метода
      if (newState == petState) {
        return;
      }

      // Присваиваем новое состояние
      petState = newState;
      // Умедомляем приемники об изменении состояния животного
      dispatchEvent(new Event(VirtualPet.STATE_CHANGE));
    }

    // Возвращает целое число, представляющее текущее
    // физическое состояние животного
    public function getPetState ( ):int {
      return petState;
    }

    // Приемник события для объекта Timer, который управляет
    // процессом переваривания пищи
    private function digestTimerListener (e:TimerEvent):void {
      // Перевариваем часть пищи
      digest( );
    }
  }
}
```

В листинге П.3 представлен код для класса `Food`, который является суперклассом для различных видов пищи, поедаемой животным.

Листинг П.3. Класс Food

```
package zoo {
  // Класс Food является суперклассом для различных видов пищи,
  // поедаемой животными.
  public class Food {
    // Хранит количество калорий, которым обладает данный кусок пищи
    private var calories:int;
    // Удобочитаемое название данного куска пищи
    private var name:String;
```

```
    // Конструктор
    public function Food (initialCalories:int) {
      // Сохраняем указанное первоначальное количество калорий
      setCalories(initialCalories);
    }

    // Возвращает количество калорий, которым обладает данный кусок пищи
    public function getCalories ( ):int {
      return calories;
    }

    // Присваивает количество калорий, которым обладает данный кусок пищи
    public function setCalories (newCalories:int):void {
      calories = newCalories;
    }

    // Возвращает удобочитаемое название данного куска пищи
    public function getName ( ):String {
      return name;
    }

    // Присваивает удобочитаемое название данному куску пищи
    public function setName (newName:String):void {
      name = newName;
    }
  }
}
```

В листинге П.4 продемонстрирован код для класса `Apple`, который представляет конкретный вид пищи, принимаемой животным.

Листинг П.4. Класс Apple

```
package zoo {
  // Класс Apple представляет один из видов пищи,
  // принимаемой животным
  public class Apple extends Food {
    // Количество калорий в объекте Apple, если не указано другое
    private static var DEFAULT_CALORIES:int = 100;
    // Хранит информацию о том, является ли
    // данный объект Apple червивым
    private var wormInApple:Boolean;

    // Конструктор
    public function Apple (initialCalories:int = 0) {
      // Если не было указано допустимое количество калорий...
      if (initialCalories <= 0) {
      //...присваиваем данному объекту Apple количество калорий
      //по умолчанию
        initialCalories = Apple.DEFAULT_CALORIES;
      }
      // Вызываем конструктор класса Food
      super(initialCalories);
```

```
      // Случайным образом определяем, будет ли данный объект Apple червивым
      // (50 % вероятность)
      wormInApple = Math.random( ) >= .5;

      // Присваиваем название этому куску пищи
      setName("Apple");
    }

    // Возвращает значение типа Boolean, которое сообщает о том, является ли
    // данный объект Apple червивым
    public function hasWorm ( ):Boolean {
      return wormInApple;
    }
  }
}
```

Наконец, в листинге П.5 продемонстрирован код для класса `Sushi`, который представляет конкретный вид пищи, принимаемой животным.

Листинг П.5. Класс Sushi

```
package zoo {
  // Класс Sushi представляет один из видов пищи, принимаемой животным
  public class Sushi extends Food {
    // Количество калорий в объекте Sushi, если не указано другое
    private static var DEFAULT_CALORIES:int = 500;

    // Конструктор
    public function Sushi (initialCalories:int = 0) {
      // Если не было указано допустимое количество калорий...
      if (initialCalories <= 0) {
      //...присваиваем данному объекту Sushi количество калорий по умолчанию
        initialCalories = Sushi.DEFAULT_CALORIES;
      }
      // Вызываем конструктор класса Food
      super(initialCalories);

      // Присваиваем название этому куску пищи
      setName("Sushi");
    }
  }
}
```

В листинге П.6 продемонстрирован код для класса `VirtualPetNameException`, который представляет исключение, генерируемое в тех случаях, когда указывается недопустимое имя животного.

Листинг П.6. Класс VirtualPetNameException

```
package zoo {
  // Класс VirtualPetNameException представляет исключение, генерируемое
  // в том случае, когда для животного указывается недопустимое имя
  public class VirtualPetNameException extends Error {
    // Конструктор
```

```
    public function VirtualPetNameException (
                          message:String = "Invalid pet name specified.") {
      // Вызываем конструктор класса Error
      super(message);
    }
  }
}
```

В листинге П.7 показан код для класса `VirtualPetExcessDataException`, который представляет исключение, генерируемое в тех случаях, когда указывается слишком длинное имя животного.

Листинг П.7. Класс VirtualPetExcessDataException

```
package zoo {
  // Класс VirtualPetExcessDataException представляет исключение,
  // генерируемое в том случае, когда для животного указывается слишком
  // длинное имя
  public class VirtualPetExcessDataException
                                        extends VirtualPetNameException {
    // Конструктор
    public function VirtualPetExcessDataException ( ) {
      // Вызываем конструктор класса VirtualPetNameException
      super("Pet name too long.");
    }
  }
}
```

В листинге П.8 показан код для класса `VirtualPetInsufficientDataException`, который представляет исключение, генерируемое в тех случаях, когда указывается слишком короткое имя животного.

Листинг П.8. Класс VirtualPetInsufficientDataException

```
package zoo {
  // Класс VirtualPetInsufficientDataException представляет исключение,
  // генерируемое в том случае, когда для животного указывается слишком
  // короткое имя
  public class VirtualPetInsufficientDataException
                                        extends VirtualPetNameException {
    // Конструктор
    public function VirtualPetInsufficientDataException ( ) {
      // Вызываем конструктор класса VirtualPetNameException
      super("Pet name too short.");
    }
  }
}
```

В листинге П.9 продемонстрирован код для класса `VirtualPetView`, обеспечивающего графическое представление экземпляра класса `VirtualPet`.

Листинг П.9. Класс VirtualPetView

```
package zoo {
  import flash.display.*;
  import flash.events.*;
```

```
import flash.net.*;
import flash.text.*;

// Класс VirtualPetView обеспечивает графическое представление экземпляра
// класса VirtualPet. Изображения для животного загружаются на этапе
// выполнения.
public class VirtualPetView extends Sprite {
  // Отображаемое животное
  private var pet:VirtualPet;

  // Контейнер для изображений животного
  private var graphicsContainer:Sprite;

  // Изображения и текст животного
  private var petAlive:Loader;      // Животное в состоянии «живой»
  private var petDead:Loader;       // Животное в состоянии «мертвый»
  private var foodHungry:Loader;    // Значок для состояния «голодный»
  private var foodStarving:Loader;  // Значок для состояния «ужасно
                                    // голодный»
  private var petName:TextField;    // Отображает имя животного

  // Пользовательский интерфейс животного
  private var appleBtn:FoodButton;  // Кнопка для кормления животного
                                    // яблоком
  private var sushiBtn:FoodButton;  // Кнопка для кормления животного суши

  // Определение момента завершения загрузки
  static private var numGraphicsToLoad:int = 4; // Общее количество
                                                // изображений
  private var numGraphicsLoaded:int = 0;  // Количество изображений,
                                          // загруженных на настоящий
                                          // момент

  // Конструктор
  public function VirtualPetView (pet:VirtualPet) {
    // Сохраняем ссылку на отображаемое животное
    this.pet = pet;

    // Регистрируем приемник для получения уведомлений об изменениях
    // в имени животного
    pet.addEventListener(VirtualPet.NAME_CHANGE,
                         petNameChangeListener);
    // Регистрируем приемник для получения уведомлений об изменениях
    // в состоянии животного
    pet.addEventListener(VirtualPet.STATE_CHANGE,
                         petStateChangeListener);

    // Создаем и загружаем изображения животного
    createGraphicsContainer( );
    createNameTag( );
    createUI( );
```

```
    loadGraphics( );
  }

  // Создает контейнер, в который будут помещены изображения животного
  private function createGraphicsContainer ( ):void {
    graphicsContainer = new Sprite( );
    addChild(graphicsContainer);
  }

  // Создает объект TextField, в котором будет отображаться имя животного
  private function createNameTag ( ):void {
    petName = new TextField( );
    petName.defaultTextFormat = new TextFormat("_sans",14,0x006666,true);
    petName.autoSize = TextFieldAutoSize.CENTER;
    petName.selectable = false;
    petName.x = 250;
    petName.y = 20;
    addChild(petName);
  }

  // Создает кнопки для пользователя, предназначенные
  // для кормления животного
  private function createUI ( ):void {
    // Кнопка Feed Apple (Дать яблоко)
    appleBtn = new FoodButton("Feed Apple");
    appleBtn.y = 170;
    appleBtn.addEventListener(MouseEvent.CLICK, appleBtnClick);
    addChild(appleBtn);

    // Кнопка Feed Sushi (Дать суши)
    sushiBtn = new FoodButton("Feed Sushi");
    sushiBtn.y = 190;
    sushiBtn.addEventListener(MouseEvent.CLICK, sushiBtnClick);
    addChild(sushiBtn);
  }

  // Блокирует пользовательский интерфейс
  private function disableUI ( ):void {
    appleBtn.disable( );
    sushiBtn.disable( );
  }

  // Загружает и размещает на экране внешние изображения для животного
  private function loadGraphics ( ):void {
    // Изображение, представляющее животное в состоянии «живой»
    petAlive = new Loader( );
    petAlive.load(new URLRequest("pet-alive.gif"));
    petAlive.contentLoaderInfo.addEventListener(Event.COMPLETE,
                                        completeListener);
    petAlive.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
                                        ioErrorListener);
```

```
    // Изображение, представляющее животное в состоянии «мертвый»
    petDead = new Loader( );
    petDead.load(new URLRequest("pet-dead.gif"));
    petDead.contentLoaderInfo.addEventListener(Event.COMPLETE,
                                               completeListener);
    petDead.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
                                               ioErrorListener);

    // Значок «нужна пища»
    foodHungry = new Loader( );
    foodHungry.load(new URLRequest("food-hungry.gif"));
    foodHungry.contentLoaderInfo.addEventListener(Event.COMPLETE,
                                              completeListener);
    foodHungry.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
                                              ioErrorListener);
    foodHungry.x = 15;
    foodHungry.y = 100;

    // Значок «очень нужна пища»
    foodStarving = new Loader( );
    foodStarving.load(new URLRequest("food-starving.gif"));
    foodStarving.contentLoaderInfo.addEventListener(Event.COMPLETE,
                                               completeListener);
    foodStarving.contentLoaderInfo.addEventListener(IOErrorEvent.IO_ERROR,
                                               ioErrorListener);
    foodStarving.x = 15;
    foodStarving.y = 100;
  }

  // Вызывается при изменении состояния животного
  private function petStateChangeListener (e:Event):void {
    // Если животное умерло...
    if (pet.getPetState( ) == VirtualPet.PETSTATE_DEAD) {
      // ...блокируем кнопки для кормления
      disableUI( );
    }
    // Обновляем графику, чтобы отразить новое состояние животного
    renderCurrentPetState( );
  }

  // Получает текущее состояние животного и отображает его на экране
  private function renderCurrentPetState ( ):void {
    // Удаляем все изображения
    for (var i:int = graphicsContainer.numChildren-1; i >= 0; i--) {
      graphicsContainer.removeChildAt(i);
    }
    // Получаем текущее состояние животного
    var state:int = pet.getPetState( );

    // Отображаем подходящее изображение
    switch (state) {
      case VirtualPet.PETSTATE_FULL:
```

```
            graphicsContainer.addChild(petAlive);
            break;

        case VirtualPet.PETSTATE_HUNGRY:
            graphicsContainer.addChild(petAlive);
            graphicsContainer.addChild(foodHungry);
            break;

        case VirtualPet.PETSTATE_STARVING:
            graphicsContainer.addChild(petAlive);
            graphicsContainer.addChild(foodStarving);
            break;

        case VirtualPet.PETSTATE_DEAD:
            graphicsContainer.addChild(petDead);
            break;
    }
}

// Вызывается при изменениях в имени животного
private function petNameChangeListener (e:Event):void {
    // Обновляем имя животного на экране
    renderCurrentPetName( );
}

// Получает текущее имя животного и отображает его на экране
private function renderCurrentPetName ( ):void {
    petName.text = pet.getName( );
}

// Вызывается при нажатии кнопки «Feed Apple»
private function appleBtnClick (e:MouseEvent):void {
    // Кормим животное яблоком
    pet.eat(new Apple( ));
}

// Вызывается при нажатии кнопки «Feed Sushi»
private function sushiBtnClick (e:MouseEvent):void {
    // Кормим животное суши
    pet.eat(new Sushi( ));
}

// Вызывается при завершении процесса загрузки изображений
private function completeListener (e:Event):void {
    // Увеличиваем (на единицу) общее количество загруженных изображений
    numGraphicsLoaded++;
    // Если все изображения загружены...
    if (numGraphicsLoaded == numGraphicsToLoad) {
        // ...отображаем подходящее изображение, а затем рассылаем событие
        // Event.COMPLETE, которое обозначает, что данный объект
        // VirtualPetView готов к использованию
        renderCurrentPetState( );
```

```
        renderCurrentPetName( );
        dispatchEvent(new Event(Event.COMPLETE));
      }
    }

    // Вызывается при невозможности загрузки изображения
    private function ioErrorListener (e:IOErrorEvent):void {
      // Отображаем отладочное сообщение, описывающее суть проблемы загрузки
      trace("Load error: " + e);
    }
  }
}
```

В листинге П.10 продемонстрирован код для класса `FoodButton`, который представляет простую интерактивную кнопку-текст.

Листинг П.10. Класс FoodButton

```
package zoo {
  import flash.display.*
  import flash.events.*;
  import flash.text.*;

  // Класс FoodButton представляет простую интерактивную кнопку-текст
  public class FoodButton extends Sprite {
    // Отображаемый текст кнопки
    private var text:TextField;
    // Форматирование текста, когда указатель мыши находится не над
    // кнопкой
    private var upFormat:TextFormat;
    // Форматирование текста, когда указатель мыши находится над кнопкой
    private var overFormat:TextFormat;

    // Конструктор
    public function FoodButton (label:String) {
      // Активизирует указатель мыши «рука» для операций с этим объектом
      // (переменная buttonMode наследуется от класса Sprite).
      buttonMode = true;
      // Отключаем события мыши для детей данного объекта
      // (переменная mouseChildren наследуется от класса
      // DisplayObjectContainer.)
      mouseChildren = false;

      // Определяем форматирование текста для ситуации, когда указатель мыши
      // находится не над данным объектом
      upFormat = new TextFormat("_sans",12,0x006666,true);
      // Определяем форматирование текста для ситуации, когда указатель мыши
      // находится над данным объектом
      overFormat = new TextFormat("_sans",12,0x009999,true);

      // Создаем текстовое поле-кнопку и добавляем его в иерархию
      // отображения данного объекта
      text = new TextField( );
```

```
      text.defaultTextFormat = upFormat;
      text.text = label;
      text.autoSize = TextFieldAutoSize.CENTER;
      text.selectable = false;
      addChild(text);

      // Регистрируем приемник для получения уведомлений о перемещении
      // указателя мыши над данным объектом
      addEventListener(MouseEvent.MOUSE_OVER, mouseOverListener);
      // Регистрируем приемник для получения уведомлений о выходе указателя
      // мыши за пределы данного объекта
      addEventListener(MouseEvent.MOUSE_OUT, mouseOutListener);
    }

    // Отключает уведомления о событиях мыши для данного объекта
    public function disable ( ):void {
      // (Переменная mouseEnabled наследуется от класса InteractiveObject.)
      mouseEnabled = false;
    }

    // Вызывается, когда указатель мыши перемещается над данным объектом
    public function mouseOverListener (e:MouseEvent):void {
      // Применяем текстовый формат «под указателем мыши»
      text.setTextFormat(overFormat);
    }

    // Вызывается, когда указатель мыши выходит за пределы данного объекта
    public function mouseOutListener (e:MouseEvent):void {
      // Применяем текстовый формат «не под указателем мыши»
      text.setTextFormat(upFormat);
    }
  }
}
```

Алфавитный указатель

A

B

А

Б

В

Д

Ж

З

И

К

Л

М

Н

О

П

Р

С

T

У

Ф

Ц

Колин Мук

ActionScript 3.0 для Flash. Подробное руководство

Перевел с английского Александр Климович

Заведующий редакцией	*Д. Гурский*
Руководитель проекта	*Ю. Чернушевич*
Ведущий редактор	*Н. Гринчик*
Художник	*С. Шутов*
Корректоры	*Е. Павлович, В. Субот*
Верстка	*О. Махлина*

Подписано в печать 30.09.08. Формат 70×100/16. Усл. п. л. 79,98. Тираж 2000. Заказ № 997.

ООО «Питер Пресс», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано с готовых диапозитивов в ГП ПО «Псковская областная типография».
180004, Псков, Ротная ул., 34.