HIGHTECH

ДЖОИ ЛОТТ и др.

ADOBE AIR

Практическое руководство по среде для настольных приложений Flash и Flex

Символ®

Adobe AIR
in Action

Joey Lott, Kathryn Rotondo
Sam Ahn, Ashley Atkins

MANNING

HIGH TECH

Adobe AIR

Практическое руководство по среде для настольных приложений Flash и Flex

Джои Лотт, Кэтрин Ротондо,
Сэмюел Ан, Эшли Аткинс

Санкт-Петербург — Москва
2009

Серия «High tech»

Джои Лотт, Кэтрин Ротондо,
Сэмюел Ан, Эшли Аткинс

Adobe AIR. Практическое руководство по среде для настольных приложений Flash и Flex

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Научный редактор	*М. Антипин*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Лотт Дж., Ротондо К., Ан С., Аткинс Э.
Adobe AIR. Практическое руководство по среде для настольных приложений Flash и Flex. – Пер. с англ. – СПб.: Символ-Плюс, 2009. – 352 с., ил.
ISBN-10: 5-93286-136-3
ISBN-13: 978-5-93286-136-3

Adobe AIR – кросс-платформенная среда исполнения для развертывания приложений Flash и Flex в качестве настольных или работающих в смешанном сетевом/автономном режиме. Приложения AIR устанавливаются и выполняются локально, поэтому у них есть доступ к файловой системе, что дает им преимущества над веб-приложениями.

Авторы начинают с простых вещей, знакомят с функциями AIR API, а затем показывают, как на практике создаются приложения AIR. Рассматриваются создание и настройка системных окон, а также обмен данными с локальной файловой системой или базой данных. Обсуждается, как AIR подключается к веб-сервисам и как устраняет разрыв между Flex и JavaScript. Книга хорошо иллюстрирована и содержит массу исходного кода, доступного для загрузки из Интернета. Издание предназначено для разработчиков, знакомых с Flash и Flex и стремящихся перейти от броузерных приложений к настольным.

ISBN-10: 5-93286-136-3
ISBN-13: 978-5-93286-136-3
ISBN: 1-933988-48-7 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 30.10.2008. Формат 70x100 1/16 . Печать офсетная.
Объем 22 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Мой друг Пол Ньюмен (да, это его настоящее имя, и нет, это не *тот* Пол Ньюмен) позвонил мне год назад и поинтересовался, не приму ли я участие в написании книги об Apollo (кодовое название Adobe AIR на тот момент). Я был сильно загружен и поэтому согласился лишь после некоторых колебаний. Я имел общее представление об Apollo, но лишь с этого момента я всерьез обратился к этой технологии. Позднее Полу пришлось оставить этот проект ввиду других обязательств, но я продолжал изучать Apollo и готовиться к написанию этой книги.

Ранее у меня существовал ряд предубеждений против Apollo. Я имел десятилетний опыт работы с Flash и Flex и широко пользовался средствами Flash и Flex для создания настольных приложений. Я делал во Flash самостоятельные исполняемые модули практически с момента начала работы с ним. Для встраивания дополнительных функций в настольные приложения, сделанные на Flash, я с разной степенью успешности использовал программы FlashJester, Northcode SWF Studio и Multidmedia Zinc и рассматривал Apollo как еще одну альтернативу этим программам. Честно говоря, меня даже несколько возмущало, что такая огромная корпорация, как Adobe, пытается сокрушить работающие компании, выпустив конкурирующий продукт. Однако, поработав с Apollo, я пришел к выводу, что этот продукт существенно отличается от всех других.

Вскоре Adobe заменила название Apollo на Adobe AIR. AIR дает возможность разработчикам использовать свои навыки программирования на Flash и Flex для создания настольных приложений. В этом отношении он очень похож на упомянутые мной продукты. Однако AIR не создает выполняемые модули для конкретной системы. Приложения AIR требуют исполнительной среды AIR. В этом отношении AIR похож не на такие программы, как Zinc, а на среду выполнения типа Java или .NET. Понимание этого изменило мой подход к AIR.

По прошествии почти года рукопись написана, отредактирована и подготовлена для печати. За это время мы, авторы, много узнали об AIR и постарались поделиться своими знаниями на страницах этой книги. Мы искренне надеемся, что издание окажется полезным читателям и поможет им лучше разобраться в том, как работать с AIR.

Джои Лотт

Об авторах

Джои Лотт (Joey Lott) обладает богатым профессиональным опытом работы с такими технологиями Adobe, как Flex, Flash и ActionScript. Он автор или соавтор «ActionScript Cookbook», «Programming Flash Communication Server», «The Flash 8 Cookbook» и ряда других книг в этой области. Вместе с Сэмом Аном он является компаньоном и основателем «The Morphic Group».

Кэтрин Ротондо (Kathryn Rotondo) – программист в Schematic. Она получила диплом программиста в Harvard Extension School и сертификат по Flash в родайлендской Школе дизайна.

Сэм Ан (Sam Ahn) занимался конструированием и созданием RIA на протяжении нескольких лет, в том числе для таких клиентов, как Pfizer, Wyeth, MINIUSA и Puma. Вместе с Джои Лоттом он является компаньоном и основателем «The Morphic Group», компании по разработке интерактивных приложений, занимающейся в основном приложениями Flash/Flex.

Эшли Аткинс (Ashley Atkins) – старший программист в «Six Red Marbles» с более чем шестилетним опытом разработки на ActionScript. Он занимался как разработкой простых интерактивных обучающих программ, так и проектированием и созданием приложений на Flex и AIR.

Благодарности

Авторы книги хотели бы поблагодарить всех, кто участвовал в выпуске печатного издания, которое вы держите в руках. Это большой список сотрудников Manning, которые помогли в создании книги. Майкл Стивенс (Michael Stephens) первым увидел потенциал издания Adobe AIR, за что мы ему очень признательны. Нермина Миллер (Nermina Miller) была редактором-консультантом этой книги; без нее сроки и качество издания, несомненно, сильно пострадали бы. Это благодаря Нермине у вас в руках находится законченный и связный текст! Мы хотим также сказать спасибо Бенджамину Бергу (Benjamin Berg) за редактирование окончательного варианта рукописи.

Мы выражаем свою благодарность Карен Тегтмейер (Karen Tegtmeyer) за организацию рецензирования рукописи, которое помогло нам улучшить ее содержание и ответить на комментарии первых читателей и коллег-рецензентов. Наличие обратной связи во время написания и редактирования книги было очень ценным для нас. Особой оценки заслуживают наши коллеги-рецензенты Бернард Фаррелл (Bernard Farrell), Райан Стьюарт (Ryan Stewart), Дасти Джуетт (Dusty Jewett), Кристофер Хаупт (Christopher Haupt), Тим О'Хара (Tim O'Hare), Роби Сен (Robi Sen), Майк Клаймер (Mike Clymer), Шон Мур (Sean Moore), Клинт Тредуэй (Clint Tredway), Джереми Андерсон (Jeremy Anderson), Пэтрик Пик (Patrick Peak), Оливер Голдман (Oliver Goldman), Джек Д. Херрингтон (Jack D. Herrington), Натан Левеск (Nathan Levesque), Бруно Лоуаги (Bruno Lowagie), Дэниел Тодд (Daniel Todd) и Брендан Мюррей (Brendan Murray).

Роберт Гловер (Robert Glover) был техническим редактором этой книги, а значит, прочел каждую главу и каждую строчку кода, чтобы гарантировать их точность с технической точки зрения. Нельзя не отметить важность этого этапа, и мы искренне благодарны Роберту за его прекрасную работу. Теплых слов заслуживает также выпускающая команда, работавшая над дизайном книги и обложки и выполнившая набор текста и корректуру: Дотти Марсико (Dottie Marsico), Тиффани Тейлор (Tiffany Taylor), Анна Уэллес (Anna Welles), Лесли Хеймс (Leslie Haimes), Гэбриэл Добреску (Gabriel Dobrescu), Рон Томич (Ron Tomich) и Мэри Пиргис (Mary Piergies).

Хотим также отдать дань уважения команде разработчиков AIR в Adobe за создание замечательного продукта, предоставление отлич-

ной документации и ответы на наши вопросы. Особое спасибо Оливеру Голдману, который не только подробно прорецензировал несколько глав, но и нашел время, чтобы лично ответить на электронные письма, касающиеся технических деталей AIR и цифровых подписей.

Без вас, наших читателей и коллег, не было бы необходимости и потребности в этой книге, поэтому мы хотим поблагодарить и вас за проявленные интерес и увлечение.

Об этой книге

Это книга об Adobe AIR для разработчиков на Flash и Flex. Хотя AIR-приложения можно создавать с помощью HTML и JavaScript, но данная книга посвящена исключительно применению Flash и Flex для создания AIR-приложений. Программные интерфейсы AIR для ActionScript и JavaScript весьма схожи между собой. Однако мы пришли к выводу, что попытка одновременно рассказывать о нюансах как JavaScript, так и Flash и Flex, приведет к бесформенности книги. Поэтому мы решили сосредоточиться только на Flash и Flex.

Возможно, некоторым читателям покажется, что мы и так попытались охватить слишком многое, объединив под одной обложкой Flash и Flex. Такая критика справедлива. Иногда нам приходилось идти на компромисс, включая в одних местах примеры кода, более подходящие для Flash, а в других – более подходящие для Flex. Но это можно расценивать не только как недостаток, но и как достоинство. Нам кажется, что в результате создан более широкий контекст для понимания AIR, что позволяет принимать более правильные решения при создании приложений AIR. В конечном счете вы сами решите, оправдан или нет наш подход, хотя, конечно, мы советуем рассматривать книгу с нашей точки зрения.

Кому адресована эта книга

Естественно, что эта книга рассчитана на разработчиков Flash и Flex, которые хотят воспользоваться своими знаниями при создании AIR-приложений. Разработка на Flash, ActionScript и Flex подробно описывается в десятках, если не сотнях книг. В данной книге мы не предлагаем никакого вводного материала на эту тему. Поэтому, если вы не знакомы с API Flash, Flex или стандартного ActionScript, эта книга может вызвать у вас затруднения. Мы рекомендуем сначала изучить основы Flash и Flex и только потом пытаться строить AIR-приложения. Изложение исходит из того, что у читателя есть подготовка в области Flash и/или Flex.

Структура книги

Книга устроена необычайно просто. Это сравнительно небольшое издание, состоящее всего из восьми глав. Поэтому мы не сочли необходимым

деление на тематические части. В главе 1 читатель знакомится с тем, что представляет собой AIR, а также получает необходимые в дальнейшем базовые сведения. Глава 8 связывает все воедино, рассказывая, как реально строить, развертывать и обновлять AIR-приложения. В каждой из остальных шести глав описывается какая-то одна из логических групп AIR API. Например, в главе 2 рассказывается о доступе к локальной файловой системе, включая чтение и запись файлов.

Представление кода

Книга изобилует примерами кода – от коротких фрагментов до полных приложений. Весь код представлен моноширинным шрифтом, что выделяет его среди остального текста. Кроме того, многие крупные фрагменты кода приводятся в виде нумерованных листингов с заголовками. На эти листинги всегда ссылается близлежащий текст, и они часто снабжены аннотациями.

Загрузка кода

Практически все приведенные в книге листинги можно загрузить с сайта книги на *www.manning.com/AdobeAIRinAction*.

Форум Author Online

Приобретая «Adobe AIR in Action», вы получаете бесплатный доступ к закрытому форуму, организованному Manning Publications, на котором можно комментировать книгу, задавать технические вопросы и получать помощь от авторов и других пользователей. Регистрация в форуме и доступ к нему осуществляются с *www.manning.com/AdobeAIRinAction*. На этой странице рассказано, как попасть на форум после регистрации, какого рода помощь доступна и как вести себя на форуме.

Manning предоставляет читателям место, где может происходить содержательное общение читателей между собой и с авторами книги. При этом не оговорен какой-либо минимальный объем участия со стороны авторов, чей вклад в работу форума является добровольным (и бесплатным). Поэтому мы советуем задавать авторам каверзные вопросы, чтобы их интерес не иссяк!

Форум Author Online и архивы предыдущих дискуссий сохраняются на веб-сайте издательства, пока распространяется книга.

О названии

Объединяя первое знакомство, обзоры и примеры конкретных задач, книги серии «In Action» способствуют обучению и запоминанию. Согласно данным науки о познании лучше всего запоминается то, что открыто во время самостоятельного исследования.

В Manning нет специалистов по когнитивистике, но мы считаем, что знание закрепляется тогда, когда оно проходит через стадии изучения, применения и, что любопытно, пересказа изученного. Понять и запомнить новые вещи, т. е. овладеть ими, можно только после их активного изучения. Люди учатся на практике. Важным аспектом руководств «In Action» является их ориентированность на примеры. Они поощряют читателя к самостоятельным экспериментам с новым кодом и исследованию новых идей.

Есть и другая, более приземленная причина выбора именно такого названия книги: наши читатели – занятые люди. Книги нужны им для того, чтобы выполнить задание или решить проблему. Им нужны такие книги, которые предоставляют удобный доступ к нужной информации и позволяют узнать то, что нужно, и тогда, когда нужно. Для таких читателей и предназначены книги данной серии.

Об обложке

Фигура на обложке «Adobe AIR in Action» – «Провинциальный юрист, депутат от провинций». Иллюстрация взята из путеводителя начала 19 века «L'Encyclopedie des Voyages», напечатанного во Франции. В то время путешествовали ради удовольствия лишь немногие, и такие путеводители были популярны; они знакомили как путешественников, так и тех, кто не покидал своего кресла, с жителями других районов Франции, с ее воинами, чиновниками, аристократами, а также с людьми, населяющими заморские страны.

Разнообразие рисунков, помещенных в «Encyclopedie des Voyages», ярко свидетельствует об уникальности и своеобразии городов и местностей, существовавшем какие-то 200 лет назад. В те времена «дресс-код» двух местностей, располагавшихся на расстоянии всего нескольких десятков километров друг от друга, однозначно определял, откуда человек родом. Этот путеводитель напоминает о существовавшем в ту эпоху ощущении изолированности и удаленности – равно как в любые иные исторические эпохи, исключая нашу нынешнюю, сверхподвижную.

С той поры дресс-коды изменились, и различия по регионам, столь отчетливые в прежние времена, ушли в прошлое. Сейчас бывает даже трудно отличить обитателей одного континента от другого. Если встать на более оптимистическую точку зрения, то мы, возможно, променяли культурное и зрительное разнообразие на более диверсифицированную личную жизнь. Или более диверсифицированную и интересную интеллектуальную и техническую жизнь.

Мы в Manning полагаем, что книжная обложка, отражающая разнообразие местной жизни двухсотлетней давности, возвращенное к жизни картинками из этого путеводителя, подходит для выражения нашего восхищения изощренностью, оригинальностью и прихотливостью компьютерных технологий.

В этой главе:
- Элементы Adobe AIR
- Дескрипторы приложений AIR
- Создание новых проектов AIR
- Компиляция приложений AIR

1

Введение в Adobe AIR

Работа с HTML, Flash, Flex и тысячами других технологий имеет одно общее свойство: все получающиеся приложения используют технологии, предназначенные для Web. Это замечательно, если ваша цель – создать веб-приложение, но никуда не годится, если вам нужно настольное приложение. Интегрированная среда выполнения Adobe (Adobe integrated runtime – AIR) решает эту проблему. Благодаря Adobe AIR вы можете применить свое умение создавать веб-приложения во Flash и Flex (а также HTML и JavaScript) для создания настольных приложений. Это создает заманчивую перспективу.

Каждый полет начинается с подготовки к взлету. Путешествие по Adobe AIR пройдет аналогичным образом. Для начала мы совершим общий обзор AIR, а потом детально изучим применение Flex и Flash для создания AIR-приложений. Мы рассмотрим следующие необходимые базовые понятия, лежащие в основе работы с AIR:

- Различные части Adobe AIR, включая исполнительную систему, средства инсталляции и AIR-приложения, а также существующие между ними связи.
- Проблему безопасности и аутентификации приложений, в том числе цифровые подписи. Вы узнаете, что такое цифровая подпись, какие типы подписей существуют, и чем следует руководствоваться при их выборе.
- Основные этапы создания AIR-приложений с помощью Flex Builder, Flash CS3 или Flex 3 SDK.

Покончив с предисловиями, перейдем к выяснению, в чем, собственно, сущность AIR.

1.1. Анатомия Adobe AIR

Adobe AIR позволяет разработчикам веб-приложений применить имеющиеся у них навыки для создания настольных приложений. Опираясь на свое знание HTML, JavaScript, Flash и Flex, вы можете создавать приложения, которые могут выполняться на настольных системах с помощью исполнительной среды (runtime environment) без необходимости компиляции для конкретных целевых операционных систем. В этом разделе мы определим, что такое исполнительная среда и обсудим, в каких случаях может понадобиться создать настольное приложение. Затем мы расскажем, как могут пригодиться для этого те навыки, которые у вас уже есть.

1.1.1. Разработка приложений для исполнительной среды

Если вы работаете на компьютере под Windows, то, несомненно, вам приходится запускать множество .exe-файлов. Файл .exe представляет собой скомпилированное приложение, которое может подавать команды непосредственно той системе, на которой оно выполняется. Это означает, что .exe-файл (или эквивалентный ему) обладает преимуществом относительной самодостаточности. Однако при этом возникает определенное препятствие, связанное с тем, что необходимо скомпилировать приложение в определенном формате, специфичном для данной платформы. Это означает, что при таком подходе вы должны создать версии вашего приложения «только для Windows» или «только для OS X». Этапы традиционного подхода к созданию приложений следующие:

1. Написать код на выбранном языке.
2. Скомпилировать код в формат, который может непосредственно выполняться в конкретной операционной системе.
3. Запустить скомпилированное приложение.

Более гибкий способ – использовать исполнительную среду вместо того, чтобы компилировать программу для каждой операционной системы. Такой метод исполнительной среды применяется на многих распространенных платформах приложений, включая Java и .NET; он же принят в Adobe AIR. При использовании исполнительной среды процесс создания приложения будет таким:

1. Написать код на выбранном языке.
2. Скомпилировать код в промежуточный формат.
3. Запустить код промежуточного формата в исполнительной среде.

Исполнительные среды дают разработчикам возможность, написав код один раз, выполнять его на любом компьютере, работающем под управлением любой операционной системы, если только в ней установлена требуемая исполнительная среда. Исполнительная среда представляет собой библиотеку, непосредственно выполняемую в данной операцион-

ной системе. Исполнительная среда выступает в качестве посредника для выполняемых в ней программ. Благодаря обеспечению такого уровня абстракции в отношении запускаемых в исполнительной среде программ и операционной системы появляется теоретическая возможность создания для разных компьютеров исполнительных систем, которые будут выполнять одно и то же приложение одинаково на различных платформах.

Какое отношение имеет это все к Adobe AIR? Как уже отмечалось, AIR является исполнительной средой. При создании AIR-приложения вы компилируете его и упаковываете в промежуточный формат, называемый .air-файлом. .air-файл и его содержимое нельзя установить или запустить на компьютере, на котором не была предварительно установлена исполнительная среда AIR. Если среда AIR установлена, файл .air позволяет запустить приложение как на машине с Windows, так и на машине под OS X. Это большое благо для разработчика приложения.

Следует сказать, что у веб-приложений есть преимущества перед обычными настольными приложениями. В каких же случаях тогда вообще может возникнуть необходимость создавать настольные приложения? Раз вы читаете эту книгу, то, вероятно, у вас есть некие основания, а мы приведем свои, представляющиеся нам важными, мотивации.

1.1.2. Зачем нужны настольные приложения?

Веб-интерфейс электронной почты позволяет вам читать свою корреспонденцию с любого компьютера, подключенного к Интернету. Это пример одного из главных достоинств веб-приложений, заключающегося в том, что они не привязаны к конкретной машине. Кроме того, веб-приложения:

- позволяют легко развертывать обновления и новые версии вашего программного обеспечения;
- обычно обеспечивают определенный уровень защиты для пользователей, поскольку на них оказывают влияние средства защиты броузера и плеера (например, Flash Player);
- позволяют распределять вычисления, выполняя их частично на машине клиента, а частично – на сервере.

Однако у веб-приложений есть недостатки. Два самых существенных состоят в том, что они:

- не имеют такого же доступа к функциям операционной системы, как настольные приложения;
- требуют, чтобы компьютер был подключен к Интернету; это недостаток, если нужно работать с приложением, когда вы находитесь в самолете или где-то на природе.

Приложения AIR сочетают в себе лучшие качества как веб-приложений, так и настольных приложений. Поскольку приложения AIR основаны на технологиях веб-приложений, вам (как разработчику)

крайне просто получить доступ к сетевым ресурсам и частично или полностью интегрировать существующие веб-приложения. Но поскольку AIR-приложения выполняются локально, у них есть доступ к системным ресурсам, которого обычно нет у веб-приложений. Это означает доступность таких возможностей, как перетаскивание файлов между AIR-приложениями и файловой системой, обращение к локальным базам данных и – что, может быть, важнее всего – создание эффективных условий работы пользователя при непостоянном подключении к сети – как в режиме онлайн, так и автономно. Приложения AIR также предоставляют функции контроля за появлением обновлений, благодаря которым пользователи всегда могут быть уверены, что работают с самой свежей версией программы (эта тема обсуждается в главе 8).

Другой вопрос, на который хотелось бы получить ответ, – зачем могут понадобиться веб-технологии при создании настольных приложений. Самая очевидная причина – имеющееся у разработчика знание веб-технологий, которое хотелось бы применять разными способами. Если вы сможете создавать настольные приложения с помощью уже имеющихся навыков, вам не придется изучать новый язык и новые технологии, только чтобы сделать настольное приложение. Но есть и другие причины, по которым вы можете захотеть создавать настольные приложения с помощью веб-технологий. Веб-технологии лучше всего приспособлены для создания приложений, которые используют сетевые ресурсы. В условиях, когда настольным приложениям все чаще требуется поддержка интерактивности и доступ к Интернету, предпочтительнее создавать такие приложения с помощью языков, специально разработанных для работы в сети. Другое преимущество применения HTML, JavaScript, Flash и Flex для разработки настольных приложений состоит в том, что эти языки обычно значительно превосходят традиционные языки в возможностях создания запоминающихся, привлекательных и интересных интерфейсов пользователя.

1.1.3. Изучаем возможности AIR

AIR предоставляет разработчикам веб-приложений множество замечательных возможностей для создания настольных приложений. Но что именно вас ждет? Сейчас мы представим основные возможности, предоставляемые AIR. Подробности вы узнаете дальше, по мере чтения книги.

Все, что позволено при создании веб-приложений, позволено также при создании AIR-приложений. Это происходит благодаря включению в AIR движка WebKit (того же, что применен в броузере Safari) и Flash Player. Поэтому можно пользоваться теми же базовыми функциями ActionScript и JavaScript, что и при развертывании веб-приложений. Кроме того, становится доступен специфический API для AIR. В него входят функции, приведенные в табл. 1.1.

Таблица 1.1. Категории функций API, специфичные для AIR

Функция	Описание
Интеграция с файловой системой	AIR разрешает системные операции с файловой системой, включая чтение, запись и удаление.
Перетаскивание (drag-and-drop)	Пользователи могут перетаскивать файлы и каталоги из операционной системы в приложение AIR.
Копирование и вставка	Пользователи могут применять функции копирования-вставки операционной системы для копирования данных между приложениями AIR и операционной системой.
Локальные базы данных	AIR-приложения могут создавать локальные базы данных и подключаться к ним.
Аудио	Приложения AIR на основе HTML легко могут использовать звук.
Встроенный HTML	Приложения AIR на основе Flex и Flash могут показывать в своих объектах HTML и JavaScript.

В AIR-приложениях поддерживается доступ ко всем этим функциям. Однако для их использования AIR-приложения должны выполняться в исполнительной среде, которая их поддерживает. В следующем разделе мы посмотрим, как выполнять приложения AIR.

1.2. Выполнение AIR–приложений

Для создания приложения AIR используется набор инструментов AIR, в качестве которого может выступать Flex Builder 3, AIR SDK и т. п., после чего файлы приложения упаковываются в файле .air. Подробнее об упаковке в файле .air вы узнаете в разделе 1.4 этой главы. Пока вам достаточно знать, что файл .air – это единственный файл, который нужно передать тому, кто захочет установить ваше приложение. Наряду с термином «.air-файл» используется термин «установочный файл» (installer file).

Имея на руках .air-файл, вы можете передать его всем, у кого на компьютере уже есть исполнительная среда AIR, и они смогут легко установить вашу программу. Когда у пользователя установлена среда AIR, ему достаточно сделать двойной щелчок по .air-файлу, который он получил от вас или загрузил из Интернета.

Если на компьютере пользователя нет AIR, ему придется сначала установить эту среду. Для установки AIR есть два пути:

- *Ручная установка* – осуществляется путем загрузки программы установки для конкретной ОС (Windows или OS X) и ее запуска.
- *Автоматическая установка* – требует от разработчика публикации в сети файла .swf (называемого «значком» – badge), при щелчке по которому происходит установка вашего приложения. Если

у пользователя уже стоит AIR, у него сразу установится ваше приложение. Если нет – он сможет сначала установить AIR.

Примечание

Подробнее о распространении приложений AIR, в том числе об автоматической установке, можно узнать в главе 8.

Каким бы образом пользователь ни начал устанавливать приложение AIR – двойным щелчком по присланному ему файлу .air или щелчком по значку установки на веб-странице, – перед ним предстанет ряд стандартных экранов помощника установки. На рис. 1.1 приведен пример того, как может выглядеть первый этап. После установки приложения AIR на машине пользователя его можно запустить, как всякое другое приложение, – выполнив двойной щелчок по значку приложения на рабочем столе или выбрав его в меню.

Выяснив, как запускать AIR-приложения, можно перейти к вопросу их создания.

Рис. 1.1. *При установке приложения AIR отображается экран установки AIR с информацией о приложении и его авторе*

1.3. Безопасность и аутентичность приложений AIR

Мы были бы не правы, если бы при знакомстве с Adobe AIR упустили две связанных между собой проблемы: безопасности и аутентичности. Это важные для разработчика приложения вопросы, поскольку связанные с ними упущения или ошибки могут иметь пагубные последст-

вия. Поэтому необходимо знать, какие средства защиты и проверки подлинности приложений предоставляет AIR, и каковы должны быть ваши действия для защиты пользователей ваших приложений.

1.3.1. Безопасность приложений AIR

Один из флагманских продуктов Adobe – Flash Player, успех которого, в частности, определяется чрезвычайными мерами, предпринятыми Adobe (а до того Macromedia), чтобы гарантировать невозможность умышленного или неумышленного нанесения разработчиками Flash-приложений вреда компьютеру пользователя. У Flash Player есть множество функций безопасности, нацеленных на защиту пользователей. Благодаря им пользователь не должен испытывать беспокойство при просмотре Flash-содержимого в сети

Приложения AIR являются настольными, а потому они должны иметь больше возможностей доступа к компьютерной системе пользователя, чем сетевые Flash-приложения. Хотя приложения AIR также выполняют Flash-контент, у *этого* Flash-контента больше возможностей нанести урон системе пользователя, чем у контента Flash, находящегося в сети. Это определенный компромисс: расширение набора функций влечет за собой повышение риска.

Приложения AIR выполняются с помощью посредника – исполнительной среды. Поэтому Adobe может в значительной мере управлять тем, что позволено или не позволено делать приложению AIR. Однако, несмотря на то, что среда исполнения в значительной мере уменьшает риски, AIR предоставляет своим приложениям значительно больше привилегий, чем могло быть доступно соответствующим веб-аналогам.

Первое, что должен осознать разработчик AIR-приложений, – это необходимость проявить уважение к пользователям своего продукта, серьезно относясь к проблемам безопасности. Например, важно тщательно следить за всеми параметрами, получаемыми кодом, который выполняется в вашем приложении. Не разрешайте пользователям вводить произвольные значения и не допускайте, чтобы динамические, полученные из сети значения участвовали в качестве параметров для кода, который осуществляет такие операции, как, скажем, обращение к файловой системе. Существует подробный документ от Adobe по проблемам безопасности, который находится по адресу: download.macromedia.com/pub/labs/air/air_security.pdf.

1.3.2. Гарантии аутентичности приложения

Чтобы избавить от тревог пользователей вашего приложения, Adobe требует наличия цифровой подписи у каждого приложения AIR. (Заметьте, что подпись требуется только при создании инсталлятора, а разработку и тестирование своих приложений вы можете проводить без всяких подписей.). Цифровая подпись дает пользователю возможность проверить два свойства приложения: подлинность и целостность.

Цифровая подпись моделирует обычную подпись чернилами на листе бумаги, подтверждая подлинность издателя приложения (аутентичность), и отсутствие в приложении изменений, сделанных после публикации (целостность). Слежение среды AIR за целостностью можно проиллюстрировать простым примером. Можете сами убедиться в том, что исполнительная среда AIR откажется устанавливать модифицированный .air-файл. Для этого вам понадобятся файл .air и утилита zip. Файл .air имеет формат архива, который может прочесть любая утилита zip. Сделайте следующее:

1. Запустите файл .air и убедитесь, что сначала среда AIR предложит вам установить приложение. Не нужно щелкать по кнопке Install, когда она появится в помощнике установки. Достаточно проверить, что среда AIR предлагает вам возможность установки.
2. Щелкните по кнопке Cancel и выйдите из помощника установки.
3. С помощью утилиты zip добавьте в архив какой-нибудь файл. Годится любой файл: можете создать пустой текстовый файл и добавить его в архив. Если вы работаете в Windows, проще всего поменять расширение имени файла с .air на .zip, перетащить текстовый файл в архив .zip и затем вернуть прежнее расширение .air.
4. Запустите файл .air. На этот раз вы получите сообщение об ошибке, говорящее о том, что файл .air поврежден и не может быть установлен.

Приложения AIR поставляются с цифровыми подписями и с цифровыми сертификатами. Есть два основных вида сертификатов: подписанные самостоятельно и выпущенные сертифицирующими органами. У тех и других есть свои достоинства и недостатки.

Преимущество подписанных самостоятельно сертификатов в том, что их просто изготовить. Обновление Flash CS3 AIR и Flex 3 SDK (а затем и Flex Builder 3) предлагают средства для изготовления сертификатов ваших приложений AIR, заверенных личной подписью. Подробнее о том, как изготавливать такие сертификаты, будет рассказано далее в этой главе.

Самостоятельно подписанные сертификаты предоставляют пользователям определенный уровень безопасности, поскольку проверяют целостность приложения. Однако они практически бесполезны для проверки личности того, кто опубликовал приложение. Это примерно то же, что самому нотариально заверять подлинность своих документов. Поэтому, когда сертификат подписан самим изготовителем, помощник установки указывает, что его личные данные не известны. Очевидно, это недостаток, поскольку пользователь не может чувствовать себя в безопасности, и он будет менее склонен устанавливать приложение, когда производитель неизвестен, чем когда личные данные производителя можно проверить. Сертифицирующий орган – это организация, выпускающая сертификаты и действующая в качестве независимой стороны при проверке ваших личных данных.

Сертифицирующий орган выдает сертификаты только после проверки ваших личных данных, обычно на основании таких документов, как выпущенное государством удостоверение личности. Преимущество такого официально заверенного сертификата в том, что он дает больше уверенности в ваших подлинных данных, чем подписанный самостоятельно. Когда сертификат выпущен сертифицирующим органом, помощник установки Adobe показывает данные этого документа как данные издателя программы. С другой стороны, очевидны некоторые неудобства: получать сертификат у уполномоченного органа дольше и труднее, чем подписывать самостоятельно. Кроме того, сертифицирующие органы обычно берут плату за свои услуги. (На момент написания книги самый дорогой сертификат для подписи кода AIR-приложений стоил 299 долл. США.)

Два наиболее известных издателя сертификатов – это VeriSign (*www.verisign.com*) и thawte (*www.thawte.com*), хотя формально владельцем thawte теперь является VeriSign. Если вы хотите обеспечить высший уровень сертификации для своего приложения AIR, вам нужно приобрести сертификат в одном из этих центров. Вам потребуется так называемый сертификат для подписи кода (codesigning certificate). Подробнее об условиях приобретения сертификатов можно узнать на вебсайтах центров сертификации.

Примечание

Помимо VeriSign и thawte, существуют другие сертифицирующие организации, – в том числе некоммерческие, такие как CAcert.org, – которые могут предоставить сертификат для подписи программы. Перед приобретением того или иного сертификата вам следует провести собственные изыскания (CAcert.org заставит вас немало побегать, прежде чем выдать вам сертификат разработчика) и убедиться, что выбранный сертификат будет принят на большинстве компьютеров. Если сертификат не будет принят, то приложение AIR по-прежнему будет оцениваться как выпущенное неизвестным автором. Если вы не до конца определились с выбором, поговорите с кем-нибудь в организации, выдающей сертификаты, и задайте интересующие вас вопросы.

Приступая к созданию AIR-приложений, вы, возможно, не захотите тратить средства на покупку сертификата только для того, чтобы собрать несколько примеров и разослать инсталляторы своим друзьям. Не забывайте, что сертификат нужен только тогда, когда вы хотите создать инсталлятор. Тестировать свои приложения AIR можно и без сертификата. Однако, когда вы будете готовы создать для своего приложения файл .air, нужно продумать, какую цифровую подпись выбрать для приложения. Связать сертификат с приложением можно только один раз. Это означает, что нельзя сначала воспользоваться сертификатом с собственной подписью, а потом заменить его на сертификат, выданный сертифицирующим центром. Если вы сделаете новую подпись с другим сертификатом, то пользователи более старых версий вашего приложения не смогут его обновить.

1.4. Создание приложений AIR

Теперь, когда вы знаете о том, что такое AIR, из чего состоит AIR, как запускать приложения AIR и как обеспечиваются безопасность и аутентичность приложений AIR, почти все готово для создания приложения AIR. Фактически именно об этом будет рассказано в нескольких последующих разделах этой главы. Вы даже сможете построить несколько простых приложений AIR, чтобы прочувствовать, что вас ожидает в дальнейшем, после чтения книги. Прежде чем бросаться в неизведанное, мы потратим некоторое время на изучение предстоящего маршрута, чтобы дать вам представление о том, что вас ожидает.

Способов создания приложений AIR много. В табл. 1.2 приведено краткое описание набора инструментов.

Таблица 1.2. Инструменты Adobe AIR

Название	Тип исходного кода приложения AIR	Бесплатный	Описание
Flex Builder 3	Flex/ActionScript	Нет	Коммерческое средство для создания веб- и AIR-приложений на базе Flex. Сам инструмент построен на базе Eclipse. Flex Builder 3, автоматизирует и упрощает создание приложений AIR.
Flex 3 SDK	Flex/ActionScript	Да	Бесплатный SDK, включающий в себя все компиляторы и инструменты Flex Builder 3, но без средств автоматизации и графического интерфейса пользователя Flex Builder.
Flash CS3 с обновлением AIR	Flash/ActionScript	Нет	Flash CS3 поставляется без средств AIR. Однако бесплатное обновление для Flash CS3 позволяет строить приложения AIR прямо из среды разработки Flash.
Dreamweaver CS3 с расширением AIR	HTML/JavaScript	Нет	Коммерческий редактор HTML с расширением AIR, которое в значительной мере автоматизирует создание приложений AIR.
AIR SDK	HTML/JavaScript	Да	Бесплатный SDK со всеми инструментами командной строки, нужными для создания приложений AIR на базе HTML/JavaScript.

Для полноты картины мы включили в табл. 1.2 также инструменты HTML/JavaScript для создания приложений AIR. Однако, повторимся, в этой книге мы фокусируем внимание исключительно на применении Flex и Flash в качестве средств создания приложений AIR. В разделах 1.6, 1.7 и 1.8 вы сможете подробнее узнать, как создавать приложения AIR с помощью Flex Builder, Flash и Flex SDK, соответственно.

Примечание

Все инструменты AIR можно загрузить с сайта Adobe (*www.adobe.com/go/air*).

В разделе 1.5 вы узнаете о дескрипторах приложений. Понимание дескрипторов – важный элемент для получения общей картины создания приложений AIR. Несмотря на то, что многие инструменты AIR (Flex Builder и Flash CS3 с обновлением AIR) автоматически создают файл дескриптора для проекта, все же полезно ознакомиться с тем, как выглядит дескриптор и что в нем содержится. Рекомендуем целиком прочесть раздел 1.5, прежде чем переходить к 1.6, 1.7 или 1.8. Однако если вам не терпится поскорее начать строить приложение и вы проскочите вперед, мы никому об этом не скажем.

Сейчас мы перейдем к дескрипторам приложений. После прочтения следующего раздела переходите к разделу, в котором обсуждается тот набор инструментов, с помощью которого вы будете строить свои приложения AIR.

1.5. Знакомство с дескрипторами приложений AIR

Независимо от того, с помощью каких инструментов вы строите приложение AIR, вам необходимо создать дескриптор приложения. Некоторые инструменты автоматически генерируют базовый дескриптор, но необходимо разбираться в том, что собой представляет этот дескриптор и как им пользоваться. Дескрипторы приложений AIR являются файлами XML, в которых описываются приложения AIR. При создании из приложения AIR пакета для распространения в дескриптор помещаются некоторые сведения, необходимые инструментам AIR для правильной сборки приложения. В состав этих сведений входят, в частности, уникальный идентификатор приложения, номер версии и те данные, которые показываются в процессе инсталляции. Ниже приводится пример того, как может выглядеть простой файл дескриптора. Обратите внимание, что все файлы дескрипторов начинаются с объявления XML (`<?xml version="1.0" encoding="utf-8" ?>`).

```
<?xml version="1.0" encoding="utf-8" ?>
<application xmlns="http://ns.adobe.com/air/application/1.0.M4">

   <id>com.manning.books.airinaction.Example</id>

   <version>1.0</version>
```

```
    <filename>ExampleApplication</filename>

    <initialWindow>
       <content>ExampleMain.swf</content>
    </initialWindow>

</application>
```

Если вы хотите сразу начать строить приложение AIR, можете приступать. Приведенного примера вам будет достаточно для создания элементарного приложения AIR. Если сейчас вы проскочите вперед, вы сможете позже вернуться к этому разделу и более подробно ознакомиться с дескрипторами.

В следующих ниже разделах детально описываются элементы, составляющие файл дескриптора.

1.5.1. Элемент application

Элемент `application` является обязательным, и это корневой элемент файла дескриптора. Элемент `application` требует атрибута `xmlns`. Атрибут `xmlns` определяет пространство имен дескриптора. Значение пространства имен всегда заранее определено, и для каждого приложения, которое вы создаете для определенной версии AIR, это значение всегда одно и то же. Для AIR 1.0 значением пространства имен должно быть `http://ns.adobe.com/air/application/1.0.M4`. Пространство имен указывает, какая версия AIR необходима для выполнения приложения. Каждая новая версия AIR использует новое пространство имен.

Дополнительно можно задать атрибут `minimumPatchLevel`. С его помощью можно потребовать, чтобы пользователь установил определенное обновление среды AIR, прежде чем запускать ваше приложение. Этот атрибут не обязателен. Пользоваться им следует только тогда, когда вам известно, что для корректной работы вашего приложения необходимо определенное обновление.

Поскольку элемент `application` является корневым в файле дескриптора, все последующие элементы оказываются дочерними для него. Единственными обязательными элементами являются следующие четыре: `id`, `version`, `filename` и `initialWindow`.

1.5.2. Элемент id

Элемент `id` служит уникальным идентификатором приложения. В каждый данный момент в системе может быть установлено только одно приложение с данным идентификатором. Идентификатор приложения составлен из идентификатора издателя (который берется из сертификата, применяемого при публикации файла .air) и значения элемента `id`. Это означает, что, строго говоря, значение элемента `id` должно быть уникальным только среди всех приложений данного издателя. Хотя это не является абсолютно необходимым, но мы считаем удобным применять глобально уникальные `id`, использующие извест-

ный принцип обратных доменных имен. В приводившемся примере простого дескриптора таким идентификатором является `com.manning.books.air.Example`. Глобальную уникальность здесь обеспечивает `com.manning`, что является обратным от `manning.com`. Длина `id` может составлять от 1 до 212 символов, которыми могут быть только буквы, цифры, точки и дефисы.

1.5.3. Элемент version

Элемент `version` позволяет задать номер версии вашего приложения. AIR никак не интерпретирует номер версии приложения, но с его помощью вы можете программным образом проверить, что у пользователя стоит новейшая версия вашего приложения. Поскольку AIR не пытается каким-либо образом обрабатывать номер версии, можно воспользоваться любым строковым значением. Обычно версии обозначаются числами, например, 1.0 или 2.5.1, но могут включать в себя и буквы, обозначающие номер ревизии, например, 4.0a.

1.5.4. Элемент filename

В элементе `filename` указывается имя .air-файла. Значение имени файла используется также в качестве имени приложения (во время инсталляции), если не задан элемент `name`.

Элемент `filename` должен содержать только символы, допустимые в именах файлов, и не должен включать в себя расширение. Значение `filename` не должно также оканчиваться точкой.

1.5.5. Элемент initialWindow

Элемент `initialWindow` сообщает о том, какой фактический контент (файл .swf или .html) нужно использовать для сборки приложения. Элемент `initialWindow` служит контейнером для дополнительных элементов. Единственным обязательным дочерним элементом является content, указывающий, какой файл .swf (или .html) нужно использовать. Ниже показан простейший элемент `initialWindow`:

```
<initialWindow>
   <content>ExampleMain.swf</content>
</initialWindow>
```

Дополнительно элемент `initialWindow` может содержать следующие необязательные элементы:

- `systemChrome` – это значение указывает, должно ли окно, в котором показывается приложение, использовать оформление (рамку и панель заголовка), установленные в операционной системе. Если задать значение `standard`, будет использовано стандартное системное оформление. Если задать `none`, стандартное оформление системы использовано не будет. Для приложений AIR на базе Flex компоненты Flex имеют специальный вид, когда атрибут `systemChrome` имеет значение `none`.

- transparent – это булево значение указывает, должно ли происходить альфа-смешение окна приложения с рабочим столом (т. е. можно ли видеть рабочий стол через полупрозрачное окно). Если задать значение true, то можно пользоваться альфа-эффектами, но помните, что при этом потребуется больше ресурсов и отображение приложения может замедлиться. Кроме того, если вы хотите задать для transparent значение true, то нужно задать для systemChrome значение none.
- visible – это булево значение указывает, должно ли окно приложения быть видимым с самого начала. Обычно для этого атрибута указывается значение false, когда вы не хотите показывать окно, пока программным образом не зададите в коде самого приложения место и размеры окна. Затем вы можете программно управлять видимостью окна.
- height – высота окна приложения в пикселах.
- width – ширина окна приложения в пикселах.
- minimizable, maximizable, resizable – эти элементы позволяют указать, можно ли во время выполнения приложения минимизировать, максимизировать и изменять размеры его окна. По умолчанию имеют значения true.
- x, y – координаты x и y начального положения окна.
- minSize, maxSize – минимальный и максимальный допустимый размер окна при попытке изменить его размер.

Вот пример элемента initialWindow, для которого установлено большинство этих значений:

```
<initialWindow>
   <content>ExampleMain.swf</content>
   <systemChrome>none</systemChrome>
   <transparent>true</transparent>
   <height>500</height>
   <width>500</width>
   <minimizable>false</minimizable>
   <maximizable>false</maximizable>
   <resizable>false</resizable>
   <x>0</x>
   <y>0</y>
</initialWindow>
```

Как уже было сказано, единственным обязательным элементом initialWindow является content. Все другие можно опустить, и тогда будут использованы значения по умолчанию.

1.5.6. Элемент name

Элемент name находится на одном уровне с initialWindow, т. е. является дочерним для тега application. Значение name определяет каталог, в ко-

тором приложение устанавливается по умолчанию. Также, значение `name` отображается в заголовке окна приложения во время его работы. Кроме того, `name` появляется на первом экране инсталлятора, как видно на рис. 1.1. Если значение `name` не указано, вместо него используется значение `filename`.

1.5.7. Элементы title и description

Элементы `title` и `description` – одного ранга с `initialWindow`, т. е. должны быть дочерними для тега `application`. Оба они не обязательны и задают строки, показываемые инсталлятором. Элемент `title` определяет, что показывается в заголовках инсталлятора, как на рис. 1.1 и 1.2. Строка `description` выводится на втором экране инсталлятора, как на рис. 1.2.

Элементы `title` и `description` участвуют только в установке и никогда не появляются в работающем приложении.

1.5.8. Элемент installFolder

Элемент `installFolder` является необязательным и определяет название подкаталога по умолчанию, в который устанавливается приложение. Пользователь всегда может изменить этот каталог во время установки приложения AIR. Однако с помощью `installFolder` можно задать подкаталог, который будет частью значения по умолчанию, как на рис. 1.2.

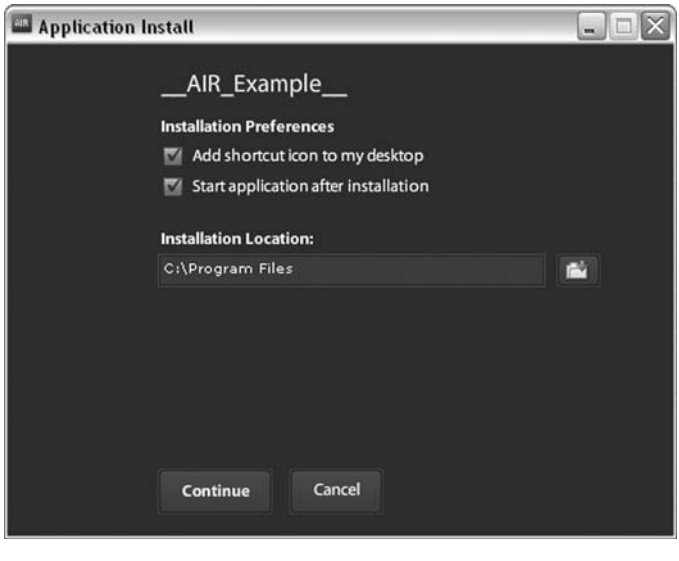


Рис. 1.2. *Второй экран инсталлятора приложения AIR, на котором пользователь может задать параметры установки*

Примечание

Нельзя изменить имя главного каталога в каталоге по умолчанию, используемого приложениями AIR.

В Windows этим каталогом всегда будет Program Files на главном диске; в OS X это всегда /Applications. Однако элемент `installFolder` позволяет изменить используемый подкаталог. Например, если задать для `installFolder` значение `ExampleInc/ExampleApplication`, то на машине под Windows приложение будет установлено в `Program Files\ExampleInc\ExampleApplication`, а на машине под OS X приложение будет установлено в `/Applications/ExampleInc/ExampleApplication`.

Элемент `installFolder` должен быть дочерним для тега `application`. Это необязательный элемент. Если опустить его, то приложение будет установлено в подкаталог, определяемый значением элемента `filename`.

1.5.9. Элемент programMenuFolder

Элемент `programMenuFolder` используется только в Windows и игнорируется другими операционными системами. Этот элемент позволяет задать имя папки, из которой берется ярлык программы для пункта меню All Programs в меню Start.

1.5.10. Элемент icon

По умолчанию приложения AIR используют стандартные значки AIR для размещения на рабочем столе, в меню Start, панели задач и т. д. Однако можно задать свои значки, использовав для этого элемент `icon` в XML-файле дескриптора. У элемента `icon` должно быть четыре дочерних элемента с именами `image16x16`, `image32x32`, `image48x48` и `image128x128`. В каждом из этих дочерних элементов прописывается путь к графическому файлу. Сами графические файлы должны иметь формат .png и быть включены в приложение AIR. Вот пример элемента `icon`:

```
<icon>
   <image16x16>icon16.png</image16x16>
   <image32x32>icon32.png</image32x32>
   <image48x48>icon48.png</image48x48>
   <image128x128>icon128.png</image128x128>
</icon>
```

Если ваши значки имеют непрямоугольную форму, не забудьте сохранить прозрачность при записи файлов .png.

1.5.11. Элемент customUpdateUI

Если в дескрипторе есть элемент `customUpdateUI`, то приложение получает возможность самостоятельно обрабатывать действия по своему обновлению. (Подробнее эта процедура описана в главе 8.) Если вы хо-

тите, чтобы приложение само управляло процессом обновления, присвойте этому элементу значение `true`. Если его значение `false` или опущено, для обновления используются стандартные диалоги AIR.

1.5.12. Элемент fileTypes

Необязательный элемент `fileTypes` позволяет связать с приложением некоторые типы файлов. Если некий тип файла связан с приложением, то при двойном щелчке по значку файла этого типа автоматически запускается приложение AIR, если оно не было запущено ранее. После этого генерируется событие, которое перехватывается работающим приложением AIR и сообщает ему сведения о файле, по которому был сделан двойной щелчок, после чего приложение AIR решает, как ему действовать дальше. Подробнее об обработке этого события можно прочесть в главе 3.

Если есть элемент `fileTypes`, у него должен быть один или несколько дочерних элементов `fileType`. Каждый элемент `fileType` должен содержать элементы `name` и `extension`. Кроме того, элемент `fileType` может содержать элементы `description` и `contentType`. Значением `contentType` может быть тип MIME. (Подробнее о типах MIME можно прочесть на en.wikipedia.org/wiki/MIME.) Ниже следует пример элемента `fileTypes`, в котором зарегистрирован единственный тип файла:

```
<fileTypes>
   <fileType>
      <name>com.manning.ExampleApplicationSavedSettings</name>
      <extension>exp</extension>
      <description>A saved settings file for Example Application
    ➥</description>
      <contentType>text/xml</contentType>
   </fileType>
</fileTypes>
```

Как видите, в этом примере для обеспечения уникальности имени использовано значение `name` в виде обратного доменного имени. Обратите также внимание на отсутствие точки перед расширением имени.

1.6. Создание приложений AIR с помощью Flex Builder

Flex Builder 3 предоставляет встроенные функции разработки приложений AIR и содержит все необходимые инструменты AIR. Если вы хотите строить AIR-приложения, основанные на Flex, то Flex Builder 3 предоставляет для этого отличные возможности.

В нескольких следующих разделах вы познакомитесь с основами применения Flex Builder 3 для создания AIR-приложений. В частности, вы узнаете, как сконфигурировать новый проект AIR, создать для

проекта MXML и другие файлы, выполнить тестирование/отладку проекта и создать инсталлятор приложения.

1.6.1. Конфигурирование нового проекта AIR

Если вы хотите начать разработку нового приложения AIR с помощью Flex Builder 3, то прежде всего необходимо создать проект AIR. Чтобы создать проект AIR, следует выбрать в меню Flex Builder 3 пункты File→New→Flex Project. В результате откроется диалоговое окно New Flex Project, приведенное на рис. 1.3.

Помощник – тот же самый, с помощью которого в Flex Builder 3 создается новый Flex-проект, поэтому для уточнения необходимых деталей вы можете обратиться к справочнику по Flex Builder. Единственное отличие – выбор «Desktop application» в качестве типа приложения, как показано на рис. 1.3.

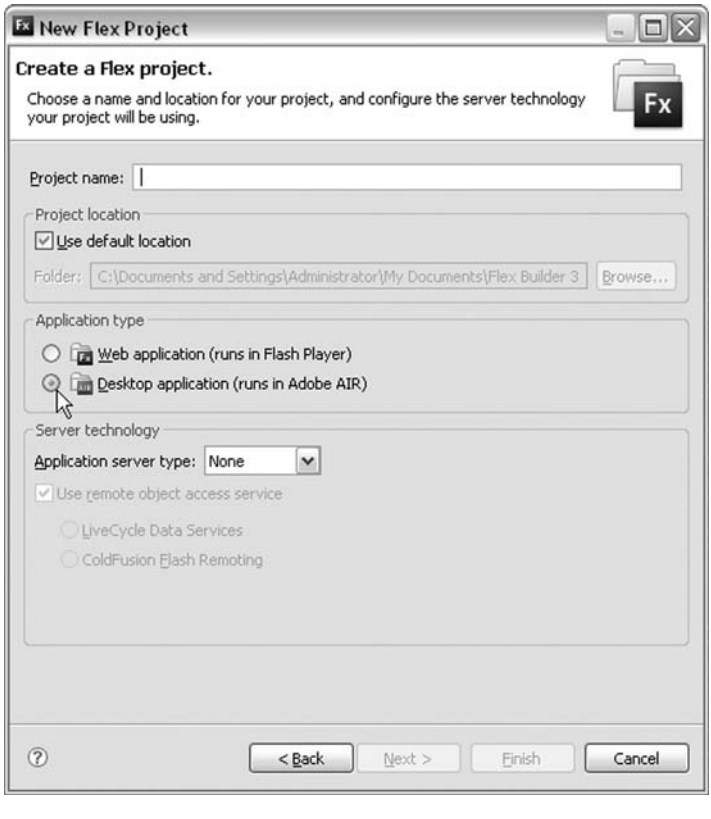


Рис. 1.3. *В диалоговом окне создания нового проекта Adobe AIR в Flex Builder 3 нужно прежде всего указать имя проекта и тип приложения. Для приложений AIR укажите в качестве типа «desktop application» (настольное приложение)*

Рис. 1.4. *В панели навигатора, показывающей файлы нового проекта AIR, приведены исходные файлы для приложения AIR и XML-файл дескриптора*

После создания проекта Flex Builder автоматически создает два файла в каталоге src: главный файл MXML и XML-файл дескриптора приложения по умолчанию. На рис. 1.4 приведена панель навигатора Flex Builder, в которой показаны эти файлы для нового проекта.

Конечно, вы можете отредактировать файл дескриптора, опираясь на полученные ранее в этой главе сведения. Кроме того, вы можете отредактировать главный файл MXML и ввести дополнительные файлы, описываемые ниже.

1.6.2. Создание файлов проекта AIR

После создания проекта AIR в Flex Builder вам, несомненно, потребуется отредактировать главный файл MXML и, возможно, потребуется добавить дополнительные файлы MXML, ActionScript, CSS и других типов. Чаще всего можно воспользоваться теми же типами файлов, кодом MXML и ActionScript, CSS и структурами, что и в стандартных приложениях Flex. Главное отличие между приложениями AIR и Flex состоит в том, что корневым тегом приложения AIR является WindowedApplication, а не Application. Ниже приводится код, который по умолчанию помещается в главный файл MXML, когда Flex Builder создает новый проект AIR:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
  ➥layout="absolute">

</mx:WindowedApplication>
```

Компонента `WindowedApplication` представляет собой подкласс компоненты `Application`, а это значит, что все функции стандартного приложения Flex включены в приложение AIR. Однако экземпляры `WindowedApplication` обладают дополнительными функциями, отражающими специфическое для AIR поведение. Например, у `WindowedApplication` есть свойство `title`, с помощью которого можно изменять текст, показываемый для данного приложения в заголовке окна и панели задач. Подробнее об использовании `WindowedApplication` рассказывается в главе 2.

1.6.3. Тестирование приложения AIR

При создании приложения AIR вы, несомненно, тут же захотите его протестировать. Иметь возможность наблюдать за прогрессом AIR-

Рис. 1.5. *Панель инструментов Flex Builder позволяет осуществлять запуск, отладку и профилирование приложения AIR*

приложения по мере его разработки так же важно, как и при разработке приложений других типов. Flex Builder 3 значительно облегчает тестирование приложений по ходу работы над ними. Для этого Flex Builder предоставляет функции запуска, отладки и профилирования. Их запуск происходит при выборе из меню Run или при щелчке по соответствующим кнопкам на панели инструментов Flex Builder, изображенным на рис. 1.5.

Запуск (run) позволяет проверить приложение AIR обычным образом – без показа какой-либо дополнительной информации, как в вариантах отладки (debug) или профилирования (profile). При выборе функции отладки Flex Builder запускает приложение AIR в отладочном режиме, при котором на консоль выводятся результаты команды `trace()` и предоставляется возможность прерывать выполнение кода с помощью точек останова. При запуске приложения в режиме профилирования можно в реальном времени узнать объемы использования памяти работающим приложением.

При тестировании приложения AIR в режимах запуска, отладки и профилирования не требуется, чтобы оно имело цифровую подпись. Последнее необходимо только при публикации файла .air.

1.6.4. Создание инсталлятора

Если вы готовы начать распространение своего приложения AIR, вам нужно создать файл .air, который будет служить программой инсталляции. В начале главы, как вы помните, говорилось о том, что файл .air содержит архив всех файлов, необходимых вашему приложению AIR, и дает пользователю возможность двойным щелчком по единственному файлу установить все приложение.

Создать в Flex Builder 3 файл .air легко. Нужно проделать следующее:

1. Выбрать в меню Flex Builder пункт File→Export→Release Build или пункт Project→Export Release Build. Появится диалоговое окно Export Release Build, приведенное на рис. 1.6. Нужно выполнить два шага: выбрать проект, приложение и .air-файл для экспорта, а затем создать цифровую подпись приложения.
2. Заполните форму, выбрав проект и приложение, которое вы хотите экспортировать, а затем указав файл .air для сохранения, как показано на рисунке. Завершив этот шаг, щелкните по кнопке Next.
3. Создайте цифровую подпись, как предлагается на рис. 1.7. Для этого вам понадобится сертификат. (О цифровых сертификатах рассказывалось выше в разделе, посвященном проверке подлинности

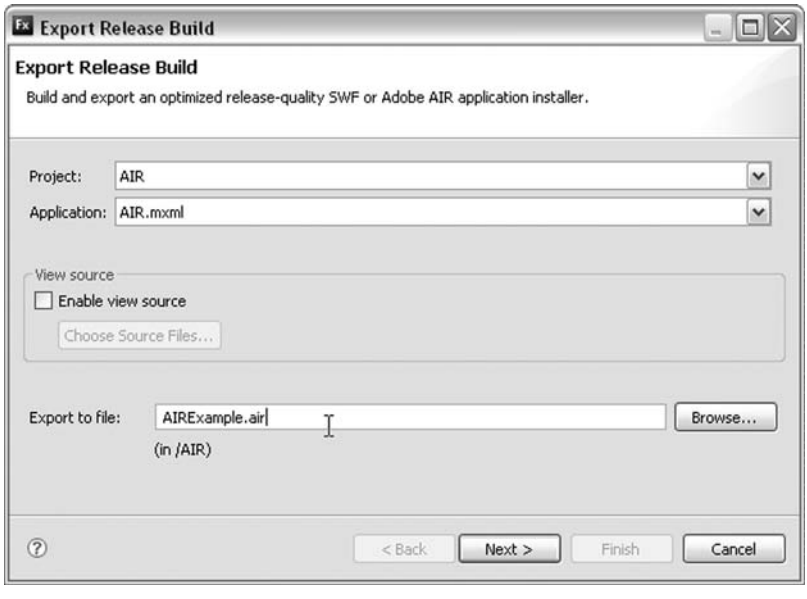


Рис. 1.6. *Первый экран диалогового окна Export Release Build предлагает выбрать тип экспорта*

приложений.) Если у вас есть сертификат, полученный от сертифицирующего органа, или у вас есть готовый сертификат, подписанный собственной подписью, можно открыть дерево каталогов и выбрать этот сертификат. В таком случае можно перейти к шагу 5. В противном случае создайте новый сертификат за собственной подписью, щелкнув по кнопке Create, и перейдите к шагу 4.

4. Щелкните по кнопке Create, чтобы показать диалоговое окно Create Self-Signed Digital Certificate (создать самостоятельно подписанный цифровой сертификат), приведенное на рис. 1.8. Заполните форму и щелкните по OK. Если вы ввели правильные данные, появится сообщение, извещающее об успешном создании сертификата, а затем вы вернетесь в окно Export Release Build.
5. Найдите в хранилище ключей тот сертификат, который хотели бы применить. (Если вы только что создали сертификат за собственной подписью, выберите тот файл .pfx, который создали на предыдущем этапе.) Выбрав хранилище ключей – существующее или новое, задайте пароль для сертификата в поле Password окна Export Release Build.
6. Щелкните по кнопке Finish, и Flex Builder создаст файл .air. Это все, что необходимо для создания с помощью Flex Builder 3 пригодной для распространения программы инсталляции приложения AIR.

Этим завершается наше обсуждение создания приложений AIR с помощью Flex Builder 3. Далее мы рассмотрим, как то же самое можно сделать с помощью Flash CS3.

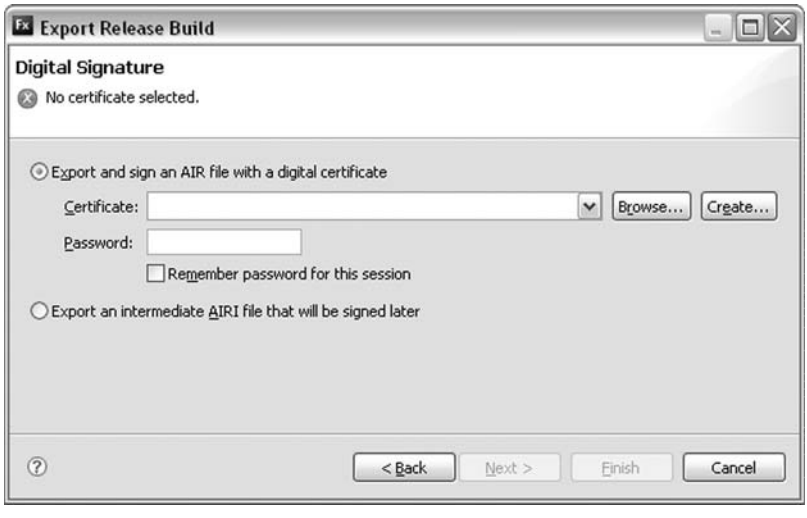


Рис. 1.7. *Второй экран диалогового окна Export Release Build просит вас создать электронную подпись приложения*

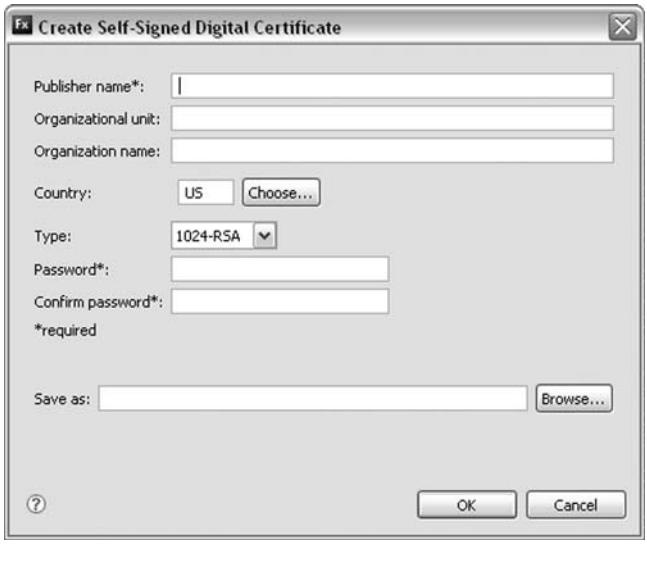


Рис. 1.8. *Создание сертификата за собственной подписью*

1.7. Создание приложений AIR с помощью Flash

Если вы хотите использовать Flash для создания приложений AIR, вам понадобится Flash CS3 (более ранние версии не могут создавать файлы AIR), а также обновление AIR, которое можно бесплатно загру-

зить с сайта Adobe *www.adobe.com/go/air*. На протяжении этого раздела (и всей оставшейся книги) мы будем предполагать, что это бесплатное обновление у вас уже установлено.

В нескольких следующих разделах мы подробно рассмотрим, как начать во Flash новый проект AIR, протестировать его, а затем создать программу установки.

1.7.1. Конфигурирование нового проекта AIR

Чтобы создать новый проект AIR с помощью Flash, нужно воспользоваться начальным экраном Flash. Если вы отключили его отображение, выберите Edit→Preferences в меню Flash CS3 и в категории General задайте Welcome Screen в пункте меню On launch.

В начальном экране, в колонке Create New, вы увидите вариант Flash File (Adobe AIR), как показано на рис. 1.9. (Если там нет такого варианта, то, скорее всего, вы не установили обновление AIR, о котором говорилось в предыдущем разделе, или это обновление встало некорректно.) Щелчок по Flash File (Adobe AIR) в колонке Create New запустит новый проект AIR. Проще некуда.[1]

Чтобы протестировать приложение, Flash и AIR должны найти файл дескриптора, поэтому необходимо сохранить файл .fla, прежде чем пытаться его запустить. Только после сохранения файла .fla Flash автоматически создаст файл дескриптора. XML-файл дескриптора помещается в тот же каталог. В разделе 1.7.4 вы узнаете, как модифицировать этот файл дескриптора. Впрочем, хранящихся в нем значений достаточно, чтобы протестировать ваше приложение.

Рис. 1.9. *Чтобы начать новый проект AIR, выберите пункт Flash File (Adobe AIR) в начальном экране*

[1] Того же эффекта можно добиться, создав обычное приложение ActionScript 3.0 и установив значение Adobe AIR 1,0 в выпадающем списке File→Publish Settings→Flash→Version. – *Прим. науч. ред.*

1.7.2. Создание файлов проектов AIR

Для AIR-проектов на базе Flash требуются те же типы файлов, что и для обычных Flash-проектов, предназначенных для работы в сети. Не требуется специально отслеживать типы создаваемых файлов и место их размещения. Вы так же можете пополнять библиотеку, работать с временной диаграммой, применять графические инструменты и писать код ActionScript, как в любом другом проекте Flash. Нужно только отметить, что AIR требует писать сценарии на ActionScript версии 3.

Примечание

Приложения AIR могут воспроизводить Flash-контент, в котором используется ActionScript 1.0 и ActionScript 2.0, если загрузить этот контент в приложение AIR на этапе исполнения. Однако если вы хотите создать новое приложение AIR, то должны применять ActionScript 3 в главном файле .swf этого приложения.

1.7.3. Тестирование приложения AIR

Для приложения AIR тестирование осуществляется точно так же, как для обычного приложения Flash. Вы можете просто выбрать Control→ Test Movie или нажать ту комбинацию клавиш, которой пользуетесь обычно. Единственная разница в том, что приложение AIR будет выполняться в окне AIR, а не в стандартном плеере Flash.

Процесс отладки приложения AIR аналогичен отладке стандартного приложения Flash. Можно просто выбрать Debug→Debug Movie или воспользоваться комбинацией клавиш, после чего отлаживать приложение AIR таким же образом, как и обычное приложение Flash.

1.7.4. Создание инсталлятора

Во Flash есть несколько команд, которые позволяют создать для приложения файл .air, не вводя никаких команд в командной строке и ничего не редактируя вручную. В меню Commands есть два пункта для AIR: AIR – Application & Installer Settings (настройки приложения и инсталлятора) и AIR – Create AIR File (создать файл AIR).

Команда AIR – Application & Installer Settings позволяет модифицировать файл дескриптора через форму, не редактируя XML вручную. Рисунок 1.10 показывает, как выглядит по умолчанию форма для проекта под названием Example.fla. Как видите, имя файла, имя и ID основаны на имени файла .fla. Кроме того, в ID по умолчанию имени файла, основанному на имени файла .fla, предшествует `com.adobe.example`. Версия по умолчанию имеет значение 1.0. Все показанные в форме элементы могут быть модифицированы, и эти изменения будут отражены в файле дескриптора.

Форма требует задания цифровой подписи. Как вы прочли в разделе 1.3.2, все приложения AIR должны иметь цифровую подпись. При

создании приложения AIR с помощью Flash вы указываете сертификат с помощью соответствующей опции в заполняемой форме. Чтобы применить сертификат, выполните следующие шаги:

1. Щелкните по кнопке Change и откройте окно Digital Signature, показанное на рис. 1.11.
2. Выберите вариант Sign the AIR file with a digital certificate, а затем сертификат, который должен быть использован. Если вы используете сертификат, выданный сертифицирующим органом или ранее созданный сертификат за собственной подписью, вы можете тут же осуществить поиск файла сертификата в файловой системе. После

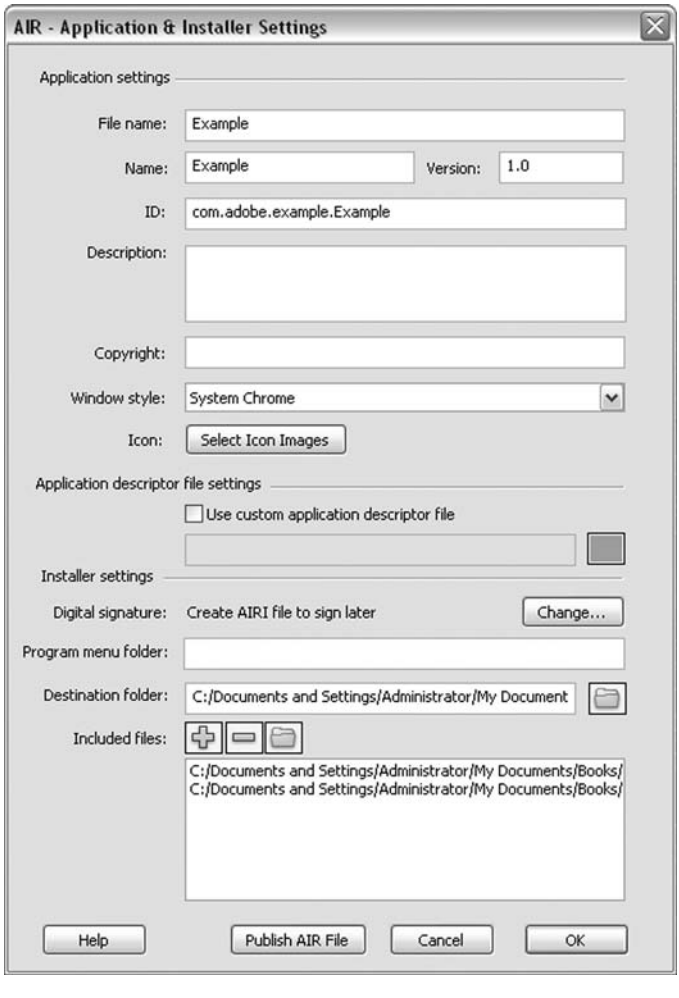


Рис. 1.10. *Редактирование настроек приложения и инсталлятора для приложения AIR в Flash*

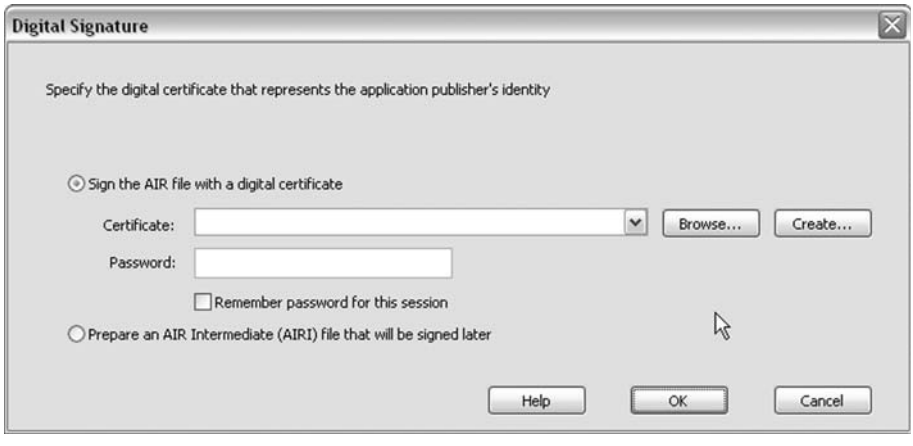


Рис. 1.11. *Диалоговое окно Digital Signature для задания сертификата для приложения AIR*

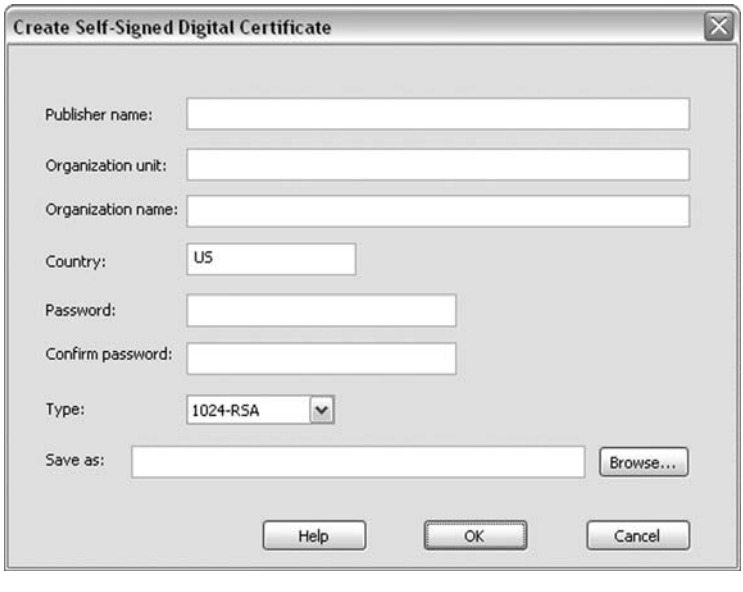


Рис. 1.12. *Диалоговое окно Create Self-Signed Digital Certificate для создания нового сертификата для приложения AIR*

этого можно сразу перейти к пункту 6. В противном случае, если вам нужно создать новый сертификат за собственной подписью, переходите к шагу 3.

3. Щелкните по кнопке Create и откройте диалоговое окно Create Self-Signed Digital Certificate (создание цифрового сертификата за собственной подписью), показанное на рис. 1.12.

4. Заполните все поля в диалоговом окне Digital Signature, включая пароль, который нужно запомнить, потому что он потребуется для применения сертификата к приложению AIR.
5. Сохраните сертификат на своей машине, щелкнув по кнопке OK. После этого вы вернетесь в предшествующее окно.
6. Найдите в файловой системе и выберите хранилище ключей для сертификата, который вы хотите применить.
7. Выбрав нужный сертификат, задайте для него пароль. Щелкните по кнопке OK; вы вернетесь в диалоговое окно AIR – Application & Installer Settings.
8. Щелчок по кнопке Publish AIR File вызовет создание файла .air. Если вы пока не готовы опубликовать файл .air, просто щелкните по кнопке OK.

Вы можете в любой момент опубликовать файл .air, вернувшись в окно AIR – Application & Installer Settings. Либо, если вы уже применили сертификат, можете просто выбрать во Flash пункт меню Commands→AIR–Create AIR File.

Таким образом, мы описали схему создания приложений AIR в Flash CS3. Далее посмотрим, как то же самое делается с помощью Flex SDK.

1.8. Создание приложений AIR с помощью Flex SDK

Если вы строите приложения Flex с помощью Flex SDK, то можете воспользоваться Flex 3 SDK для создания приложений AIR. Flex 3 SDK распространяется бесплатно и не накладывает ограничений на то, какие функции можно включать в приложения AIR. С помощью Flex 3 SDK можно делать точно такие же приложения AIR, как с помощью Flex Builder 3. Разница лишь в том, что Flex SDK не позволяет автоматизировать задачи (такие, как создание главных файлов MXML приложения и XML-файлов дескрипторов по умолчанию); кроме того, графические интерфейсы пользователя у Flex SDK и у Flex Builder различны. Следующие несколько разделов описывают, как создавать приложения AIR с помощью Flex SDK, принимая во внимание вышесказанное.

1.8.1. Конфигурирование нового проекта AIR

Так как Flex SDK ничего не автоматизирует, разработчику приходится самому создавать структуру каталогов и файлы проекта AIR. Обычно требуется создать каталог для нового проекта AIR. Затем нужно создать в нем два подкаталога: один для исходных файлов и другой – для выходных (файлов .swf и файла .air).

1.8.2. Создание файлов проекта AIR

Опять-таки, поскольку Flex SDK не создает файлы проекта автоматически, эта обязанность ложится на вас. Вам потребуется создать хотя

бы один файл MXML с корневым тегом WindowedApplication и один XML-файл дескриптора приложения. Подробнее о WindowedApplication см. раздел 1.6. Дополнительные сведения о WindowedApplication приводятся также в главе 2.

1.8.3. Тестирование приложения AIR

Тестирование приложений AIR с помощью Flex SDK состоит из двух этапов. Сначала нужно скомпилировать приложение. Затем вы можете запустить приложение через отладчик AIR. Компилировать приложение AIR можно с помощью компилятора mxmlc, почти так же, как стандартное приложение Flex. Разница в том, что при вызове mxmlc нужно добавить опцию компиляции, которая активизирует имеющийся в SDK специальный файл конфигурации для AIR. Эта опция имеет вид +configname=air. Вот пример команды для компиляции Main.mxml в качестве приложения AIR:

```
mxmlc +configname=air Main.mxml
```

Для облегчения работы во Flex SDK включена команда amxmlc, которая просто вызывает mxmlc с опцией +configname=air. Следовательно, можно не задавать эту опцию, если вызвать amxmlc вместо mxmlc:

```
amxmlc Main.mxml
```

В результате компиляции вы получаете файл .swf. Затем нужно запустить этот файл .swf в режиме отладки с помощью «AIR debug launcher» – исполняемого модуля с именем adl. Программа adl находится в том же каталоге, что и компиляторы, поэтому если компиляторы размещены в некотором каталоге вашей машины, то там же находится и adl. Когда нужно запустить приложение, вы просто задаете в качестве аргумента adl файл дескриптора приложения, которое нужно протестировать:

```
adl Main-descriptor.xml
```

Приведенная команда запустит приложение, описываемое в Main.descriptor.xml.

1.8.4. Создание инсталлятора

Чтобы создать инсталлятор AIR (файл .air) с помощью Flex SDK, нужно воспользоваться инструментом упаковки AIR – программой под названием adt. Она находится в том же каталоге, что и компиляторы Flex и средство запуска отладки AIR.

Создание сертификата

Прежде всего нужно убедиться в наличии сертификата для вашего приложения AIR. Если у вас есть сертификат, выданный сертифицирующим органом, или вы уже создали сертификат за собственной подписью, этот этап для вас пройден. Если же требуется создать сертифи-

кат с собственной подписью, нужно сделать это с помощью adt. Следующий синтаксис позволяет создать самоподписанный сертификат:

```
adt -certificate -cn name key_type pfx_file password
```

Замените здесь name, key_type, pfx_file и password нужными вам значениями. name – это обычное имя сертификата, которое может быть произвольной строкой. Значением key_type может быть 1024-RSA или 2048-RSA. Значение pfx_file указывает путь, по которому вы хотите сохранить сертификат (включая расширение имени файла .pfx). Значение password нужно запомнить, поскольку оно потребуется, когда вы будете заверять этим сертификатом свое приложение. Вот пример создания нового сертификата:

```
adt -certificate -cn ExampleCertificate 1024-RSA certificate.pfx 2u4fs8
```

Упаковка приложения AIR

При наличии сертификата можно создать инсталлятор AIR, воспользовавшись следующим синтаксисом:

```
adt -package SIGNING_OPTIONS air_file descriptor FILES_TO_INCLUDE
```

В этом синтаксисе air_file и descriptor нужно заменить фактическими значениями, которые вы хотите использовать. Значение air_file должно содержать путь, по которому вы хотите сохранить файл .air. Значение descriptor задает полный путь к XML-файлу дескриптора приложения. SIGNING_OPTIONS и FILES_TO_INCLUDE – сложные группы параметров, которые мы подробно обсудим ниже. Группа аргументов SIGNING_OPTIONS существенно различается в зависимости от способа цифровой подписи приложения. Есть целый ряд аргументов подписи, детально описанных в документации Adobe AIR. Не станем ее здесь воспроизводить, а лучше обсудим наиболее распространенные сценарии подписи и их реализацию с помощью adt.

- Подпись сертификатом PKCS#12 (включая использование файла .pfx, который вы, возможно, сгенерировали при создании своего сертификата). Если у вас есть сертификат PKCS#12, вам требуется задать только два аргумента: storetype и keystore. Аргумент storetype должен иметь значение pkcs12, а аргумент keystore должен содержать путь к файлу хранилища ключей, в котором находится сертификат (например, файл .pfx). Ниже приводится пример вызова adt с использованием самостоятельно подписанного сертификата, хранящегося в файле .pfx:

```
adt -package -storetype pkcs12 -keystore selfsigned.pfx
➥installer.air descriptor.xml Main.swf
```

- Подпись с помощью хранилища ключей Java. Хранилище ключей Java имеет тип JKS. Проще всего работать с хранилищем ключей Java, если оно содержит всего один сертификат и вы хотите воспользоваться хранилищем ключей Java, установленным по умолчанию.

Тогда вам нужно только указать аргумент `storetype` и задать для него значение `jks`, как в следующем примере:

```
adt -package -storetype jks installer.air descriptor.xml Main.swf
```

Если в хранилище несколько сертификатов, можно задать псевдоним нужного сертификата с помощью аргумента `alias`, как в следующем примере:

```
adt -package -alias sampleCertificateAlias -storetype jks installer.air
➥descriptor.xml Main.swf
```

А если вы хотите использовать нестандартное хранилище ключей, можете сделать это, задав значение аргумента `keystore`, как в следующем примере:

```
adt -package -storetype jks -keystore codeSigningCertificates.keystore
➥installer.air descriptor.xml Main.swf
```

`adt` позволяет использовать хранилища ключей других типов, включая PKCS#11 (аппаратно реализованное хранилище), KeychainStore (хранилище OS X), Windows-MY и Windows-ROOT.

Выданные сертифицирующими органами сертификаты имеют ограниченный срок действия (обычно 1 год). При создании приложения AIR с использованием сертификата срок действия инсталлятора обычно истекает одновременно со сроком действия сертификата. (На установленные экземпляры приложения это не оказывает влияния, но сам инсталлятор после истечения срока годности сертификата работать не будет). Однако если вы можете проверить действительность сертификата в момент подписи приложения (при создании файла .air), вы снимете ограничение срока годности с инсталлятора приложения AIR. Для проверки действительности сертификата нужно адресовать `adt` на сервер временных меток (он должен соответствовать RFC3161), для чего служит аргумент `tsa`. Ниже следует пример проставления временной отметки приложения с помощью сервера *http://ns.szikszi.hu:8080/tsa*:

```
adt -package -storetype pkcs12 -keystore certificates.pfx -tsa
➥http://ns.szikszi.hu:8080/tsa installer.air descriptor.xml Main.swf
```

Группа `FILES_TO_INCLUDE` позволяет задать все файлы, которые нужно включить в файл .air. Не забывайте, что файл .air является архивом. Все файлы, указанные в группе `FILES_TO_INCLUDE`, будут упакованы в архив, а при установке приложения будут извлечены из архива. В инсталлятор всегда должен быть включен файл контента начального окна (.swf). Ниже приводится пример создания файла .air, содержащего только контент начального окна (полагаем, что это Main.swf):

```
adt -package -storetype pkcs12 -keystore selfsigned.pfx installer.air
➥descriptor.xml Main.swf
```

Однако вам может понадобиться включить в инсталлятор дополнительные файлы – графику, звук, видео, текст и т. п. Достаточно просто перечислить их наряду с контентом начального окна:

```
adt -package -storetype pkcs12 -keystore selfsigned.pfx installer.air
➥descriptor.xml Main.swf image.jpg video.flv data.txt
```

Можно также задавать целые каталоги. В следующем примере включены все файлы из текущего каталога:

```
adt -package -storetype pkcs12 -keystore selfsigned.pfx installer.air
➥descriptor.xml
```

Можно также воспользоваться флажком -C, чтобы изменить каталог, и тогда все, что за ним последует, будет считаться относящимся к этому каталогу. Например, следующая команда включает файл Main.swf из текущего каталога и несколько графических файлов из подкаталога assets/images:

```
adt -package -storetype pkcs12 -keystore selfsigned.pfx installer.air
➥descriptor.xml Main.swf assets/images/image1.jpg
➥assets/images/image2.jpg assets/images/image3.jpg
➥assets/images/image4.jpg
```

При упаковке приложения таким способом структура каталогов будет воссоздана в том виде, как она указана в списке файлов. Это означает, что после установки приложения графические файлы окажутся в подкаталоге assets/images каталога, содержащего Main.swf. Если ваше приложение готово к такой организации файлов, проблем нет. Однако если вы хотите добавить файлы из разных мест, но организовать их по-другому в файле .air (и, следовательно, после установки приложения), нужно использовать флаг -C. Рассмотрим такую команду:

```
adt -package -storetype pkcs12 -keystore selfsigned.pfx installer.air
➥descriptor.xml Main.swf -C assets/images image1.jpg image2.jpg
➥image3.jpg image4.jpg
```

В этом примере графика помещается в архив в тот же самый каталог, что и Main.swf. Это значит, что если Main.swf ищет графические файлы в собственном каталоге, то файлы должны быть добавлены в архив таким способом.

Примечание

Есть и другие параметры, передаваемые `adt` при создании инсталлятора. Мы отметили лишь необходимые шаги для создания файла .air. Дополнительные сведения можно найти в официальной документации.

После запуска `adt` вы получите файл .air, который можно распространять.

На этом мы завершаем обсуждение создания приложений AIR с помощью Flex SDK. Теперь мы рассмотрим несколько учебных приложений. Мы попрактикуемся на них в применении полученных знаний для создания простых действующих приложений AIR.

1.9. Простое приложение AIR для Flex

В этом разделе мы построим простое приложение с помощью Flex Builder. Особой практической ценности оно иметь не будет. Наша задача сейчас не в том, чтобы создать нечто полезное, а в том, чтобы пройти все этапы построения приложения AIR с помощью Flex. Данное приложение всего лишь использует компоненту дерева, чтобы показать содержимое файловой системы пользователя, и дает возможность выбрать и показать графические файлы.

Прежде всего, создадим новый проект Flex, выбрав в качестве типа приложения Desktop application (настольное). Назовем этот проект QuickStartAIRApplication. При создании проекта Flex Builder автоматически создает файлы QuickStartAIRApplication.mxml и дескриптора.

Поместите код, приведенный в листинге 1.1, в файл QuickStartAIRApplication.mxml.

Листинг 1.1. Документ MXML для учебной программы Flex

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml">

   <mx:Script>
      <![CDATA[

         private function changeHandler(event:Event):void {
            var ext:String = event.target.selectedItem.extension;

            if(ext == "jpg" || ext == "png") {
               image.source = event.target.selectedItem.nativePath;
            }
         }

      ]]>
   </mx:Script>
   <mx:HBox width="100%" height="100%">
      <mx:FileSystemTree width="50%" height="100%"
         change="changeHandler(event);"/>
      <mx:Image id="image" width="50%" height="100%"
         scaleContent="true" />
   </mx:HBox>
</mx:WindowedApplication>
```

Корневым элементом является WindowedApplication ❶

Получить расширение файла ❷

jpg или png

❸ Показать содержимое файловой системы

Если вам приходилось ранее работать с Flex, то структура кода в целом должна быть понятна. Однако некоторые элементы могут оказаться незнакомыми, поскольку они специфичны для AIR. Не пугайтесь. Многие из них будут более детально обсуждаться далее в книге. Сейчас мы попытаемся лишь кратко объяснить назначение этих незнакомых фрагментов. Прежде всего, обратите внимание, что корневым тегом является `WindowedApplication` ❶, а не `Application`. Это необходимое требование для приложений AIR на основе Flex. Обратите внимание на использование специфической для AIR компоненты под названием

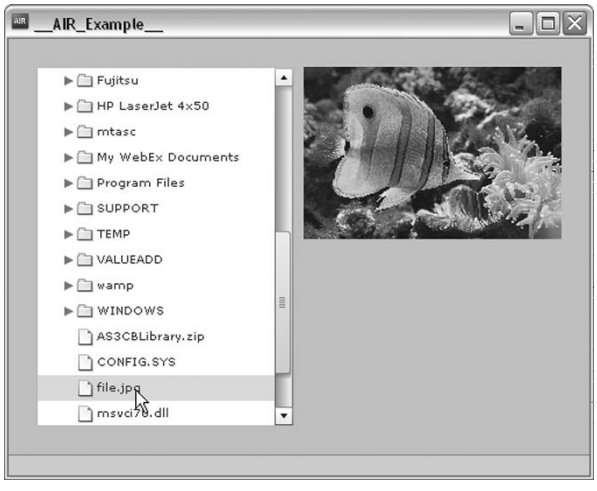


Рис. 1.13. *Это простое приложение позволяет перемещаться по локальной файловой системе и находить графические файлы*

`FileSystemTree` ❸, которая показывает содержимое локальной файловой системы. Когда пользователь щелкает по элементу этой компоненты, ее свойством `selectedItem` становится объект `File` со свойством `extension` ❷, которое возвратит расширение имени выбранного файла.

Вот и все наше простое приложение для Flex. Возьмите и запустите его. Результат показан на рис. 1.13.

Как видите, создавать в Flex Builder приложения AIR легко. Хотя в этом примере могли оказаться некоторые новые для вас вещи, но в основном он должен быть вам понятен. Вам уже известны MXML и большинство компонент, использованных в примере. Новые компоненты следуют тем же правилам, что и уже известные вам.

Теперь мы построим другое учебное приложение, воспользовавшись Flash CS3.

1.10. Простое приложение AIR для Flash

В этом разделе мы построим простое приложение с помощью Flash. Это приложение намеренно простое и довольно бесполезное. Однако наша цель не в том, чтобы сделать действительно полезное приложение AIR, а в том, чтобы, пройдя все необходимые этапы, создать с помощью Flash реально работающее приложение AIR. В приложении использована компонента List, чтобы отобразить систему каталогов файловой системы, начиная с рабочего стола машины пользователя. С помощью двойных щелчков пользователь может перемещаться по файловой системе.

Прежде всего, нужно создать во Flash новый проект AIR, для чего выбрать Flash File (Adobe AIR) в колонке Create New начального экрана

Flash. В результате откроется новый файл Flash, настроенный для AIR. Сразу сохраните этот файл под именем QuickStartAIRApplication.fla.

После сохранения файла создайте в том же каталоге, куда вы его поместили, цепочку каталогов com/manning/books/airinaction. Потом создайте из Flash новый файл ActionScript и сохраните его в com/manning/books/airinaction под именем QuickStartAIRApplication.as.

В файле Flash откройте панель компонент и панель библиотеки. Перетащите компоненту List из панели компонент на панель библиотеки. Тем самым вы добавите в проект компоненту List, и теперь можно будет создавать ее экземпляры с помощью кода ActionScript. Снова сохраните файл Flash.

Вернитесь в файл ActionScript и добавьте в него код из листинга 1.2. Этот файл класса будет использован в качестве класса документа для приложения Flash.

Листинг 1.2. Код класса документа для простой программы Flash

```
package com.manning.books.airinaction {
   import  flash.display.MovieClip;
   import  flash.events.MouseEvent;
   import  fl.controls.List;
   import  fl.data.DataProvider;
   import  flash.filesystem.File;

   public class QuickStartAIRApplication extends MovieClip {

      private var _list:List;

      public function QuickStartAIRApplication() {

         _list = new List();
        addChild(_list);

         _list.addEventListener(MouseEvent.DOUBLE_CLICK, clickHandler);

         _list.width = 550;
         _list.height = 400;

         _list.labelField = "name";

         _list.dataProvider = new
➥DataProvider(File.desktopDirectory.getDirectoryListing());
      }

      private function clickHandler(event:MouseEvent):void {
         if(_list.selectedItem.isDirectory) {
             _list.dataProvider = new
➥DataProvider(_list.selectedItem.getDirectoryListing());
        }
      }

   }

}
```

1 Импортировать класс File

2 Показать имя файла/каталога

3 Показать рабочий стол пользователя

4 Проверить, является ли выбранный элемент каталогом

5 Обновить список

Часть этого кода может оказаться для вас непривычной, поскольку она специфична для AIR. Весь этот новый код мы будем подробно обсуждать на протяжении всей книги. В данное время достаточным будет кратко отметить, где встречается новый код и какие задачи он решает. Во-первых, как можно заметить, мы импортируем класс `File` ❶. Этот класс служит для представления файлов и каталогов локальной файловой системы, о чем будет рассказано в главе 2. Затем мы настраиваем список так, чтобы он показывал значение свойства `name` для каждого элемента источника данных ❷. В качестве источника данных мы используем коллекцию объектов `File`, обладающих свойством `name`. Свойство `File.desktopDirectory` указывает на каталог рабочего стола пользователя, а обращение к `getDirectoryListing()` возвращает список содержимого этого каталога. Мы помещаем его в создаваемый экземпляр `DataProvider`, который присваиваем списку ❸. Свойство `isDirectory` сообщит нам, представляет ли объект `File` каталог. Мы будем менять список каталогов, только если пользователь сделает двойной щелчок по каталогу. Для этого мы проверяем значение `isDirectory` ❹. И только в этом случае мы обновляем источник данных списка, на этот раз путем вызова `getDirectoryListing()` ❺ для выбранного элемента.

Сохраните файл ActionScript и вернитесь к файлу Flash. В файле Flash задайте класс документа в виде `com.manning.books.airinaction.QuickStartAIRApplication`. Сохраните файл Flash и протестируйте приложение. Вы должны увидеть список файлов и каталогов, расположенных на рабочем столе. Если сделать двойной щелчок по элементу списка, список обновится и покажет содержимое нового каталога (если вы действительно щелкнули по каталогу). Рисунок 1.14 иллюстрирует, как выглядит приложение.

Вы только что написали свое первое, хоть и простое, приложение AIR с помощью Flash. Это лишь начало. Теперь, познакомившись с тем, как просто создавать приложения AIR, вы станете с помощью этой книги строить более сложные и полезные приложения.

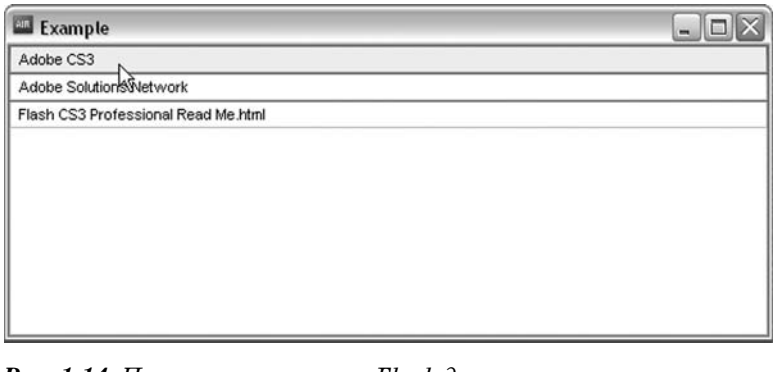


Рис. 1.14. *Простое приложение Flash дает пользователю возможность перемещаться по локальной файловой системе*

1.11. Резюме

В данной главе мы исходили из предположения, что читателю ничего не известно относительно AIR, и к концу построили первые приложения AIR с помощью Flex и/или Flash. Вы узнали, что такое AIR, какие наборы инструментов нужно использовать и с чего нужно начать. Конкретно, вы теперь знаете, что основанные на Flash или Flex приложения можно строить с помощью Flex Builder 3, Flex 3 SDK или Flash CS3 с расширением AIR. Поскольку приложения AIR выполняются внутри особой исполнительной среды, а не во Flash Player, им доступны более широкие функции, чем приложениям для работы в сети. Приложения AIR имеют доступ к локальной файловой системе, могут создавать локальные базы данных и осуществлять обмен данными с ними, участвовать в системных функциях перетаскивания (drag-and-drop) и выполнять другие функции, не доступные веб-приложениям для Flash или Flex.

В следующей главе вы научитесь управлять окнами, меню и приложениями AIR. Вы узнаете, как создавать новые окна, и какие типы окон можно создавать. Вы также научитесь правильно работать с открытыми окнами. Затем вы узнаете о разных способах работы с меню в приложениях AIR – от меню приложений и окон до контекстных меню.

В этой главе:

- Создание новых окон
- Управление открытыми окнами
- Выполнение команд уровня приложения
- Добавление в приложения меню системного уровня

2

Приложения, окна и меню

В главе 1 вы узнали много полезной базовой информации, которой должно хватить для понимания того, что представляет собой AIR и какова общая процедура построения и развертывания AIR-приложений. Теперь мы готовы к рассмотрению всех деталей создания приложений AIR. Начнем с самого начала. В этой главе освещаются следующие темы:

- *Приложения.* Прежде всего мы рассмотрим, как работать с приложениями AIR и понимать их с точки зрения программирования. В первой части этой главы рассказывается обо всем, что нужно знать для программирования приложений AIR.
- *Окна.* У каждого приложения AIR, каким бы простым или сложным оно ни было, есть, по крайней мере, одно окно, а часто их несколько. Окна представляют собой базовый и в то же время весьма непростой элемент. AIR предоставляет широкие возможности управления окнами, включая их стиль, форму, поведение и многое другое. Все эти вопросы освещаются в данной главе.
- *Меню.* Приложения AIR позволяют создавать множество различных меню, включая системные меню, меню приложения, контекстные меню и меню иконок. Обо всех этих типах меню будет рассказано в данной главе.

Сведений этой главы будет почти достаточно, чтобы начать строить разнообразные приложения AIR. На практике мы начнем работу над приложением, которое будет работать с сервисом YouTube и даст возможность осуществлять поиск среди клипов YouTube и воспроизводить их на своей машине. Материала данной главы будет достаточно, чтобы все это осуществить.

Мы начнем с рассмотрения метафоры, которая используется в AIR для структурирования приложения в виде объекта приложения и объектов

окон. В следующем разделе вы узнаете, как работать с объектом приложения и объектами окон с помощью Flash и Flex.

2.1. Общие сведения о приложениях и окнах

Несмотря на достаточную очевидность, стоит явно отметить, что в приложениях AIR есть программные конструкции для всего, что приложение представляет наглядно или в виде поведения. Это касается не только уже знакомых вам по Flex и Flash элементов (клипов, кнопок, элементов управления пользовательского интерфейса), но и специфичных для AIR понятий, таких как приложение или окно.

Для каждого приложения AIR есть один объект приложения и один или несколько объектов окон. Отсюда следует, что в каждом приложении AIR, основанном на Flex или Flash, есть всего один объект ActionScript, представляющий это приложение, который предоставляет доступ к информации уровня приложения (данные дескриптора приложения, тайм-аут по бездействию пользователя) и поведению (регистрация типов файлов, завершение приложения). Далее, у каждого окна в приложении AIR, основанном на Flex или Flash, есть представляющий его объект ActionScript. Эти объекты предоставляют доступ к специфичной для окон информации (ширина, высота, расположение на экране) и поведению (минимизация, восстановление).

У каждого приложения AIR есть, по крайней мере, одно окно – то, которое задано как содержимое начального окна в файле дескриптора приложения. Это то окно, которое вы увидите, запустив приложение. Однако в каждом приложении можно открыть несколько окон. Каждое окно можно представить себе как новый поток исполнения в приложении, подобно новым экземплярам веб-броузера, либо считать каждое окно панелью вашего приложения. Оба представления совершенно допустимы в отношении окон приложения AIR. Все зависит от вашей задачи. В любом случае есть лишь один объект `application` в ActionScript для каждого приложения AIR, и этот объект следит за всеми окнами в вашем приложении (рис. 2.1). На протяжении этой главы мы будем изучать способы работы с этими объектами и их взаимосвязи между собой.

Методы работы с приложением и его окнами хотя и близки, но несколько различаются в зависимости от того, что используется для создания AIR-приложения – Flash или Flex. Однако основные принципы ActionScript, используемые при создании приложений AIR, основанных на Flash, являются базовыми для разработки как для Flash-приложений, так и Flex-приложений. Поэтому, если вы применяете Flex для создания приложений AIR, вам нужно изучить основные принципы применения ActionScript, а также специфичные для Flex понятия. В следующих разделах мы обсудим эти базовые понятия. Если при создании AIR-приложений вы пользуетесь только Flash, вам нужно прочесть лишь раздел 2.1.1. Если вы пользуетесь Flex, то в дополнение к этому разделу прочтите раздел 2.1.2.

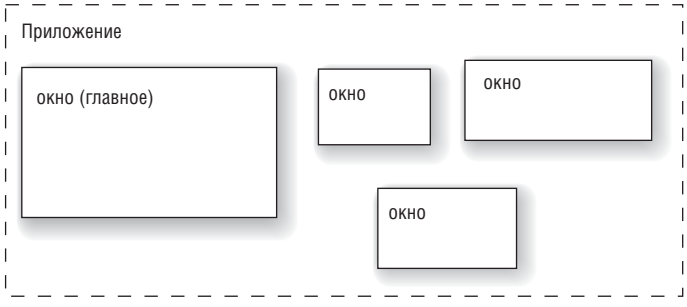


***Рис. 2.1.** В каждом приложении AIR есть один объект application и один или несколько объектов window*

2.1.1. Приложение Flash и окна

Работая с внутренним ActionScript API для приложений и окон AIR, вы должны разобраться в двух базовых классах: `flash.desktop.NativeApplication` и `flash.display.NativeWindow`. Если вы строите приложения AIR на основе Flash, то вам потребуется иметь дело только с этими двумя классами приложения и окна.

Создание приложения

Каждое приложение AIR автоматически получает один экземпляр `NativeApplication`. Нельзя создать несколько экземпляров `NativeApplication`. На самом деле, создать экземпляр `NativeApplication` вы вообще не сможете. Один экземпляр будет создан при запуске приложения, и он доступен в виде статического свойства класса `NativeApplication` в виде `NativeApplication.nativeApplication`. Многообразные применения этого экземпляра `NativeApplication` будут продемонстрированы далее в этой главе.

Создание окон

В основе каждого окна приложения AIR лежит объект NativeWindow. Если начальное окно автоматически создается при запуске приложения, то все дополнительные окна, которые могут понадобиться приложению, должны быть созданы программно разработчиком этого приложения. Получить новые окна можно путем создания новых объектов `NativeWindow` и последующего их открытия.

Чтобы создать объект `NativeWindow`, нужно сначала создать объект `flash.display.NativeWindowInitOptions`, который используется конструктором `NativeWindow` для определения ряда начальных параметров, таких как тип окна, параметры его оформления и т. д. Пожалуй, главными свойствами объекта `NativeWindowInitOptions` являются `type`, `systemChrome` и `transparent`. Эти свойства являются взаимозависимыми.

Свойство `type` может принимать одно из трех значений, которые определены тремя константами в классе `flash.display.NativeWindowType`:

STANDARD, UTILITY и LIGHTWEIGHT. По умолчанию устанавливается тип «стандартный», что означает использование всех системных параметров оформления окна и создание уникального системного окна. (Оно появляется в панели задач Windows или оконном меню OS X.) Стандартные окна более всего подходят для показа объектов, в принципе являющихся отдельными сущностями, такими как новая фотография для редактирования в графическом редакторе. Вспомогательные (utility) окна в меньшей степени наследуют системные параметры окон. В отличие от стандартных окон они не показываются в панели задач или оконном меню. Благодаря этому они лучше подходят для контента, связанного по смыслу с главным окном, например, для панелей инструментов. Облегченные (lightweight) окна никак не связаны с системными параметрами оформления. Так же, как вспомогательные окна, они не отображаются в панели задач или оконном меню. Поскольку облегченные окна не несут системных параметров, их свойство systemChrome должно быть установлено в none.

Свойство systemChrome определяет параметры оформления окна. Возможные значения определяются двумя константами класса flash.display.NativeWindowSystemChrome: STANDARD и NONE. Стандартное оформление использует параметры операционной системы. Это значит, что окна AIR будут выглядеть так же, как окна обычных приложений, выполняющихся на том же компьютере. Если установить свойство systemChrome в none, то окно будет лишено всех элементов оформления. (Начальное окно составляет исключение из этого правила, поскольку использует оформление AIR, если системное оформление задано в файле дескриптора как none). Это означает, что окна, созданные при значении systemChrome, установленном в none, не будут иметь встроенных механизмов развертывания, свертывания, восстановления, закрытия, изменения размеров или перемещения. Если в окне такого типа вам нужны подобные режимы, потрудитесь реализовать их программно. (Эти вопросы освещаются ниже в данной главе.)

Свойство transparent является булевым и указывает на то, может ли данное окно использовать альфа-слияние для реализации прозрачности, благодаря чему под данным окном могут быть видны другие окна. По умолчанию данное свойство имеет значение false. Задание значения true разрешает альфа-слияние. Учтите, что включение прозрачности повлечет повышенное использование системных ресурсов. Кроме того, если включена прозрачность, то systemChrome нужно установить в none. В отличие от обычных окон прозрачные окна могут иметь непрямоугольную форму и использовать эффект затухания.

Примечание

По умолчанию у всех окон есть цвет фона. Присваивание true прозрачности окна удаляет цвет фона, что дает возможность альфа-слияния. Кроме того, появляется возможность создания окон неправильной формы. Подробности – в разделе «Создание окон неправильной формы» ниже в этой главе.

Кроме того, с помощью свойств `minimizable`, `maximizable` и `resizable` объекта `NativeWindowInitOptions` можно указать, будет ли окно позволять свертывание, развертывание и изменение размеров. По умолчанию все эти свойства имеют значение `true`.

После того как создан объект `NativeWindowInitOptions`, можно создать объект `NativeWindow`, вызвав конструктор и передав ему объект `NativeWindowInitOptions`, как показано в листинге 2.1.

Листинг 2.1. Создание объекта NativeWindow

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindowType;

   public class Example extends MovieClip {

        public function Example() {
         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         var window:NativeWindow = new NativeWindow(options);

         window.width = 200;
         window.height = 200;

       }

   }

}
```

❶ Создать параметры окна

❷ Создать новое окно

❸ Установить начальную ширину и высоту

В этом примере мы сначала создаем параметры окна и устанавливаем для этого объекта параметров значения типа и оформления ❶. Затем мы создаем собственно окно, передавая ему параметры ❷. Мы также задаем начальные размеры 200 на 200 ❸. Подробнее о том, как действует настройка ширины и высоты, см. в разделе «Добавление в окна контента». Мы успешно создали окно, задали его параметры и даже установили размер. Однако в результате выполнения этого кода окно еще не отобразится на экране. Теперь посмотрим, как этого добиться.

Открытие окон

Если выполнить код, приведенный в листинге 2.1, никакого нового окна не появится. Причина кроется в том, что хотя вы и создали новое окно, вы еще не сообщили приложению, что его нужно открыть. Открыть окно можно с помощью метода `activate()`. Добавив одну строку кода (см. жирный текст в листинге 2.2), мы увидим новое вспомогательное окно размером 200 на 200.

Листинг 2.2. Открытие нового окна

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindowType;

   public class Example extends MovieClip {

      public function Example() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         var window:NativeWindow = new NativeWindow(options);
         window.width = 200;
         window.height = 200;

         window.activate();

      }

   }

}
```

Новое окно, созданное в этом примере, имеет размер 200 на 200 пикселов и белый фон. Но другого содержимого в нем пока нет. В большинстве окон есть то или иное содержимое, и вы должны его добавить, как будет показано в следующем разделе.

Добавление контента в окно

Когда создается новое окно, у него нет иного контента, кроме фона (и даже фон отсутствует, если вы создали прозрачное окно). Контент помещается в окно программным образом с помощью свойства `stage`.

Возможно, вас удивит, что `NativeWindow`, несмотря на присутствие в пакете `flash.display`, фактически не является отображаемым объектом. Он не наследует от `DisplayObject` – базового отображаемого типа в ActionScript. Вместо этого отображением объектов управляют окна. У объекта `NativeWindow` есть свойство `stage` типа `flash.display.Stage`. `stage` является ссылкой на отображаемый объект, используемый в качестве контейнера для содержимого окна. Поскольку объект `Stage` является контейнером отображаемых объектов, он позволяет добавлять, удалять и управлять контентом с помощью методов `addChild()`, `removeChild()` и подобных, как и всякий контейнер отображаемых объектов.

Листинг 2.3 берет за основу код листинга 2.2 и добавляет в окно текстовое поле. Добавленный код выделен жирным шрифтом.

Листинг 2.3. Добавление контента в окно

```
package {

   import flash.display.MovieClip;
```

```
import flash.display.NativeWindow;
import flash.display.NativeWindowInitOptions;
import flash.display.NativeWindowType;
import flash.text.TextField;
import flash.text.TextFieldAutoSize;

public class Example extends MovieClip {

   public function Example() {

   var options:NativeWindowInitOptions =
   ➥new NativeWindowInitOptions();
   options.type = NativeWindowType.UTILITY;

   var window:NativeWindow = new NativeWindow(options);
   window.width = 200;
   window.height = 200;

   var textField:TextField = new TextField();
   textField.autoSize = TextFieldAutoSize.LEFT;
   textField.text = "New Window Content";

   window.stage.addChild(textField);

   window.activate();

  }

 }

}
```

❶ Создать новый объект для показа

❷ Добавить контент в окно

В этом примере появилось два изменения. Во-первых, мы добавили новое текстовое поле с текстом New Window Content ❶. Мы взяли текстовое поле, но это мог быть любой другой отображаемый объект. Затем мы поместили текстовое поле в окно через свойство stage ❷. Для добавления текстового поля мы воспользовались методом `addChild()`, что является стандартным способом добавления содержимого в контейнер отображаемых объектов. Рисунок 2.2 иллюстрирует результат работы этого кода (в Windows).

Рис. 2.2. *В новом окне с текстом содержимое масштабируется, и текст может выглядеть не так, как ожидалось*

Вероятно, этот текст выглядит иначе, чем вы предполагали. Это вызвано тем, что (возможно, неожиданно) содержимое (сцена) нового окна по умолчанию подверглось масштабированию. Дело в том, что если задать ширину и высоту окна (как в нашем примере), то содержимое масштабируется относительно первоначальных размеров окна. В данном случае размеры окна были заметно увеличены, из-за чего текст оказался крупнее, чем можно было предположить.

С помощью свойства сцены `scaleMode` можно изменить такое поведение. В данном случае, лучше было бы не масштабировать содержимое. Можно присвоить `scaleMode` значение константы `StageScaleMode.NO_SCALE`, и масштабирования больше не будет. После запуска нового кода становится ясно, что нужно изменить также свойство выравнивания. В данном случае лучше всего, если сцена будет всегда выравниваться по левому верхнему углу. На листинге 2.4 эти дополнения выделены жирным шрифтом.

Листинг 2.4. Настройка масштаба и выравнивания содержимого нового окна

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindowType;
   import flash.text.TextField;
   import flash.text.TextFieldAutoSize;
   import flash.display.StageScaleMode;
   import flash.display.StageAlign;

   public class Example extends MovieClip {

      public function Example() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         var window:NativeWindow = new NativeWindow(options);
         window.width = 200;
         window.height = 200;

         var textField:TextField = new TextField();
         textField.autoSize = TextFieldAutoSize.LEFT;
         textField.text = "New Window Content";

         window.stage.scaleMode = StageScaleMode.NO_SCALE;
         window.stage.align = StageAlign.TOP_LEFT;

         window.stage.addChild(textField);

         window.activate();

      }

   }

}
```

На рис. 2.3 показано, как выглядит новое окно.

Рис. 2.3. *После задания свойств сцены scaleMode и align содержимое нового окна отображается правильно*

Примечание

Если вы протестируете какой-либо из приведенных примеров, то можете обнаружить, что вспомогательные окна не закрываются автоматически при закрытии главного окна приложения. Хотя вспомогательное окно недоступно из панели инструментов или главного меню, оно остается открытым, пока вы его не закроете. Открытое окно может помешать вам снова протестировать свое приложение. Следите за тем, чтобы при закрытии главного окна приложения закрыть и его вспомогательные окна. Подробнее о том, как управлять вспомогательными окнами, рассказывается в разделе 2.2.3.

Теперь, когда мы рассмотрели элементарные способы создания окон с помощью ActionScript, мы можем обратиться к созданию классов ActionScript для окон.

Создание окон на базе классов ACTIONSCRIPT

Пока мы рассмотрели только создание окон с использованием базовых приемов. По мере создания все более сложных окон обычно становится осмысленным заключение кода окон в классы ActionScript. Каждый класс окна должен наследовать от NativeWindow. Затем в класс можно включить весь код, служащий добавлению содержимого, масштабированию, выравниванию и т. д. В листинге 2.5 приведен пример простого класса окна.

Листинг 2.5. Создание базового класса окна

```
package {

   import flash.display.NativeWindow;
   import flash.display.NativeWindowType;
   import flash.display.NativeWindowInitOptions;

   public class ExampleWindow extends NativeWindow {
```

```
        public function ExampleWindow() {
            var options:NativeWindowInitOptions =
            ➥new NativeWindowInitOptions();

            options.type = NativeWindowType.UTILITY;

            super(options);

            width = 200;
            height = 200;

        }

    }

}
```

Как можно видеть, это окно само устанавливает для себя оформление и тип, а также ширину и высоту. Обратите внимание, что прежде всего оно создает объект `NativeWindowInitOptions` и передает его конструктору надкласса. Создать экземпляр такого окна можно точно так же, как любой другой экземпляр `NativeWindow`: вызвать конструктор, а затем открыть окно с помощью `activate()`. Листинг 2.6 демонстрирует, как будет выглядеть этот код.

Листинг 2.6. Создание и открытие экземпляра ExampleWindow

```
package {

    import flash.display.MovieClip;

    public class Example extends MovieClip {

        public function Example() {

            var window:ExampleWindow = new ExampleWindow();
            window.width = 200;
            window.height = 200;
            window.activate();

        }

    }

}
```

Прежде чем переходить к новым темам, нам нужно обсудить еще одно базовое понятие. До сих пор мы создавали только прямоугольные окна. Сейчас мы посмотрим, как создавать окна неправильной формы.

Создание окон неправильной формы

Возможность легко создавать окна неправильной формы – приятная особенность приложений AIR. Окно неправильной формы создается так же, как прямоугольное, но необходимо отключить системное оформление (хром) и сделать окно прозрачным. После этого можно задать для окна фон неправильной формы через свойство stage. В листинге 2.7 показан пример соответствующей модификации кода листинга 2.5.

Листинг 2.7. Создание окна непрямоугольной формы

```
package {

   import flash.display.NativeWindow;
   import flash.display.NativeWindowSystemChrome;
   import flash.display.NativeWindowType;
   import flash.display.NativeWindowInitOptions;
   import flash.display.Sprite;
   import flash.display.Stage;
   import flash.display.StageAlign;
   import flash.display.StageScaleMode;

   public class ExampleWindow extends NativeWindow {

      private var _background:Sprite;

      public function ExampleWindow() {
         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();

         options.systemChrome = NativeWindowSystemChrome.NONE;
         options.type = NativeWindowType.LIGHTWEIGHT;

         options.transparent = true;

         super(options);

         _background = new Sprite();
         drawBackground(200, 200);
         stage.addChild(_background);

         width = 200;
         height = 200;

         stage.align = StageAlign.TOP_LEFT;
         stage.scaleMode = StageScaleMode.NO_SCALE;
      }

      private function drawBackground(newWidth:Number, newHeight:Number):
      ➥void  {
         _background.graphics.clear();
         _background.graphics.lineStyle(0, 0, 0);
         _background.graphics.beginFill(0x0000FF, .5);
         _background.graphics.drawRoundRectComplex(0, 0, newWidth,
                              newHeight, 20, 20, 20, 1);
       _background.graphics.beginFill(0xFFFFFF, .9);
       _background.graphics.drawRoundRectComplex(5, 5, newWidth - 10,
                              newHeight - 10, 20, 20, 20, 1);
       _background.graphics.endFill();

     }

   }

}
```

❶ Задать systemChrome как none

❷ Сделать окно прозрачным

❸ Создать фон

В этом примере много кода, но его легко понять, если рассматривать поэтапно. Прежде всего мы должны присвоить свойству `systemChrome` объекта `options` значение `none` ❶. Это удалит из окна все элементы оформления, которые иначе потребовали бы прямоугольной рамки. Затем мы задаем для окна режим прозрачности ❷. Это важно, потому что у обычного окна есть сплошной прямоугольный фон. Чтобы задать непрямоугольную фигуру, нужно скрыть фон. После этого мы создадим объект фона, нарисуем на нем непрямоугольную фигуру и поместим ее на сцену ❸. В данном примере мы нарисуем прямоугольник с закругленными углами, представляющий некоторую разновидность обычного прямоугольного фона.

На рис. 2.4 показан результат работы такого кода.

При создании окон неправильной формы вам приходится создавать необходимые элементы интерфейса пользователя и код для функций, обычно автоматически обеспечиваемых системой: закрытие, минимизация, максимизация и перемещение. Подробнее о том, как это сделать, будет рассказано в разделе 2.2.

Рис. 2.4. Окно неправильной формы с прозрачным фоном, накрывающим значки рабочего стола и главное окно приложения

2.1.2. Приложение Flex и окна

При создании приложений AIR с помощью Flex рабочий процесс при создании и управлении окнами становится несколько иным. Тем не менее, базовые принципы остаются такими же, как для Flash-приложений AIR, использующих `NativeApplication` и `NativeWindow`. Разница в том, что Flex предоставляет две компоненты, облегчающие программное управление приложением и окнами. Компонента `WindowedApplication` обеспечивает работу с приложениями, а компонента `Window` – с окнами.

Создание приложения

Все приложения AIR на основе Flex должны компилироваться из документов MXML, использующих WindowedApplication в качестве корневого элемента. Поэтому при создании во Flex Builder в проекте AIR нового документа приложения MXML вы получаете следующий код-заглушку:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
       layout="absolute">

</mx:WindowedApplication>
```

Так как в каждом проекте Flex можно иметь только один документ приложения MXML, то и в каждом приложении может быть лишь один объект WindowedApplication. Компонента WindowedApplication расширяет компоненту Application, обычно используемую сетевыми Flex-приложениями. Поэтому свойства и методы Application доступны также в WindowedApplication. Однако статические свойства подклассами не наследуются. Поэтому свойство Application.application, ссылающееся на объект главного приложения, не наследуется. Если вам нужно обратиться к экземпляру WindowedApplication (вне документа MXML – внутри него это можно сделать посредством this), вы должны воспользоваться Application.application. В качестве типа объекта Application.application указывается класс Application, а не класс WindowedApplication. Поэтому вы должны привести тип объекта, если собираетесь обращаться к нему как к WindowedApplication:

```
var windowedApplication:WindowedApplication =
➥Application.application as WindowedApplication;
```

Экземпляры WindowedApplication обладают рядом свойств и методов, часть которых мы подробно рассмотрим в этой главе. Однако чаще всего вам потребуется главное свойство, через которое можно получить доступ к основным базовым значениям и поведениям, – свойство nativeApplication, являющееся ссылкой на соответствующий объект NativeApplication.

Создание окон

При создании приложений AIR на основе Flex все окна должны быть основаны на компоненте Window. Хотя можно создавать окна непосредственно с помощью NativeWindow (и NativeWindow все равно незримо используется), но специфическая для Flex компонента Window хорошо интегрируется со всей средой Flex и упрощает ряд аспектов создания окон, как будет вскоре показано в разделе «Добавление контента в окна».

Создавать окна с помощью Flex еще проще, чем с помощью Flash. При непосредственной работе с объектами NativeWindow, как вы уже знаете, нужно сначала создать объект NativeWindowInitOptions. Во Flex компонента Window избавляет вас от этого. Вам нужно лишь создать новый

объект Window (или объект на основе подкласса Window), а затем, если требуется, задать несколько свойств прямо для этого объекта. Например, код листинга 2.8 создает новое вспомогательное окно.

Листинг 2.8. Создание нового окна с помощью Flex

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import mx.core.Window;

         private function creationCompleteHandler():void {
            var window:Window = new Window();

            window.width = 200;

            window.height = 200;

            window.type = NativeWindowType.UTILITY;
         }

      ]]>
   </mx:Script>
</mx:WindowedApplication>
```

Создать новое окно

Задать начальные размеры

Сделать окно вспомогательным

Примечание

Подобно тому, как у объектов WindowedApplication есть свойство nativeApplication, указывающее на связанный с ними объект NativeApplication, так и у объектов Window есть свойство nativeWindow, указывающее на связанный с ними объект NativeWindow. Стоит заметить, что у объектов WindowedApplication тоже есть свойство nativeWindow, указывающее на связанный с ними объект NativeWindow для главного окна.

Если вы тестировали приведенный код, то могли заметить, что при его запуске не появляется никакого окна. Так же как при непосредственной работе с объектами NativeWindow после создания окна нужно явно открыть его.

Открытие окон

Как мы только что убедились, после того, как вы создали окно, еще требуется программным образом открыть его. Для объекта Window окно открывается обращением к методу open(). Листинг 2.9 содержит модифицированный код листинга 2.8, в который добавлен вызов open(), чтобы открыть новое окно.

Листинг 2.9. Чтобы открыть окно, достаточно вызвать метод open( )

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
```

```
        <![CDATA[
           import mx.core.Window;

           private function creationCompleteHandler():void {
              var window:Window = new Window();

              window.width = 200;
              window.height = 200;

              window.type = NativeWindowType.UTILITY;

              window.open();
           }

        ]]>
    </mx:Script>
</mx:WindowedApplication>
```

В этом примере новое окно имеет размеры 200 на 200 пикселов, серый цвет фона (по умолчанию во Flex) и никакого другого содержимого. Посмотрим теперь, как добавлять содержимое в окно с помощью Flex.

Добавление контента в окно

Добавление контента в окно во Flex обычно происходит иначе, чем при добавлении контента в окна объектов NativeWindow во Flash. В последнем случае вы должны программным способом добавить содержимое на сцену объекта NativeWindow после того, как создадите его. При работе с окнами Flex чаще просто создают новые компоненты MXML, основанные на Window, помещают в эти компоненты содержимое, а затем открывают экземпляры этих компонент как окна. Приведем пример.

В разделе, посвященном действиям с окнами на основе NativeWindow, мы видели пример создания нового окна и добавления в него содержимого. Теперь мы получим аналогичный результат, действуя методами Flex. Начнем с того, что создадим новую компоненту MXML и назовем ее SimpleTextWindow.mxml. Код этой компоненты показан на листинге 2.10.

Листинг 2.10. Создание простой оконной компоненты

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" width="200"
 height="200" type="utility">
   <mx:Label text="New Window Content" />
</mx:Window>
```

Обратите внимание на использование `Window` в качестве корневого элемента. Это существенно. Все компоненты, которые вы хотите использовать в качестве окон, должны расширять `Window`. Кроме того, мы задаем ширину и высоту компоненты, а также ее тип в самом документе MXML. Можно было бы задать эти свойства с помощью ActionScript, но в данном случае более оправданно задание их в самой компоненте окна, чтобы обеспечить идентичность всех экземпляров этого окна.

Теперь мы создадим экземпляр окна в документе MXML главного приложения, как показано на листинге 2.11.

Листинг 2.11. Создание экземпляра компоненты окна

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[

         private function creationCompleteHandler():void {
            var window:SimpleTextWindow = new SimpleTextWindow();
            window.open();
         }

      ]]>
   </mx:Script>
</mx:WindowedApplication>
```

Обратите внимание, что в этом коде мы создаем новый экземпляр компоненты `SimpleTextWindow` вместо общего объекта `Window`. Кроме того, поскольку мы задаем ширину, высоту и тип в самой компоненте MXML, не требуется устанавливать их при создании объекта. Рисунок 2.5 демонстрирует, как выглядит окно.

На этом наше обсуждение базовых прямоугольных окон в Flex завершается. Прежде чем перейти к совсем другой теме, мы обсудим, как с помощью Flex создавать окна неправильной формы.

Рис. 2.5. *Простое текстовое окно, созданное во Flex*

Создание окон неправильной формы

Мы уже научились создавать окна неправильной формы с помощью ActionScript, поэтому справедливо будет рассмотреть, как это делается с помощью Flex. Основная идея одинакова как во Flash, так и во Flex: создать окно с прозрачным фоном и без системного оформления. В обоих случаях для этого нужно выполнить одинаковые действия (установить `systemChrome` в `none` и `transparent` в `true`). Однако с окнами Flex

связана маленькая неприятность. В листинге 2.12 показан документ MXML компоненты окна. Код задает для свойства `systemChrome` значение `none` и для свойства `transparent` значение `true`. Затем с помощью `ActionScript` вычерчивается круг, который добавляется к списку отображаемых объектов.

Листинг 2.12. Создание во Flex прозрачного окна с systemChrome, установленным в none

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" systemChrome="none"
 type="lightweight" transparent="true" width="200" height="200"
 creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[

         private function creationCompleteHandler():void {

            var shape:Shape = new Shape();
            shape.graphics.lineStyle(0, 0, 0);
            shape.graphics.beginFill(0xFFFFFF, 1);
            shape.graphics.drawCircle(100, 100, 100);
            shape.graphics.endFill();

            rawChildren.addChild(shape);

         }

      ]]>
   </mx:Script>
</mx:Window>
```

Если создать экземпляр этого окна, то можно обнаружить, что к окну оформление все же было применено. Результат показан на рис. 2.6.

По умолчанию, если `systemChrome` установлено в `none`, Flex применяет свой стиль оформления окна. Если нужно удалить все элементы оформления, как в данном случае, нужно сделать еще один шаг и задать для свойства `showFlexChrome` нашего окна значение `false`. Все другие рассматривавшиеся нами свойства окон Flex могут быть заданы для конкретного экземпляра с помощью ActionScript или в MXML с помощью атрибутов. Свойство же `showFlexChrome` можно задать только с помощью MXML, поскольку оно должно быть определено до создания экземпляра

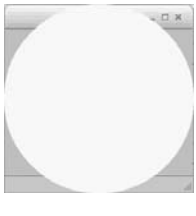


***Рис. 2.6.** Окно Flex с установленным параметром systemChrome*

окна. Код листинга 2.13 показывает, какие изменения нужно внести. После задания этого свойства оформление Flex будет также удалено, и окно примет форму круга.

Листинг 2.13. Задание для showFlexChrome значения false скрывает оформление окна, установленное во Flex

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" showFlexChrome="false"
 systemChrome="none" type="lightweight" transparent="true" width="200"
 height="200" creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[

         private function creationCompleteHandler():void {

            var shape:Shape = new Shape();
            shape.graphics.lineStyle(0, 0, 0);
            shape.graphics.beginFill(0xFFFFFF, 1);
            shape.graphics.drawCircle(100, 100, 100);
            shape.graphics.endFill();

            rawChildren.addChild(shape);

         }

      ]]>
   </mx:Script>
</mx:Window>
```

Теперь, когда вы научились созданию окон на низком уровне с помощью специальных приемов ActionScript и Flex, посмотрим, как управлять этими окнами.

2.2. Управление окнами

Создание и открытие окон – лишь первый шаг в эффективной работе с окнами. Есть целый ряд важных приемов, которыми нужно овладеть, чтобы считать себя настоящим специалистом по работе с окнами. Несколько последующих разделов посвящено тому, как располагать, упорядочивать, передвигать, изменять размер и закрывать окна.

2.2.1. Получение ссылок на окна

Обычно оказывается полезным иметь ссылки на окна, открытые в приложении. Ссылки могут понадобиться для разных целей, например для задания положения окон, их упорядочения, передачи данных и пр.

Объект `NativeApplication` приложения AIR хранит данные обо всех окнах, открытых в приложении. Свойство `mainWindow` указывает на главное окно приложения (начальное окно). Свойство `openedWindows` хранит массив всех окон, открытых в данный момент в приложении. Можно также получить ссылку на объект `NativeWindow`, соответствующий объ-

екту Stage, через свойство nativeWindow объекта Stage. В следующих разделах мы рассмотрим разнообразные способы применения этих свойств.

2.2.2. Размещение окон

Размещение окон – важная задача, в которой нужно тщательно разобраться, прежде чем начинать интенсивно работать с окнами. В противном случае ваши окна могут отображаться в случайных местах, окажутся заслонены другими окнами или сами будут перекрывать другие открытые окна не так, как вам того бы хотелось.

Размещение объектов NATIVEWINDOW

Чтобы задать координаты окна x и y на рабочем столе, можно задать значения свойств x и y объекта NativeWindow. Если непосредственно располагать объекты NativeWindow, то нужный результат достигается легко. Вы просто устанавливаете свойства x и y сразу после создания объекта или в любой нужный момент. Однако если вы создаете компоненту Window с помощью Flex, нужно учитывать, что соответствующий объект NativeWindow для компоненты Window не создается в тот момент, когда вы строите новый объект Window. В следующем разделе мы увидим, как решать эту проблему.

Размещение объектов WINDOW

Как уже было сказано, объект NativeWindow, связанный с компонентой Window, не генерируется сразу при создании компоненты Window. Вам придется подождать, пока этот объект Window не сгенерирует событие windowComplete, и только после этого вы сможете получить доступ к связанному с ним объекту NativeWindow и задать свойства x и y. Два обычных способа для задания координат x и y для окна – из окна, которое создает новое окно, или внутри самого нового окна.

Если вы задаете положение нового окна из того окна, в котором оно создается, вам нужно зарегистрировать обработчик события, который будет ждать события windowComplete, и, когда оно произойдет, задаст свойства x и y объекта nativeWindow для нового окна. Это проиллюстрировано в листинге 2.14. Листинг 2.14 является модификацией листинга 2.9, и внесенные изменения выделены жирным шрифтом.

Листинг 2.14. Размещение нового окна из того окна, в котором оно создается

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import mx.events.AIREvent;

         private function creationCompleteHandler():void {
            var window:SimpleTextWindow = new SimpleTextWindow();
```

```
            window.addEventListener(
                    AIREvent.WINDOW_COMPLETE,
                     windowCompleteHandler);
            window.open();
        }

        private function windowCompleteHandler(event:AIREvent):void {
            event.target.stage.nativeWindow.x = 0;
            event.target.stage.nativeWindow.y = 0;
        }

    ]]>
  </mx:Script>
</mx:WindowedApplication>
```

❶ Зарегистрировать обработчик

❷ Задать координаты окна

В этом примере сразу после создания экземпляра окна мы регистрируем метод, который будет обработчиком события `windowComplete` ❶. Собственно метод, обрабатывающий событие ❷, устанавливает для окна свойства `x` и `y`.

В предыдущем примере было показано, как задать положение окна из того окна, которое его открыло. Если вы предпочитаете устанавливать положение окна в самом создаваемом окне, можно ждать события `windowComplete` в самой этой компоненте. Рассмотрим пример того, как это можно сделать. В листинге 2.15 показано окно `SimpleTextWindow` из листинга 2.8 с необходимыми изменениями, выделенными жирным шрифтом. В данном примере мы предполагаем, что документ приложения сохранил тот вид, какой он имел в листинге 2.11.

Листинг 2.15. Размещение окна из компоненты окна

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml"
   width="200" height="200" type="utility"
windowComplete="windowCompleteHandler();">
   <mx:Script>
      <![CDATA[

         private function windowCompleteHandler():void {
            nativeWindow.x = 0;
            nativeWindow.y = 0;
         }

      ]]>
   </mx:Script>
   <mx:Label text="New Window Content" />
</mx:Window>
```

В этих примерах мы рассмотрели, как размещать окна при их первом открытии. Конечно, свойствами `x` и `y` объекта `nativeWindow` можно воспользоваться для размещения окна в любой момент, а не только при его первоначальном открытии. Более того, тем же приемом можно воспользоваться для задания позиции главного окна при его инициализации.

Листинг 2.16 показывает один из способов расположить приложение в центре экрана.

Листинг 2.16. Размещение приложения в центре экрана

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute"
   windowComplete="windowCompleteHandler();"
   creationComplete="creationCompleteHandler();">

   <mx:Script>
      <![CDATA[
         import mx.events.AIREvent;

         private function creationCompleteHandler():void {
            var window:SimpleTextWindow = new SimpleTextWindow();
            window.open();
         }

         private function windowCompleteHandler():void {
            nativeWindow.x = (Capabilities.screenResolutionX - width) / 2;
            nativeWindow.y = (Capabilities.screenResolutionY - height) / 2;
         }

      ]]>
   </mx:Script>
</mx:WindowedApplication>
```

❶ Ждать события windowComplete

❷ Разместить по центру

В этом примере мы прежде всего сообщаем окну, как обрабатывать событие `windowComplete` ❶. В данном случае окно должно вызвать метод `windowCompleteHandler()` ❷, определенный в блоке `script`. Обработчик события использует разрешение экрана, которое извлекает из внутреннего объекта `Capabilities`, чтобы переместить окно в центр экрана.

Далее мы рассмотрим несколько более сложный вариант той же схемы, действующий с виртуальным рабочим столом.

Работа с виртуальным рабочим столом

В нынешние времена не столь редки системы с несколькими мониторами. Если у пользователя вашего приложения есть два или более мониторов, неплохо предоставить ему дополнительное экранное пространство при работе с вашим приложением. Приложения AIR дают пользователям возможность перетаскивать окна в любое место рабочего стола, включая дополнительные мониторы. Однако и при программном размещении окон нужно учесть возможность наличия дополнительных мониторов. Пользователи часто решают разместить на основном мониторе главное окно приложения, а вспомогательные окна – на втором мониторе. Но пользователь может с таким же успехом пожелать переместить главное окно приложения на свой второй монитор. В любом случае ваше приложение AIR покажет себя с лучшей стороны, если запомнит, куда пользователь помещал окна в предыдущий

раз, и восстановит их положение при новом запуске приложения. Сохранить настройки пользователя, в частности, расположение окон, можно в объекте SharedObject (это базовый прием ActionScript), в локальной базе данных (см. главу 5) или даже в текстовом файле (см. главу 3). В данный момент нас не интересует, где конкретно будут храниться данные. Нам сейчас важно определить правильные значения и выяснить, на каком из экранов находится окно.

Приложения AIR рассматривают все пространство всех мониторов как один виртуальный рабочий стол. Идею иллюстрирует рис. 2.7.

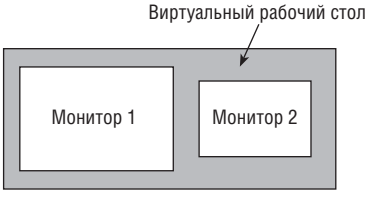


Рис. 2.7. *Приложения AIR рассматривают все мониторы как один виртуальный рабочий стол*

С позиции программирования у каждого из этих мониторов есть представление в виде объекта `flash.display.Screen`. Объект `Screen` предоставляет данные о размере экрана с помощью свойств `bounds` и `visibleBounds`. Так как приложения AIR не могут изменять разрешение мониторов, свойства `bounds` и `visibleBounds` доступны только для чтения. Оба свойства имеют тип `flash.geom.Rectangle`, а значения `x` и `y` отсчитываются относительно верхнего левого угла виртуального рабочего стола.

У класса `Screen` есть также два статических свойства, с помощью которых можно получить ссылки на объекты `Screen`, доступные приложению AIR. Свойство `Screen.mainScreen` возвращает ссылку на главный экран компьютера. Свойство `Screen.screens` возвращает массив, состоящий из всех экранов, в котором первой идет та же ссылка на `mainScreen`.

У класса `Screen` есть также статический метод `getScreensForRectangle()`. Этот метод позволяет получить массив объектов `Screen`, которые накрываются данным прямоугольником (относительно рабочего стола). Практическая польза его в том, что он позволяет выяснить, где находится конкретная область (возможно, окно) – на одном экране или на другом, на обоих или ни на каком.

Пример листинга 2.17 иллюстрирует некоторые понятия виртуального рабочего стола и экранов путем примитивно реализованного алгоритма привязки окон. Окно притягивается к краю экрана, на котором оно расположено, если от него до края экрана меньше 100 пикселов. Код примера в значительной мере повторяет листинг 2.2. Отличия показаны жирным шрифтом.

Листинг 2.17. Прилипание окон к краю экрана

```
package {

   import  flash.display.MovieClip;
   import  flash.display.NativeWindow;
   import  flash.display.NativeWindowInitOptions;
   import  flash.display.NativeWindowType;
   import  flash.display.NativeWindowSystemChrome;
   import  flash.events.NativeWindowBoundsEvent;
   import  flash.display.Screen;

   public class Example extends MovieClip {

      public function Example() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         var window:NativeWindow = new NativeWindow(options);
         window.width = 200;
         window.height = 200;

         window.activate();

         window.addEventListener(NativeWindowBoundsEvent.MOVE,
                                                 moveHandler);

      }

      private function moveHandler(event:NativeWindowBoundsEvent):void {

         var window:NativeWindow = event.target as NativeWindow;

         var screens:Array =
         ➥Screen.getScreensForRectangle(window.bounds);
         var screen:Screen;

         if(screens.length == 1) {
             screen = screens[0];
         }
         else if(screens.length == 2) {
             screen = screens[1];
         }

         if(window.x < screen.bounds.x + 100) {
             window.x = screen.bounds.x;
         }
         else if(window.x > screen.bounds.x + screen.bounds.width -
         ➥window.width - 100) {
             window.x = screen.bounds.x + screen.bounds.width -
             ➥window.width;
         }
         if(window.y < screen.bounds.y + 100) {
             window.y = screen.bounds.y;
         }
```

❶ Ждать события перемещения

❷ Определить перекрытие с экраном

❸ Выбрать экран для прилипания

❹ Прилипнуть к краям при расстоянии меньше 100 пикселов

```
            else if(window.y > screen.bounds.y + screen.bounds.height -
            ➥window.height - 100) {
                window.y = screen.bounds.y + screen.bounds.height -
                ➥window.height;
            }
        }
    }
}
```

Первое, что нужно сделать, – это поймать событие перемещения ❶, которое окно генерирует, когда пользователь его перетаскивает. При обработке события окну предписывается предпринять действия. Сначала оно должно получить массив всех экранов, с которыми перекрывается окно, воспользовавшись методом `Screen.getScreensForRectangle()` ❷. Затем принимается решение: если окно перекрывается только с одним экраном, получить ссылку на этот экран; в противном случае – выбрать второй из экранов, с которыми есть перекрытие ❸. Остальная логика ❹ проверяет, к какому краю ближе окно, и привязывает к нему, если до края меньше 100 пикселов.

Пример демонстрирует несколько важных приемов работы с экранами, в том числе определение экрана, на котором находится окно, и получение границ экрана. Следующей будет важная тема закрытия окон.

2.2.3. Закрытие окон

Может показаться, что это простая задача, но на самом деле она важна и имеет ряд нюансов. В следующих разделах мы посмотрим, как решать несколько проблем, возникающих в связи с закрытием окон, в том числе повторное открытие закрытых окон, закрытие всех окон при выходе из приложения и закрытие окон без элементов оформления.

Повторное открытие закрытых окон

Изучая понятия, описываемые в этой главе, вы, вероятно, запускали код соответствующих примеров и создали немалое количество окон. При этом вы могли заметить, что если окно имеет элементы оформления, то автоматически активизируется кнопка закрытия, позволяющая пользователю закрыть окно. Несмотря на все удобство, это может стать источником проблем, потому что в приложении AIR нельзя снова открыть окно, если оно закрыто. Хотя в некоторых случаях такое поведение оправдано, во многих других хотелось бы разрешить пользователю снова открыть окно после того, как он его закрыл. Возьмите, к примеру, вспомогательное окно с палитрой цветов в графическом редакторе. У пользователя должна быть возможность показывать и скрывать это окно. Если оставить для этого окна режим закрытия по умолчанию, AIR не даст снова показать окно, если пользователь закроет его. Листинг 2.18 содержит пример, иллюстрирующий эту проблему.

Листинг 2.18. Нельзя снова открыть окно, которое было закрыто

```
package {

   import flash.display.NativeWindowType;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindow;
   import flash.display.MovieClip;
   import flash.events.MouseEvent;

   public class WindowExample extends MovieClip {

      private var _window:NativeWindow;

      public function WindowExample() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         _window = new NativeWindow(options);
         _window.width = 200;
         _window.height = 200;

         _window.activate();

         stage.addEventListener(MouseEvent.CLICK, openWindow);
      }

      private function openWindow(event:MouseEvent):void {
         _window.activate();
      }

   }

}
```

Замысел этого примера состоял в том, чтобы при каждом щелчке пользователя по главному окну приложение вызывало вспомогательное окно с помощью метода `activate()`. Однако если протестировать этот пример, можно обнаружить, что если закрыть вспомогательное окно, то любая попытка повторно открыть его приводит к ошибке времени исполнения, сообщающей о невозможности открытия ранее закрытого окна.

Как решить эту проблему? Ответ проще, чем можно предположить: вместо закрытия окна при щелчке по кнопке закрытия нужно просто сделать его невидимым. При этом окно исчезнет, но его можно будет снова открыть. Чтобы реализовать такую схему, нужно перехватить управление окном после щелчка пользователя по кнопке закрытия и до того, как AIR успеет выполнить назначенное действие и действительно закрыть окно. Это можно сделать, если перехватить событие закрытия. Все окна генерируют событие типа `Event.CLOSING` немедленно после того, как пользователь щелкает по кнопке закрытия, но перед тем, как окно действительно закроется. При обработке этого события нужно сделать две вещи: задать для свойства `visible` значение `false` и запретить поведение по умолчанию. Последнее совершенно необходимо, потому что иначе AIR закроет окно. В ActionScript отменить действие по

умолчанию для отменяемых событий (в том числе события закрытия окна) можно с помощью метода `preventDefault()` объекта события. Листинг 2.19 показывает, как перехватить событие закрытия окна, изменить видимость и не допустить действия, выполняемого по умолчанию.

Листинг 2.19. Сокрытие и показ окна пользователем

```
package {

   import flash.display.NativeWindowType;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindow;
   import flash.display.MovieClip;
   import flash.events.MouseEvent;
   import flash.events.Event;

   public class WindowExample extends MovieClip {

      private var _window:NativeWindow;

      public function WindowExample() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         _window = new NativeWindow(options);
         _window.width = 200;
         _window.height = 200;

         _window.activate();

         stage.addEventListener(MouseEvent.CLICK, openWindow);

         _window.addEventListener(Event.CLOSING, closingHandler);
      }

      private function closingHandler(event:Event):void {
         _window.visible = false;
         event.preventDefault();
      }

      private function openWindow(event:MouseEvent):void {
        _window.activate();
      }

   }

}
```

Как видите, элементарное действие – закрытие окон – при создании приложений приходится организовывать, если вы хотите, чтобы пользователи могли снова открыть окна, которые они закрыли.

Закрытие всех окон по завершении приложения

Как вы могли заметить по предыдущим примерам, окна, которые открываются из начального окна приложения, не закрываются автоматически при закрытии начального окна приложения. Такое поведение

оправдано, если дополнительные окна являются стандартными окнами, отображаемыми в панели задач или оконном меню. Однако если дополнительные окна являются вспомогательными или облегченными, то, скорее всего, было бы желательно закрывать их, когда закрывается главное окно приложения, с которым они связаны.

Если воспользоваться свойством `openedWindows` объекта `NativeApplication`, то можно получить массив ссылок на все объекты `NativeWindow` данного приложения. Обойдя все элементы массива `openedWindows`, можно программно закрыть все вспомогательные и облегченные окна. Обычно это нужно сделать при закрытии приложения. Узнать, что приложение закрывается, можно, если ловить появление события выхода, которое сгенерирует объект `NativeApplication`. Закрыть окно программным образом можно с помощью метода `close()` объекта `NativeWindow`. Листинг 2.20 демонстрирует, как можно программно закрыть все окна, когда закрывается главное окно приложения.

Листинг 2.20. Закрытие окон по завершении приложения

```
package {

   import  flash.display.MovieClip;
   import  flash.display.NativeWindow;
   import  flash.display.NativeWindowInitOptions;
   import  flash.display.NativeWindowType;
   import  flash.desktop.NativeApplication;
   import  flash.events.Event;

   public class Example extends MovieClip {

      public function Example() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         var window:NativeWindow = new NativeWindow(options);
         window.width = 200;
         window.height = 200;

         window.activate();

         this.stage.nativeWindow.addEventListener(
         ➥Event.CLOSING, closingHandler);                    ❶ Ждать событие выхода
      }

      private function closingHandler(event:Event):void {
         var windows:Array =
         ➥NativeApplication.nativeApplication.openedWindows;
         for(var i:Number = 0; i < windows.length; i++) {
             windows[i].close();                              ❷ Закрыть все открытые окна
         }
      }

   }

}
```

Чтобы узнать, когда закрывать все окна, нужно ждать событие закрытия, которое генерирует экземпляр `NativeApplication` ❶. Затем, во время обработки этого события ❷, мы обходим все элементы массива `openedWindows` объекта `NativeApplication` и вызываем для каждого окна метод `close()`.

Мы узнали, как можно закрыть все открытые окна, когда происходит событие закрытия главного окна. Если при завершении приложения не закрыть все окна, у пользователя могут возникнуть проблемы. Если окно скрыто, но формально остается открытым, пользователь не сможет никаким способом закрыть его после выхода из приложения. Это приведет к ненужному расходу ресурсов системы.

Добавление в окна специфического механизма закрытия

Как мы узнали в предыдущем разделе, можно закрыть окно с помощью метода `close()`. Тот же метод можно использовать для закрытия окна с помощью особого элемента интерфейса, специально добавленного, чтобы дать пользователю возможность закрыть окно. Это важная идея для окон, не имеющих элементов оформления (например, окон неправильной формы), поскольку по умолчанию у них нет кнопки закрытия окна.

2.2.4. Упорядочение окон

Взаимное расположение окон вдоль оси z может быть организовано разными способами. Обычно окно с фокусом ввода располагается поверх всех остальных, и обычно пользователи желают сами управлять порядком укладки окон на своем рабочем столе. Однако бывают веские причины, чтобы управлять порядком окон программным образом, в частности, следующие:

- При создании новых окон бывает предпочтительным открывать их или на переднем плане, или под всеми существующими окнами.
- Может потребоваться, чтобы окно всегда располагалось поверх всех остальных окон, даже если оно не обладает фокусом. Это удобно, если окно требует внимания пользователя или содержит данные, которые всегда должны быть видны ему даже при работе с другими приложениями.
- Может оказаться желательным, чтобы вспомогательные окна выдвигались вперед относительно других приложений, когда соответствующее приложение AIR получает фокус.

Есть ряд методов и одно свойство, которые управляют порядком окон. Все эти методы и свойство применимы к объектам `NativeWindow` во Flash и компонентам `Window` во Flex. Вот эти методы и свойство:

- `orderToFront()` – поместить окно поверх всех остальных окон данного приложения AIR.
- `orderToBack()` – поместить окно под всеми остальными окнами данного приложения AIR.

- `orderInFrontOf(window:IWindow)` — поместить одно окно впереди другого.
- `orderInBackOf(window:IWindow)` — поместить одно окно позади другого.
- `alwaysInFront` — такое окно всегда размещается поверх всех других окон на рабочем столе.

Как можно было убедиться по предыдущим примерам, если в приложении есть вспомогательное окно и фокус переходит к другому приложению, то при возврате фокуса к приложению AIR его вспомогательное окно не выходит на передний план относительно окон других приложений, выполняющихся в системе. Из-за этого вспомогательные окна могут легко «потеряться» за другими приложениями. В приложениях AIR со вспомогательными окнами полезной оказывается такая функция, как автоматическое выведение этих вспомогательных окон на передний план вместе с главным окном приложения, когда оно получает фокус. Когда приложение теряет фокус, объект `NativeApplication` генерирует событие деактивации; когда приложение получает фокус, оно генерирует событие активации. Поэтому, если перехватить событие активации, обойти все открытые окна и вывести их на передний план, вы гарантированно не потеряете свои окна за другими приложениями. Простой пример того, как это сделать, содержит листинг 2.21.

Листинг 2.21. Перемещение вспомогательных окон на передний план вместе с главным окном приложения

```
package {

   import  flash.display.MovieClip;
   import  flash.display.NativeWindow;
   import  flash.display.NativeWindowInitOptions;
   import  flash.display.NativeWindowType;
   import  flash.desktop.NativeApplication;
   import  flash.events.Event;
   import  flash.text.TextField;
   import  flash.text.TextFieldAutoSize;
   import  flash.display.StageScaleMode;
   import  flash.display.StageAlign;

   public class Example extends MovieClip {

      public function Example() {

         var options:NativeWindowInitOptions =
         ➥new NativeWindowInitOptions();
         options.type = NativeWindowType.UTILITY;

         var window:NativeWindow = new NativeWindow(options);
         window.width = 200;
         window.height = 200;

         var textField:TextField = new TextField();
         textField.autoSize = TextFieldAutoSize.LEFT;
         textField.text = "New Window Content";
```

```
            window.stage.scaleMode = StageScaleMode.NO_SCALE;
            window.stage.align = StageAlign.TOP_LEFT;

            window.stage.addChild(textField);

            window.activate();

            this.stage.nativeWindow.addEventListener(Event.ACTIVATE,
                                        activateHandler);
        }

        private function activateHandler(event:Event):void {
            var windows:Array =
            ➥NativeApplication.nativeApplication.openedWindows;
            for(var i:Number = 0; i < windows.length; i++) {
                windows[i].orderToFront();
            }
        }

    }

}
```

Если вы протестируете приведенный код, то увидите, что всякий раз, когда вы переводите окно приложения на передний план, все вспомогательные окна также перемещаются вперед.

2.2.5. Перемещение окон и изменение их размеров

Когда у окна есть рамка и заголовок, пользователь может передвигать окно и изменять его размер с помощью стандартных элементов интерфейса (перемещать окно перетаскиванием панели заголовка и изменять размер перетаскиванием рамок). Если же вы показываете окно без этих системных элементов оформления, пользователь лишается такой возможности. Если вы все же хотите предоставить ему такие возможности, это нужно сделать программным способом. К счастью, AIR облегчает эту задачу.

У каждого объекта `NativeWindow` есть методы `startMove()` и `startResize()`. Если вызывать эти методы в обработчике события `mouseDown`, то событие `mouseUp` автоматически прекращает перемещение или изменение размеров – в полном соответствии с ожиданиями пользователя.

Метод `startMove()` не требует параметров. Можете просто вызвать `startMove()` для окна, если пользователь щелкнет по объекту. После этого окно будет перемещаться вместе с мышью, пока пользователь не отпустит кнопку. Пример этого приведен в листинге 2.22, содержащем модифицированный код `ExampleWindow` из листинга 2.7.

Листинг 2.22. Инициация перетаскивания окна с помощью startMove( )

```
package {

    import flash.display.NativeWindow;
    import flash.display.NativeWindowSystemChrome;
```

```
import flash.display.NativeWindowType;
import flash.display.NativeWindowInitOptions;
import flash.display.Sprite;
import flash.display.Stage;
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.events.MouseEvent;
import flash.events.Event;


public class ExampleWindow extends NativeWindow {

   private var _background:Sprite;

   public function ExampleWindow() {
      var options:NativeWindowInitOptions =
      ➥new NativeWindowInitOptions();

      options.systemChrome = NativeWindowSystemChrome.NONE;
      options.type = NativeWindowType.LIGHTWEIGHT;
      options.transparent = true;

      super(options);
      _background = new Sprite();
      drawBackground(200, 200);
      stage.addChild(_background);

      width = 200;
      height = 200;

      stage.align = StageAlign.TOP_LEFT;
      stage.scaleMode = StageScaleMode.NO_SCALE;
      _background.addEventListener(MouseEvent.MOUSE_DOWN,
                                   startMoveWindow);
   }

   private function drawBackground(newWidth:Number, newHeight:Number):
   ➥void  {
      _background.graphics.clear();
      _background.graphics.lineStyle(0, 0, 0);
      _background.graphics.beginFill(0x0000FF, .5);
      _background.graphics.drawRoundRectComplex(0, 0, newWidth,
                                 newHeight, 20, 20, 20, 1);
      _background.graphics.beginFill(0xFFFFFF, .9);
      _background.graphics.drawRoundRectComplex(5, 5, newWidth - 10,
                                 newHeight - 10, 20, 20, 20, 1);
      _background.graphics.endFill();
   }

   private function startMoveWindow(event:MouseEvent):void {
      startMove();
   }

}
```

Методу `startResize()` нужно указать край или угол окна, с которого начинается изменение размера. Допустимые значения представлены константами класса `flash.display.NativeWindowResize`: `TOP`, `BOTTOM`, `LEFT`, `RIGHT`, `TOP_LEFT`, `TOP_RIGHT`, `BOTTOM_LEFT` и `BOTTOM_RIGHT` (верхний, нижний, левый, правый, верхний левый, верхний правый, нижний левый и нижний правый). Например, если вызвать `startResize()` с параметром `NativeWindowResize.BOTTOM_RIGHT`, то правый нижний угол окна будет перемещаться вместе с мышью, в то время как левый верхний угол будет сохранять свое положение. Можно вызывать `startResize()`, когда пользователь щелкает по какому-нибудь объекту. Окно будет изменять свои размеры в соответствии с движениями мыши, пока пользователь не отпустит кнопку мыши. Листинг 2.23 содержит пример кода, меняющего размер окна. Код основан на окне из листинга 2.22.

Листинг 2.23. Изменение размеров окна

```
package {

   import flash.display.NativeWindow;
   import flash.display.NativeWindowSystemChrome;
   import flash.display.NativeWindowType;
   import flash.display.NativeWindowInitOptions;
   import flash.display.Sprite;
   import flash.display.Stage;
   import flash.display.StageAlign;
   import flash.display.StageScaleMode;
   import flash.events.MouseEvent;
   import flash.events.Event;
   import flash.display.NativeWindowResize;

   public class ExampleWindow extends NativeWindow {

      private var _background:Sprite;

      private var _resizer:Sprite;

      public function ExampleWindow() {
      var options:NativeWindowInitOptions =
      ➥new NativeWindowInitOptions();

      options.systemChrome = NativeWindowSystemChrome.NONE;
      options.type = NativeWindowType.LIGHTWEIGHT;
      options.transparent = true;

      super(options);
      _background = new Sprite();
      drawBackground(200, 200);
      stage.addChild(_background);

      width = 200;
      height = 200;

      stage.align = StageAlign.TOP_LEFT;
      stage.scaleMode = StageScaleMode.NO_SCALE;
```

```
            _background.addEventListener(MouseEvent.MOUSE_DOWN,
                                         startMoveWindow);

            _resizer = new Sprite();
            _resizer.graphics.lineStyle(0, 0, 0);
            _resizer.graphics.beginFill(0xCCCCCC, 1);
            _resizer.graphics.drawRect(0, 0, 10, 10);
            _resizer.graphics.endFill();
            _resizer.x = 180;
            _resizer.y = 180;
            stage.addChild(_resizer);

            _resizer.addEventListener(MouseEvent.MOUSE_DOWN,
                                         startResizeWindow);

            addEventListener("resizing", resizingHandler);

        }

        private function drawBackground(newWidth:Number, newHeight:Number):
          ➥void  {
            _background.graphics.clear();
            _background.graphics.lineStyle(0, 0, 0);
            _background.graphics.beginFill(0x0000FF, .5);
            _background.graphics.drawRoundRectComplex(0, 0, newWidth,
                                       newHeight, 20, 20, 20, 1);
            _background.graphics.beginFill(0xFFFFFF, .9);
            _background.graphics.drawRoundRectComplex(5, 5, newWidth - 10,
                                       newHeight - 10, 20, 20, 20, 1);
            _background.graphics.endFill();

        }

        private function startMoveWindow(event:MouseEvent):void {
            startMove();
        }

        private function resizingHandler(event:Event):void {
            drawBackground(width, height);
            _resizer.x = width - 20;
            _resizer.y = height - 20;
        }

        private function startResizeWindow(event:MouseEvent):void {
            startResize(NativeWindowResize.BOTTOM_RIGHT);
        }

    }

}
```

Вы могли заметить, что в этом примере окно генерирует событие изменения размера. С помощью этого события можно перерисовывать содержимое окна или изменять его структуру желаемым образом.

Мы осветили все темы, касающиеся поведения окон, и готовы перейти к более высокому уровню, на котором рассматривается поведение

приложения в целом. Соответствующие темы, включая обнаружение бездействия и перехода в полноэкранный режим, рассматриваются в следующем разделе.

2.3. Управление приложением

Вы сейчас знакомы со всеми основными сведениями, касающимися окон, необходимыми для построения приложений AIR. Теперь мы рассмотрим вопросы уровня приложения, относящиеся к управлению ими. Мы рассмотрим такие темы:

- Обнаружение бездействия пользователя
- Запуск приложений AIR одновременно с запуском системы
- Привязка файлов к приложениям
- Выдача пользователю предупредительных сообщений
- Выполнение приложений в полноэкранном режиме

Как будет показано ниже, AIR упрощает решение всех этих задач.

2.3.1. Обнаружение бездействия пользователя

Часто бывает полезно или необходимо определить, что пользователь больше не работает с приложением. Например, приложение для обмена мгновенными сообщениями может автоматически установить для пользователя статус «отсутствует» или «занят», если он отошел от компьютера. Приложения AIR автоматически определяют, что пользователь не обращался к приложению в течение заданного промежутка времени. Как только достигнут заданный порог, объект `NativeApplication` генерирует событие `userIdle`. Когда пользователь возвращается к приложению, объект `NativeApplication` генерирует событие `userPresent`. Оба события имеют тип `flash.events.Event`, а имена событий заданы константами `Event.USER_IDLE` и `Event.USER_PRESENT`.

По умолчанию тайм-аут составляет 300 секунд. Это значение можно изменить с помощью свойства `idleThreshold` объекта `NativeApplication`. Значение должно быть целым числом, которое определяет количество секунд, в течение которого не должно отмечаться никакой активности, чтобы объект `NativeApplication` сгенерировал событие `userIdle`.

Кроме того, можно узнать, сколько времени прошло с того момента, когда в последний раз была отмечена активность пользователя. Свойство `timeSinceLastUserInput` объекта `NativeApplication` сообщит вам, сколько секунд прошло с момента получения приложением входных данных от мыши или клавиатуры.

2.3.2. Запуск приложений при входе в систему

Обычно приложение AIR запускается, только если пользователь сделает двойной щелчок по приложению или файлу, тип которого с ним

ассоциирован. Однако можно пометить приложение AIR таким образом, что оно будет автоматически запускаться при входе пользователя в систему. Для этого достаточно задать для свойства `startAtLogin` объекта `NativeApplication` значение `true`.

Учтите, что если `startAtLogin` имеет значение `true`, то приложение должно быть установлено в системе. Если вы просто проверите, что получится при запуске вашего приложения из Flash или Flex Builder, то возникнет ошибка этапа исполнения.

2.3.3. Привязка файлов к приложениям

Устанавливая связь файла с приложением, вы даете пользователю возможность автоматически запускать приложение AIR при щелчке по файлу этого типа. В главе 1 было показано, как задать типы файлов для приложения с помощью файла дескриптора. Для перечисленнных в дескрипторе типов файлов возможны два варианта дальнейших событий: если этот тип файла еще не связан ни с каким другим приложением в системе, он автоматически регистрируется за данным приложением AIR; если тип файла привязан к другому приложению, новая связь не образуется. Во втором случае вы можете только программно переопределить имеющуюся связь. Сделать это можно лишь на этапе исполнения с помощью методов, перечисленных в табл. 2.1.

Если вы хотите создать связь с файлами типов, обычно используемых другими приложениями, полезно бывает попросить у пользователя разрешения установить новую связь. Допустим, например, что вы ассоциировали файлы .mp3 со своим приложением AIR посредством файла дескриптора (и, следовательно, не спрашивая разрешения пользователя). Поскольку у многих пользователей уже выбрана связь для файлов .mp3, им может быть неприятно, что вы изменили их настройки без их разрешения. Поэтому, хотя можно программно задать связь типа файла с приложением без разрешения пользователя, лучше все же предварительно задать ему вопрос.

Таблица 2.1. Методы изменения связи файлов с приложениями

Имя метода	Описание
`setAsDefaultApplication()`	Задать текущее приложение AIR в качестве приложения по умолчанию для конкретного типа файла.
`removeAsDefaultApplication()`	Удалить связь между текущим приложением AIR и некоторым типом файла.
`isSetAsDefaultApplication()`	Узнать, является ли текущее приложение AIR приложением по умолчанию для файла данного типа.
`getDefaultApplication()`	Получить имя приложения по умолчанию для данного типа файла.

Чтобы ассоциировать типы файлов с приложением на этапе исполнения, воспользуйтесь методом `setAsDefaultApplication()` объекта `NativeApplication`. Этот метод требует передачи одного параметра, задающего расширение имени для того типа файлов, которые вы хотите связать с приложением AIR. Значение параметра должно содержать расширение в виде строки, без начальной точки. Например, следующий код ассоциирует файлы .mp3 с приложением, выполняемым в данный момент:

```
NativeApplication.nativeApplication.setAsDefaultApplication("mp3");
```

Если вы хотите разорвать связь, можно просто вызвать метод `removeAsDefaultApplication()`, передав ему расширение типа файла, связь с которым вы хотите прекратить:

```
NativeApplication.nativeApplication.removeAsDefaultApplication("mp3");
```

Метод `isSetAsDefaultApplication()` возвращает булево значение, указывающее, является ли приложение AIR обработчиком по умолчанию для файлов с указанным расширением:

```
var isDefault:Boolean = NativeApplication.nativeApplication.
➥isSetAsDefaultApplication("mp3");
```

Можно также с помощью метода `getDefaultApplication()` получить имя приложения, с которым ассоциирован данный тип файлов. Этот метод требует передачи одного параметра, являющегося строкой с расширением имени:

```
var defaultApplication:String = NativeApplication.nativeApplication.
➥getDefaultApplication("mp3");
```

Все эти методы действуют только с типами файлов, включенными в раздел fileTypes файла дескриптора. Кроме того, нужно учитывать, что они будут действовать только после установки приложения AIR. Это означает, что при тестировании приложения во Flash или Flex Builder они будут работать некорректно.

2.3.4. Оповещение пользователя

Иногда приложению AIR нужно известить пользователя о каком-то событии, требующем его внимания, даже когда приложение минимизировано или не имеет фокуса. В операционных системах есть стандартные средства оповещения для таких случаев. Например, в Windows начинает мигать соответствующий элемент в панели задач, а в OS X элемент подпрыгивает в доке приложений. AIR дает возможность оповестить пользователя стандартным образом с помощью метода `notifyUser()` объекта `NativeWindow`.

Метод `notifyUser()` требует передачи параметра, содержащего одну из двух констант класса `flash.desktop.NotificationType`: `NotificationType.INFORMATIONAL` или `NotificationType.CRITICAL`. Эти два типа соответствуют двум типам оповещений, допускаемых операционной системой.

Следующий пример (листинг 2.24) иллюстрирует оповещение. Каждые пять секунд приложение проверяет, активно ли главное окно (имеет ли оно фокус). Если нет, пользователю посылается сообщение.

Листинг 2.24. Предупреждение пользователя о том, что главное окно не активно

```
package {

   import  flash.display.MovieClip;
   import  flash.display.NativeWindow;
   import  flash.desktop.NativeApplication;
   import  flash.utils.Timer;
   import  flash.events.TimerEvent;
   import  flash.desktop.NotificationType;

   public class Example extends MovieClip {

      private var _timer:Timer;

      public function Example() {

         _timer = new Timer(5000);

         _timer.addEventListener(TimerEvent.TIMER, timerHandler);
         _timer.start();

      }

      private function timerHandler(event:TimerEvent):void {
         var mainWindow:NativeWindow =
➥NativeApplication.nativeApplication.openedWindows[0] as NativeWindow;
         if(!mainWindow.active) {
             mainWindow.notifyUser(NotificationType.INFORMATIONAL);
         }
      }

   }

}
```

Как видите, известить пользователя о том, что некоторое окно требует его внимания, оказывается достаточно просто.

Беспроигрышный способ привлечь внимание пользователя – это запустить приложение в режиме полного экрана. В следующем разделе мы узнаем, как это делается.

2.3.5. Полноэкранный режим

Запустить приложение в полноэкранном режиме можно с помощью свойства `displayState` объекта `Stage` окна. Установите для этого свойства значение `StageDisplayState.FULL_SCREEN`, как в примере листинга 2.25. Учтите, что поскольку эти приложения выполняются в полноэкранном режиме, не имея стандартных механизмов завершения приложения, вы должны закрывать приложение с помощью стандартных горячих клавиш вашей операционной системы.

Листинг 2.25. Открытие окна в режиме полного экрана

```
package {

   import flash.display.MovieClip;
   import flash.text.TextField;
   import flash.display.StageDisplayState;

   public class Example extends MovieClip {

      public function Example() {

         stage.displayState = StageDisplayState.FULL_SCREEN;

         var textField:TextField = new TextField();
         textField.text = "Full screen example";
         addChild(textField);

      }

   }

}
```

Для иллюстрации принципа в данном примере используется текстовое поле. Свойства по умолчанию объекта `Stage` задают масштабирование объекта. Если вы не хотите, чтобы содержимое масштабировалось, нужно задать свойство `scaleMode`. Часто оказывается также желательным модифицировать свойство `align`. В данном примере текст может заметно увеличиться (это зависит от исходных размеров окна и разрешения экрана). В листинге 2.26 в код внесены изменения, которые требуют не масштабировать содержимое и выровнять его по левому верхнему углу.

Листинг 2.26. Настройка режима масштабирования и выравнивания полноэкранного окна

```
package {

   import flash.display.MovieClip;
   import flash.text.TextField;
   import flash.display.StageDisplayState;
   import flash.display.StageScaleMode;
   import flash.display.StageAlign;

   public class Example extends MovieClip {

      public function Example() {

         stage.displayState = StageDisplayState.FULL_SCREEN;

         stage.scaleMode = StageScaleMode.NO_SCALE;
         stage.align = StageAlign.TOP_LEFT;

         var textField:TextField = new TextField();
         textField.text = "Full screen example";
         addChild(textField);

      }

   }

}
```

Мы охватили все основные темы управления на уровне приложения. Теперь мы перейдем к новому вопросу: действиям с меню.

2.4. Меню

Приложения AIR могут использовать меню разными способами. Меню могут появляться на уровне приложения или окна, они могут быть контекстными и могут быть всплывающими. Ниже мы узнаем, как создавать меню, и применим все указанные способы.

2.4.1. Создание меню

Все меню в приложениях AIR имеют тип `flash.display.NativeMenu`. Новое меню создается с помощью нового объекта `NativeMenu`. Конструктор не требует параметров:

```
var exampleMenu:NativeMenu = new NativeMenu();
```

Это – все, что касается создания меню. Теперь мы посмотрим, как добавлять в меню элементы.

2.4.2. Добавление элементов в меню

После того как меню создано, можно добавлять в него элементы. Элементы могут в целом быть одного из двух типов: пункты меню или другие меню (подменю).

Пункты меню имеют тип `flash.display.NativeMenuItem`. Создать новый `NativeMenuItem` можно с помощью конструктора, которому передается название пункта. Это название показывается в меню:

```
var item:NativeMenuItem = new NativeMenuItem("Example Item");
```

Новый пункт добавляется в меню с помощью метода `addItem()` (или метода `addItemAt()`, если вы хотите вставить пункт не в конец меню, а в заданной позиции):

```
exampleMenu.addItem(item);
```

Как уже говорилось, можно добавлять в меню другие меню, создавая подменю. Для этого применяется метод `addSubmenu()` (или `addSubmenuAt()`), которому передается добавляемое меню:

```
exampleMenu.addItem(submenu);
```

Теперь вы знаете, как создавать меню и добавлять в них элементы. Однако чтобы меню функционировало, нужно уметь определять, что пользователь выбрал некий пункт меню. О том, как это делается, говорится в следующем разделе.

2.4.3. Перехват события – выбора пункта меню

Когда пользователь выбирает некий пункт меню, этот пункт (объект `NativeMenuItem`) генерирует событие выбора. Программист должен доба-

вить обработчик событий меню, чтобы приложение могло реагировать на выбор пользователя.

Событие выбора имеет тип `flash.events.Event`. Свойство `target` объекта события указывает на объект `NativeMenuItem`, который сгенерировал событие. Иногда с пунктами меню нужно связать определенные данные, чтобы приложение могло предпринять правильные и зависящие от контекста действия, когда пользователь выбирает этот пункт. Для этого свойству `data` объекта `NativeMenuItem` можно присвоить любые данные. Образец этого есть в примере, который приведен в листинге 2.27.

Есть еще одна тема, которую нужно обсудить, прежде чем вставлять меню в свои приложения: особые пункты меню вроде помеченных галочкой или линий-разделителей. Этим мы сейчас и займемся.

2.4.4. Создание особых пунктов меню

Обычные пункты меню выглядят просто как текст. Однако можно применять и специальные типы пунктов: помеченные галочкой и разделители.

Пунктами с галочкой удобно пользоваться, когда пункт меню включает и выключает какую-то опцию или позволяет выбрать элемент из списка. Чтобы показать галочку рядом с пунктом, нужно установить свойство `checked` объекта `NativeMenuItem`.

Разделители – это специальные пункты, осуществляющие разделение пунктов меню на логические группы. Они представляют собой обычные объекты `NativeMenuItem`, у которых свойство `isSeparator` установлено в `true`. Когда свойство `isSeparator` имеет значение `true`, этот пункт отображается в виде линии, и его нельзя выбрать.

Но достаточно теории. Теперь нужно рассмотреть несколько практических примеров меню в действии. В следующем разделе мы рассмотрим различные способы применения меню.

2.4.5. Применение меню

В приложениях AIR меню могут применяться различными способами: меню приложений и окон, меню иконок приложений в доке или в области уведомлений панели задач, контекстные и всплывающие меню. Во всех этих случаях меню являются экземплярами `NativeMenu`. Различаются эти случаи только способами применения меню.

Меню приложений или окон

Меню приложений и окон выполняют одну и ту же роль. Но меню приложений есть только в OS X, а меню окон есть только в Windows. На рис. 2.8 приведен пример меню окна.

Чтобы применить меню к окну, нужно присвоить его свойству `menu` того объекта `NativeWindow`, к которому нужно применить меню. Чтобы при-

Рис. 2.8. *Меню окна находится в верхней части, и вид его знаком большинству пользователей компьютеров*

менить меню к приложению, нужно присвоить его свойству `menu` объекта `NativeApplication`. Чтобы выяснить, какой тип меню поддерживает операционная система, можно воспользоваться статическим свойством `supportsMenu` классов `NativeWindow` и `NativeApplication`. Если `NativeWindow.supportsMenu` имеет значение `true`, то можно задать значение свойства `menu` объекта `NativeWindow`. Если свойство `NativeApplication.supportsMenu` имеет значение `true`, то можно задать свойство `menu` объекта `NativeApplication` данного приложения. Поскольку приложения меню и окна обычно предназначены для выполнения одних и тех же задач, после создания объекта `NativeMenu` обычно требуется выполнить код, сходный с приведенным ниже. В последующем коде можно считать, что `customMenu` является объектом типа `NativeMenu`, созданным ранее:

```
if(NativeApplication.supportsMenu) {
   NativeApplication.nativeApplication.menu = customMenu;
}
else if(NativeWindow.supportsMenu) {
   NativeApplication.nativeApplication.openedWindows[0].menu = customMenu;
 }
```

В листинге 2.27 приводится полный действующий пример меню окна или приложения, в котором использованы многие из обсуждавшихся в этом разделе понятий. В этом примере создается меню, с помощью которого пользователь может сохранить (не функционирует) или закрыть приложение, открыть новое окно и переключить фокус между открытыми окнами.

Листинг 2.27. Создание меню приложений или окон

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeWindow;
   import flash.display.NativeWindowInitOptions;
   import flash.display.NativeWindowType;
   import flash.desktop.NativeApplication;
   import flash.display.NativeMenu;
   import flash.display.NativeMenuItem;
   import flash.events.Event;
```

```
public class Example extends MovieClip {

   private var _windowsMenu:NativeMenu;

   public function Example() {

      var mainMenu:NativeMenu = new NativeMenu();

      var applicationMenu:NativeMenu = new NativeMenu();

      var save:NativeMenuItem = new NativeMenuItem("save");
      var close:NativeMenuItem = new NativeMenuItem("close");

      close.addEventListener(Event.SELECT, selectHandler);

      applicationMenu.addItem(save);
      applicationMenu.addItem(close);

      _windowsMenu = new NativeMenu();

      var newWindow:NativeMenuItem =
        ➥ new NativeMenuItem("new window");
        newWindow.addEventListener(Event.SELECT, selectHandler);

        var line:NativeMenuItem = new NativeMenuItem("", true);

      _windowsMenu.addItem(newWindow);
      _windowsMenu.addItem(line);

      mainMenu.addSubmenu(applicationMenu, "Application");
      mainMenu.addSubmenu(_windowsMenu, "Windows");

      var mainWindow:NativeWindow =
NativeApplication.nativeApplication.openedWindows[0] as NativeWindow;

      if(NativeApplication.supportsMenu) {
         NativeApplication.nativeApplication.menu = mainMenu;
      }
      else if(NativeWindow.supportsMenu) {
          mainWindow.menu = mainMenu;
      }

      mainWindow.addEventListener(Event.CLOSE, closeAll);

   }

   private function closeAll(event:Event = null):void {
      var windows:Array =
      ➥ NativeApplication.nativeApplication.openedWindows;
      for(var i:Number = 0; i < windows.length; i++) {
         windows[i].removeEventListener(Event.CLOSE, closeHandler);
         windows[i].close();
      }

   }

   private function selectHandler(event:Event):void {
      if(event.target.label == "close") {
         closeAll();
      }
```

1. Создать главное меню
2. Создать меню приложения
3. Слушать событие select
4. Добавить пункты меню
5. Создать меню окна
6. Слушать события select
7. Создать пункт-разделитель
8. Добавить подменю
9. Ссылка на главное окно
10. Применить меню
11. Слушать событие close
12. Закрыть открытые окна
13. Удалить слушателя во избежание ошибки
14. Закрыть все окна

```
            else if(event.target.label == "new window") {
                var windowTitle:String = "Window " +
NativeApplication.nativeApplication.openedWindows.length;
                var options:NativeWindowInitOptions =
new NativeWindowInitOptions();
                options.type = NativeWindowType.UTILITY;
                var window:NativeWindow = new NativeWindow(options);
                window.width = 200;
                window.height = 200;
                window.title = windowTitle;

                window.addEventListener(Event.ACTIVATE, activateHandler);
                window.addEventListener(Event.CLOSE, closeHandler);

                var menuItem:NativeMenuItem = new NativeMenuItem();
                menuItem.label = windowTitle;
                menuItem.data = window;
                menuItem.addEventListener(Event.SELECT,
                                          selectWindowHandler);

                _windowsMenu.addItem(menuItem);

                window.activate();
            }
        }

        private function selectWindowHandler(event:Event):void {
            event.target.data.activate();
        }

        private function activateHandler(event:Event):void {
            var item:NativeMenuItem;
            for(var i:Number = _windowsMenu.numItems - 1; i >= 0; i--) {
                item = _windowsMenu.getItemAt(i);
                item.checked = (item.data == event.target);
            }
        }

        private function closeHandler(event:Event):void {
            var item:NativeMenuItem;
            for(var i:Number = 0; i < _windowsMenu.numItems; i++) {
                item = _windowsMenu.getItemAt(i);
                if(item.data == event.target) {
                    _windowsMenu.removeItem(item);
                    break;
                }
            }
        }
    }

}
```

❶5 Открыть новое окно

❶6 Создать уникальный заголовок окна

❶7 Создать вспомогательное окно

❶8 Обработать события

❶9 Создать пункт меню окна

❷0 Добавить пункт в меню

❷1 Активировать нужное окно

❷2 Обновить галочку в меню

❷3 Удалить соответствующий пункт меню

Это длинный пример. Мы разберем его по частям. Сначала создадим новое меню ❶, поскольку это необходимо для всего последующего. На-

зовем это меню главным, потому что оно будет контейнером для остальных меню, которые мы будем добавлять. Затем создадим еще одно меню и два пункта ❷. Это будет подменю главного меню. Добавим к одному из пунктов слушателя событий ❸, чтобы иметь возможность ответить на выбор пользователем этого пункта меню. Мы создали пункты меню, но пока не добавили их в меню. Сделаем это с помощью метода `addItem()` ❹. В главном меню будет еще одно подменю, которое мы теперь и создадим ❺ вместе с вариантом выбора. Как и для пункта `close` меню приложения, назначим обработчик события выбора нового окна ❻. Меню окон является динамическим, поскольку во время работы приложения в него будет добавляться новый пункт каждый раз, когда пользователь откроет окно. Чтобы отделить в меню окон статические элементы от динамических, поместим в него разделитель. Для этого создадим новый пункт меню, указав в качестве второго параметра `true` ❼. Создав два подменю, можем добавить их в главное меню ❽. Создав меню, нужно применить его в качестве меню окна или меню приложения. Получим ссылку на главное окно ❾, узнаем, какого типа меню поддерживаются в системе, и соответственно этому присвоим меню ❿. Кроме того, будем ловить событие закрытия, чтобы закрыть все открытые окна, когда закроется главное окно ⓫.

Метод `closeAll()` ⓬ действует обычно одинаково. Он обходит все открытые окна и для каждого вызывает метод `close()`. Единственная особенность этого примера в том, что помимо вызова метода `close()` в методе `closeAll()` также удаляются из каждого окна подписки на событие закрытия ⓭. Это важное обстоятельство, потому что в этом примере, как будет видно, при создании новых окон мы подписываем их на событие закрытия. Если не удалить эту подписку перед закрытием окна, мы войдем в бесконечный цикл.

Метод `selectHandler()` — это обработчик событий выбора пункта меню. Сначала определяем, какой пункт выбрал пользователь. Если он выбрал пункт `close` ⓮, вызываем метод `closeAll()`. Иначе, если он выбрал пункт открытия нового окна ⓯, выполняем код, открывающий новое окно. Чтобы создать новое окно, мы сначала создаем для него уникальный заголовок ⓰, пользуясь счетчиком существующих окон, и генерируем такие имена, как Window 1, Window 2 и т. д. Затем создаем новое вспомогательное окно размером 200 на 200 пикселов ⓱. Мы должны выполнить определенные действия при закрытии или активации пользователем окна, поэтому регистрируем обработчики соответствующих событий ⓲. У каждого окна в меню окна должен быть пункт меню. Добавляем пункт меню ⓳, используя в качестве его имени заголовок окна, и сохраняем в его свойстве `data` ссылку на окно. Ссылка нужна, чтобы активировать окно, выбранное в меню. Затем добавляем новый пункт в меню окна ⓴.

Осталось три события, которые нужно обработать: выбор пункта меню, соответствующего окну, активация окна и закрытие окна. При выборе пункта меню активация окна происходит на основе данных,

содержащихся в свойстве data пункта меню ❷❶. При активации окна ❷❷ происходит обход всех пунктов меню и обновление для каждого свойства «галочка», с тем чтобы пометить только пункт меню активированного окна. При закрытии окна ❷❸ нужно обойти все пункты меню и удалить пункт, соответствующий окну.

Тестируя это приложение, вы можете добавлять новые окна, выбирая соответствующий пункт в меню окна. При добавлении новых окон в меню окон появляются новые пункты. Выбор окна изменяет пометку выбранного окна в меню. При закрытии окон удаляются соответствующие пункты меню.

Меню иконок

Меню иконок – это меню, которые открываются в области уведомлений панели задач (Windows) или в доке (OS X) и содержат пиктограммы приложений. Настроить меню иконок можно с помощью свойства icon.menu объекта NativeApplication.

```
NativeApplication.nativeApplication.icon.menu = customMenu;
```

Это все, что требуется для настройки меню иконок.

Меню окна, приложения и значка появляются вне контента вашего приложения. Теперь мы посмотрим, как пользоваться меню, возникающими внутри контента приложения.

Контекстные меню

Контекстные меню можно показывать для любых отображаемых объектов в приложениях AIR на базе Flash или Flex. Механизм добавления контекстных меню к отображаемым объектам в приложениях AIR точно такой же, как в веб-приложениях. Нужно присвоить меню свойству contextMenu отображаемого объекта. Это стандартный прием Flash и Flex, используемый и при разработке веб-приложений. Однако для приложений AIR есть два важных отличия:

- Контекстные меню в приложениях AIR являются экземплярами NativeMenu, тогда как в веб-приложениях они являются экземплярами ContextMenu.
- Контекстные меню в приложениях AIR относятся к системному уровню, а потому могут находиться впереди и за границами окна приложения.

Простое контекстное меню приведено в листинге 2.28.

Листинг 2.28. Добавление контекстного меню

```
package {

   import flash.display.MovieClip;
   import flash.display.NativeMenu;
   import flash.display.NativeMenuItem;
   import flash.display.Sprite;
```

```
    public class Example extends MovieClip {

        public function Example() {

            var rectangle:Sprite = new Sprite();
            rectangle.graphics.lineStyle(0, 0, 1);
            rectangle.graphics.beginFill(0, 1);
            rectangle.graphics.drawRect(0, 0, 100, 100);
            rectangle.graphics.endFill();
            addChild(rectangle);

            var menu:NativeMenu = new NativeMenu();
            var item:NativeMenuItem = new NativeMenuItem("copy");
            menu.addItem(item);

            rectangle.contextMenu = menu;
        }
    }
}
```

Рисунок 2.9 иллюстрирует работу этого кода.

Есть еще один способ применения меню в приложениях AIR – всплывающие меню, которые мы теперь и рассмотрим.

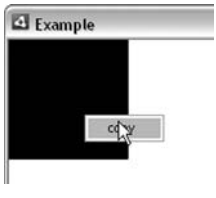


***Рис. 2.9.** Контекстные меню AIR могут появляться вне границ окна приложения*

Всплывающие меню

Всплывающие меню можно в любой момент показывать программным способом. Часто требуется открыть всплывающее меню, когда пользователь щелкает мышью или нажимает определенную клавишу. Каким бы образом вы ни инициировали открытие меню, код для его показа остается одним и тем же. Нужно просто вызвать метод `display()` объекта `NativeMenu`. Метод `display()` требует, чтобы ему передали три параметра: объект `Stage`, на котором нужно показать меню, и координаты `x` и `y`, по которым его нужно поместить. Простое всплывающее меню приведено в листинге 2.29.

Листинг 2.29. Добавление всплывающего меню

```
package {

    import flash.display.MovieClip;
    import flash.display.NativeMenu;
    import flash.display.NativeMenuItem;
```

```
import flash.display.Sprite;
import flash.events.MouseEvent;

public class Example extends MovieClip {

   private var _menu:NativeMenu;

   public function Example() {

      _menu = new NativeMenu();
      var item:NativeMenuItem = new NativeMenuItem("a");
      _menu.addItem(item);
      item = new NativeMenuItem("b");
      _menu.addItem(item);
      item = new NativeMenuItem("c");
      _menu.addItem(item);
      item = new NativeMenuItem("d");
      _menu.addItem(item);

      var rectangle:Sprite = new Sprite();
      rectangle.graphics.lineStyle(0, 0, 1);
      rectangle.graphics.beginFill(0, 1);
      rectangle.graphics.drawRect(0, 0, 100, 100);
      rectangle.graphics.endFill();
      addChild(rectangle);

      rectangle.addEventListener(MouseEvent.MOUSE_DOWN, showMenu);

   }

   private function showMenu(event:MouseEvent):void {
      _menu.display(stage, mouseX, mouseY);
   }
}

}
```

❶ Создать новое меню

❷ Создать прямоугольник

❸ Создать событие mouseDown

❹ Показать всплывающее меню

В этом примере мы сначала создаем новое меню и добавляем в него четыре пункта ❶. Затем мы создаем отображаемый объект ❷, с помощью которого станем запускать всплывающее меню. Мы запустим всплывающее меню, когда пользователь щелкнет по созданному прямоугольнику. Для этого мы зарегистрируем обработчик события `mouseDown` ❸. Когда пользователь щелкнет по прямоугольнику, мы выведем всплывающее меню в той точке, где щелкнул пользователь ❹.

Теперь вам известны все способы применения меню в приложениях AIR: меню окна, меню приложения, меню иконок, контекстные меню и всплывающие меню. На этом завершается не только обсуждение меню, но и весь теоретический материал данной главы. Теперь мы воспользуемся полученными знаниями для создания простого приложения.

2.5. Начинаем разработку приложения AirTube

На протяжении всей книги мы будем прибегать к разнообразным маленьким учебным приложениям. Однако у нас будет одно большое

центральное приложение, к которому мы будем возвращаться по мере изучения новых тем. Это приложение называется AirTube, потому что в нем с помощью Adobe AIR организован доступ к популярному видеосервису YouTube. В этой главе мы разработаем основу всего приложения, создав окна, которые будет использовать приложение. Кроме того, мы создадим некоторые базовые классы и компоненты, используемые приложением.

2.5.1. Обзор AirTube

В AirTube используется открытый API разработчика YouTube для построения приложения AIR, которое позволяет осуществлять поиск по ключевым словам в каталоге YouTube, воспроизводить видео и сохранять видео для воспроизведения в автономном режиме. AirTube имеет следующие важные характеристики:

- Настольное приложение с несколькими окнами
- Поиск в видео YouTube по ключевым словам
- Воспроизведение видео
- Просмотр страниц YouTube с видео
- Сохранение фильмов на локальной машине
- Поиск и воспроизведение локальных фильмов
- Определение доступности сети
- Сохранение на рабочем столе ярлыков для фильмов.

Прежде чем вникать в детали устройства приложений, посмотрим два снимка экрана готового приложения. На рис. 2.10 приведен главный экран. Этот экран дает пользователям возможность искать фильмы по ключевым словам/тегам. Список результатов выводится в две колонки.

Теперь взглянем на окно видео. Рисунок 2.11 демонстрирует это окно, которое не только воспроизводит выбранный фильм, но дает возможность перейти на страницу YouTube с фильмом или загрузить фильм для автономного просмотра.

В следующих разделах мы разберем первые шаги для построения этого приложения.

2.5.2. Начало

Прежде чем писать код, нужно выполнить некоторые предварительные действия:

- Подписаться на ключ API разработчика YouTube
- Загрузить две библиотеки ActionScript
- Настроить новый проект AIR

Это приложение зависит от сервиса YouTube. (См. *www.youtube.com/dev*.) Это бесплатный открытый сервис, с помощью которого разработ-

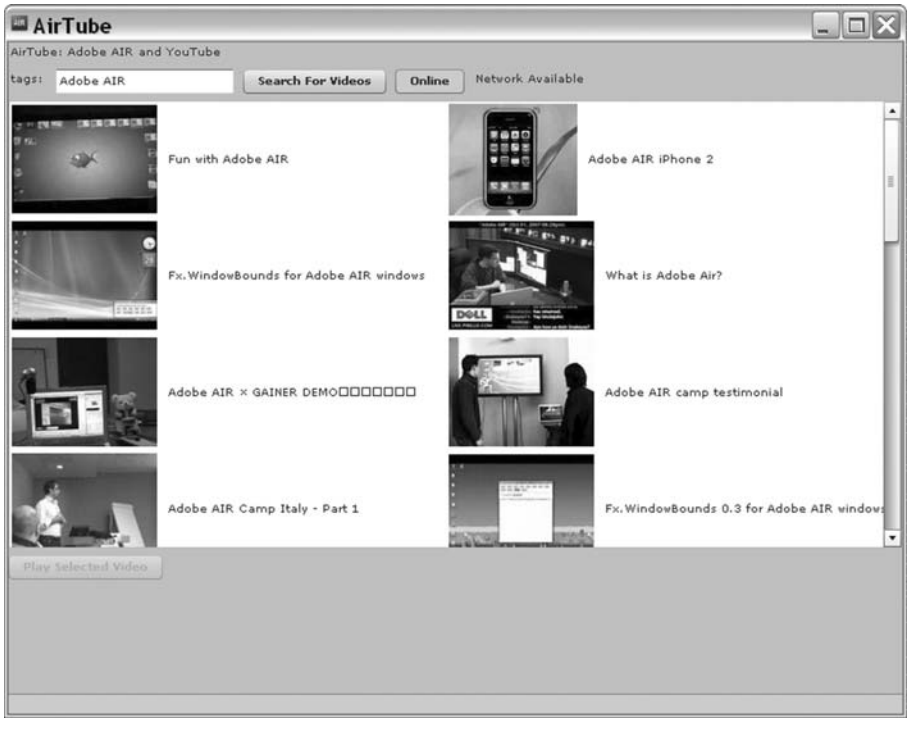


Рис. 2.10. *Главный экран приложения AirTube позволяет осуществлять поиск и перемещаться по каталогу фильмов*

чики могут строить приложения, выполняющие поиск в видеотеке YouTube, и воспроизводить фильмы. Сервис бесплатный, но для доступа к нему нужна регистрация.

1. Если у вас еще нет регистрации, можете открыть бесплатную учетную запись YouTube на *www.youtube.com/signup*. Просто заполните форму и щелкните по кнопке Sign Up.
2. После создания учетной записи и регистрации в системе нужно перейти на свою страницу по адресу *www.youtube.com/мой_аккаунт*.
3. На своей странице выберите опцию Developer Profile или сразу перейдите на *www.youtube.com/my_profile_dev*.
4. Попросите новый ID разработчика. Это ключ, который потребуется вам для доступа к сервису YouTube.

После того как у вас появились необходимые учетная запись и ключ YouTube, надо обзавестись соответствующими библиотеками ActionScript. Конечно, можно написать на ActionScript собственный код, чтобы непосредственно работать с YouTube, но проще воспользоваться имеющимися библиотеками, специально созданными для этой цели. Нужно загрузить библиотеку as3youtubelib, а также as3corelib. Офи-

***Рис. 2.11.** Экран видео позволяет воспроизводить фильм, переходить к страницам YouTube и сохранять фильмы локально*

циальными сайтами для этих библиотек являются code.google.com/p/as3youtubelib и code.google.com/p/as3corelib, соответственно. Чтобы гарантированно работать с теми версиями библиотек, которые использованы в данной книге, можно загрузить эти библиотеки с официального сайта книги на *www.manning.com/lott*.

Следующий шаг – создание нового проекта AIR для приложения AirTube. В этом проекте нет ничего особенного, поэтому можете создать его как обычно – с помощью Flex Builder или вручную. Создав проект, добавьте в него библиотеки as3youtubelib и as3corelib. Если вы загрузили их с сайта книги, то вместе с ними получили и исходный код, и для работы удобнее всего распаковать код в каталог с исходным кодом проекта.

Это все, что требуется для настройки проекта. Теперь можно начать строить приложение.

2.5.3. Создание модели данных

Приложение AirTube строится вокруг локатора модели данных, который мы назовем `ApplicationData`. Класс `ApplicationData` основан на шаблоне проектирования Singleton («одиночка»), который гарантирует наличие во всем приложении только одного глобально доступного экземпляра класса. С помощью экземпляра `ApplicationData` мы организуем хранение всей информации о состоянии приложения: подключено приложение к сети или работает автономно, есть ли какие-то результаты запроса видео, находится ли видео в состоянии загрузки и т. д. Все эти данные записываются в `ApplicationData`.

Мы построим ApplicationData (или, по крайней мере, приступим к строительству) в самом скором времени. Но сначала мы должны создать другой класс, который войдет в модель данных приложения. Поскольку AirTube по существу представляет собой приложение для поиска и просмотра видео, для нашей модели мы построим класс видео, который назовем AirTubeVideo. Класс AirTubeVideo служит оболочкой для класса Video из библиотеки as3youtube. Этот класс Video служит примером сериализации любых данных, возвращаемых вызовами API, и содержит такую информацию, как название видео, ID, миниатюрное изображение и т. д. Класс оболочки (AirTubeVideo) для моделирования видеоданных нужен нам для того, чтобы помимо данных, возвращаемых YouTube API, сохранить для каждого фильма дополнительную информацию, такую как URL файла .flv и отметку о записи данного фильма на локальный диск. Для создания модели данных приложения AirTube выполните следующие шаги:

1. Создайте новый документ класса ActionScript и сохраните его в com/manning/airtube/data/AirTubeVideo.as относительно каталога исходных файлов вашего проекта.
2. Добавьте в класс AirTubeVideo код листинга 2.30. Как можно видеть, AirTubeVideo требует при создании нового экземпляра объект Video в качестве параметра. Класс предоставляет метод доступа для объекта Video и еще два вида данных: URL файла .flv и состояние офлайн для видео (был ли фильм загружен локально).

Листинг 2.30. Класс AirTubeVideo

```
package com.manning.airtube.data {

   import com.adobe.webapis.youtube.Video;

   import flash.events.Event;
   import flash.events.EventDispatcher;

   public class AirTubeVideo extends EventDispatcher {

      private var _video:Video;
      private var _flvUrl:String;
      private var _offline:Boolean;

      public function get video():Video {
         return _video;
      }

      [Bindable(event="flvUrlChanged")]
      public function get flvUrl():String {
         return _flvUrl;
      }

      public function set flvUrl(value:String):void {
         _flvUrl = value;
         dispatchEvent(new Event("flvUrlChanged"));
      }
```

❶ Включить связывание данных Flex

```
        [Bindable(event="offlineChanged")]
        public function set offline(value:Boolean):void {
            _offline = value;
            dispatchEvent(new Event("offlineChanged"));
        }

        public function get offline():Boolean {
            return _offline;
        }

        public function AirTubeVideo(value:Video) {
            _video = value;
        }

    }
}
```

В коде есть метатег `[Bindable]` ❶, используемый Flex для связывания данных. Всюду в этой книге мы придерживаемся единого правила для именования событий и тегов метаданных `[Bindable]`: имя события – это имя метода установки/чтения плюс `Changed`. В данном примере методы называются `flvUrl`, а событие, соответственно, `flvUrlChanged`.

3. Создайте новый документ класса ActionScript и сохраните его в com/manning/airtube/data/ApplicationData.as относительно каталога исходных файлов вашего проекта.
4. Добавьте в класс `ApplicationData` код из листинга 2.31. Класс `ApplicationData` построен по шаблону Singleton, поэтому у него есть статическое свойствво _instance и метод доступа (`getInstance()`) для получения одного экземпляра класса. Помимо этого в `ApplicationData` есть два вида данных: массив видео и ссылка на видео, выбранное в данный момент.

Листинг 2.31. Класс ApplicationData

```
package com.manning.airtube.data {

    import flash.events.Event;
    import flash.events.EventDispatcher;

    public class ApplicationData extends EventDispatcher {

        static private var _instance:ApplicationData;

        private var _videos:Array;
        private var _currentVideo:AirTubeVideo;

        [Bindable(event="videosChanged")]
        public function set videos(value:Array):void {
            _videos = value;
            dispatchEvent(new Event("videosChanged"));
        }

        public function get videos():Array {
```

❶ Управляемый экземпляр класса

```
            return _videos;
        }

        [Bindable(event="currentVideoChanged")]
        public function set currentVideo(value:AirTubeVideo):void {
            _currentVideo = value;
            dispatchEvent(new Event("currentVideoChanged"));
        }

        public function get currentVideo():AirTubeVideo {
            return _currentVideo;
        }

        public function ApplicationData() {
        }

        static public function getInstance():ApplicationData {
            if(_instance == null) {
                _instance = new ApplicationData();
            }
            return _instance;
        }

    }
}
```

Доступ к управляемому экземпляру ❷

Если вы не знакомы с шаблоном Singleton, то вам необходимо в этом коде обратить внимание на два главных момента. Во-первых, имеется статическое свойство того же типа, что и сам класс ❶. Это свойство используется для хранения единственного экземпляра класса. Кроме того, имеется открытый статический метод `getInstance()` ❷, который возвращает ссылку на этот единственный экземпляр класса. По такому же шаблону сконструировано еще несколько классов приложения AirTube.

Это все, что требуется для построения базовой модели данных приложения AirTube. Теперь мы начнем строить сервис для работы с AirTube.

2.5.4. Разработка сервиса AirTube

Приложение AirTube в основном занято применением API разработчика YouTube. Мы напишем класс `AirTubeService`, который действует в качестве прокси для API YouTube. Со временем мы добавим в класс сервиса дополнительные функции. Следующие шаги приведут нас к начальному варианту класса сервиса:

1. Откройте новый документ класса `ActionScript` и сохраните его в com/manning/airtube/services/AirTubeService.as относительно каталога исходных файлов вашего проекта.
2. Добавьте в класс `AirTubeService` код из листинга 2.32. Как видите, `AirTubeService` также использует шаблон Singleton.

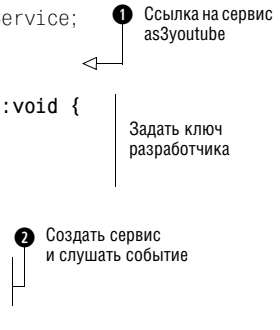
Листинг 2.32. Класс AirTubeService

```
package com.manning.airtube.services {

   import  com.adobe.webapis.youtube.YouTubeService;
   import  com.adobe.webapis.youtube.events.YouTubeServiceEvent;
   import  com.manning.airtube.data.AirTubeVideo;
   import  com.manning.airtube.data.ApplicationData;

   import flash.events.Event;

   public class AirTubeService {

      static private var _instance:AirTubeService;

      public function AirTubeService() {
      }

      static public function getInstance():AirTubeService {
         if(_instance == null) {
             _instance = new AirTubeService();
         }
         return _instance;
      }
   }
}
```

3. Добавьте свойство _proxied, содержащее ссылку на экземпляр класса YouTubeService из библиотеки as3youtube. Этот экземпляр позволит AirTubeService вызывать методы YouTubeService. Листинг 2.33 показывает AirTubeService с выделенными жирным шрифтом модификациями.

Листинг 2.33. Класс AirTubeService с запросом сервиса через прокси

```
package com.manning.airtube.services {

   import  com.adobe.webapis.youtube.YouTubeService;
   import  com.adobe.webapis.youtube.events.YouTubeServiceEvent;
   import  com.manning.airtube.data.AirTubeVideo;
   import  com.manning.airtube.data.ApplicationData;

   import flash.events.Event;

   public class AirTubeService {

      static private var _instance:AirTubeService;

      private var _proxied:YouTubeService;

      public function set key(value:String):void {

        _proxied.apiKey = value;
      }

      public function AirTubeService() {

         _proxied = new YouTubeService();
         _proxied.addEventListener(
```



```
➥YouTubeServiceEvent.VIDEOS_LIST_BY_TAG, getVideosByTagsResultHandler);
        }

    static public function getInstance():AirTubeService {
            if(_instance == null) {
                _instance = new AirTubeService();
            }
            return _instance;
        }

        public function getVideosByTags(tags:String):void {
            if(_proxied.apiKey.length == 0) {
                throw Error("YouTube API key not set");
            }
            _proxied.videos.listByTag(tags);
        }

        private function getVideosByTagsResultHandler(
        ➥event:YouTubeServiceEvent):void    {
            var videos:Array = event.data.videoList as Array;
            for(var i:Number = 0; i < videos.length; i++) {
                videos[i] = new AirTubeVideo(videos[i]);
            }
            ApplicationData.getInstance().videos = videos;
        }
    }
}
```

❸ Искать видео

Обработчик события завершения поиска

❹ Просмотреть результаты

Обновить модель данных ❺

Свойством `_proxied` ❶ мы будем пользоваться для хранения ссылки на экземпляр класса `YouTubeService` из библиотеки as3youtube. С помощью этого экземпляра будет выполняться обращение к сервису YouTube. Экземпляр этого сервиса создает конструктор ❷.

Сервис YouTube требует ключа разработчика, о котором говорилось в начальном разделе. Класс `YouTubeService` требует передать этот ключ через свойство `apiKey`. Мы создаем в `AirTubeService` метод, с помощью которого можно передать ключ экземпляру `YouTubeService`.

Метод `getVideosByTags()` ❸ выдает запрос на получение фильмов, содержащих один и более заданных тегов/ключевых слов. Этот метод просто вызывает метод `listByTag()`, предоставляемый сервисом YouTube через экземпляр `_proxied`. Помимо пересылки запроса мы лишь проверяем, что задан ключ разработчика. Если ключ не задан, сервис не будет работать. Поэтому при отсутствии ключа генерируется ошибка.

Метод `getVideosByTagsResultHandler()` обрабатывает событие возврата данных методом `listByTag()` класса `YouTubeService`. Полученные результаты преобразуются к удобному виду и присваиваются экземпляру `ApplicationData`. Данные возвращаются в виде массива объектов `Video` (из библиотеки as3youtube). Массив хранится в свойстве `data.videoList` объекта события. Мы хотим перебрать все результаты и заключить их в объекты `AirTubeVideo` ❹. После того как фильмы

примут вид объектов AirTubeVideo, можно записать их массив в свойство videos объекта ApplicationData ❺. Это вынудит ApplicationData сгенерировать событие, оповещающее слушателей событий о том, что они должны обновить себя в соответствии с новыми данными.

Теперь у нас есть обслуживающий класс для приложения AirTube. Далее мы создадим для приложения окна и соединим все вместе.

2.5.5. Получение URL для .flv

На момент написания данной книги YouTube API не возвращает прямых URL файлов .flv для видео. Вместо этого при запросе видео с YouTube возвращается URL на видеоплеер, написанный на Flash, который и открывает файлы .flv. Чтобы загрузить фильм и сохранить его локально, приложение AirTube должно получить прямой URL на файл .flv этого фильма. Достижение такого результата требует некоторой изобретательности. В этом разделе мы построим класс, который извлекает действительный URL файла .flv для данного фильма по URL плеера этого фильма на YouTube.

Примечание

Используемый нами способ получения URL по файлу .flv на YouTube является, грубо говоря, хакерским. Поэтому нельзя гарантировать, что YouTube и впредь будет поддерживать такой метод доступа к файлам. Если система доступа изменится, то мы постараемся найти новое практическое решение и опубликовать его на сайте этой книги.

В данный момент для URL, по которому можно получить файл .flv с YouTube, нужны два параметра, которые YouTube называет video_id и t. Их можно получить, сделав запрос по URL плеера, возвращенному сервисом YouTube, и прочтя значения video_id и t в URL, на который переадресует URL плеера. Для ясности приведем пример. Вот некий URL плеера, возвращенный сервисом YouTube:

```
http://www.youtube.com/v/llRw9UG48Dw
```

Если посмотреть его в веб-броузере, то можно заметить, что происходит переадресация на следующий URL:

```
http://www.youtube.com/swf/l.swf?video_id=llRw9UG48Dw&rel=1&eurl=&i
➥url=http%3A//i.ytimg.com/vi/llRw9UG48Dw/default.jpg&
➥t=OEgsToPDskJtLebBhzjJbUnpN-uo9iSI
```

Мы выделили здесь значения video_id и t полужирным шрифтом. Эти значения мы хотим извлечь из URL. С их помощью мы сможем получить файл .flv по приведенному далее URL, в котором выделенный курсивом текст нужно заменить полученными нами значениями video_id и t:

```
http://www.youtube.com/get_video.php?video_id=video_id&t=t
```

Для решения этой задачи мы напишем вспомогательный класс `YouTubeFlvUrlRetriever` и добавим некий метод в класс `AirTubeService`. Выполните следующие действия:

1. Откройте новый документ класса ActionScript и сохраните его в com/manning/airtube/utilities/YouTubeFlvUrlRetriever.as относительно каталога исходных файлов вашего проекта.
2. Добавьте в код класса `YouTubeFlvUrlRetriever` код из листинга 2.34.

Листинг 2.34. Класс YouTubeFlvUrlRetriever

```
package com.manning.airtube.utilities {
   import com.manning.airtube.data.AirTubeVideo;
   import flash.display.Loader;
   import flash.events.Event;
   import flash.net.URLRequest;

   import flash.net.URLVariables;

   public class YouTubeFlvUrlRetriever {

      private var _currentVideo:AirTubeVideo;
      private var _loader:Loader;

      public function YouTubeFlvUrlRetriever() {
         _loader = new Loader();
      }

      public function getUrl(video:AirTubeVideo):void {
         _currentVideo = video;
         var request:URLRequest = new
            URLRequest(video.video.playerURL);
         _loader.contentLoaderInfo.addEventListener(Event.INIT,
         ➥videoInitializeHandler);
         _loader.load(request);
      }

      private function videoInitializeHandler(event:Event):void {
         var variables:URLVariables = new URLVariables();

         variables.decode(
         ➥_loader.contentLoaderInfo.url.split("?")[1]);
         var flvUrl:String = "http://www.youtube.com/get_video.php?" +
            "video_id=" + variables.video_id + "&t=" + variables.t;
         _currentVideo.flvUrl = flvUrl;
         _loader.unload();
      }

   }
}
```

❶ Создать загрузчик

❷ Сохранить текущий фильм

❸ Составить запрос на URL плеера

❹ Слушать событие init

❺ Декодировать переменные URL

❻ Составить URL для .flv

❼ Сохранить URL

❽ Прервать загрузку плеера

В этом коде мы прежде всего строим новый объект `Loader` ❶. С помощью объекта `Loader` мы делаем запрос HTTP по URL плеера YouTube и извлекаем переменные `video_id` и `t`.

Когда мы готовы запросить URL, то сначала запоминаем ссылку на фильм ❷, чтобы обновить его свойство `flvUrl` после того, как получим URL. Затем можно создать запрос по URL YouTube-плеера фильма ❸, а потом загрузить этот URL ❹. В результате будет выполнен запрос и после инициализации получен URL переадресации.

Когда приложение получит сообщение о начале загрузки (событие.init), в свойстве `contentLoaderInfo.url` объекта `Loader` будет находиться URL переадресации, содержащий переменные `video_id` и `t`. Мы разрежем URL в районе вопросительного знака, чтобы оставить только строку запроса, а потом пропустим ее через `decode()` ❺, чтобы объект `URLVariables` выделил нужные переменные. С помощью декодированных переменных мы сможем построить URL файла .flv ❻. Составив правильный URL, мы сможем обновить свойство `flvUrl` текущего видео ❼. Мы должны также выгрузить объект Loader ❽, чтобы прекратить загрузку плеера YouTube, потому что запрос был нужен нам для получения переменных, а не самого плеера.

3. Откройте `AirTubeService` и добавьте в него метод из листинга 2.35. Этот метод устанавливает свойство `currentVideo` объекта `ApplicationData`, чтобы помнить, какой фильм выбрал пользователь, а потом проверяет, не равно ли свойство фильма `flvUrl` значению null (переменная не определена). Значение null означает, что `YouTubeFlvUrlRetriever` еще не настроил фильм. Если свойство не определено, получаем его с помощью `YouTubeFlvUrlRetriever`.

Листинг 2.35. Метод configureVideoForPlayback( ) method

```
public function configureVideoForPlayback(video:AirTubeVideo):void {
   ApplicationData.getInstance().currentVideo = video;
   if(video.flvUrl == null) {
      new YouTubeFlvUrlRetriever().getUrl(video);
   }
}
```

Мы смогли написать код для получения URL файла .flv. Теперь построим окна, которые дадут пользователю возможность искать фильмы, видеть результаты поиска и даже воспроизводить фильмы.

2.5.6. Создание главного окна AirTube

Главное окно приложения AirTube (показанное на рис. 2.10) состоит из панели управления поиском, панели со списком результатов поиска и кнопки для запуска воспроизведения выбранного фильма. В этом разделе мы построим главное окно и организуем с его помощью отправку запросов сервису YouTube. Выполните следующие действия:

1. Создайте новый документ MXML и сохраните его под именем AirTube.mxml в каталоге исходных файлов проекта AirTube.
2. Добавьте в документ код из листинга 2.36. Обратите внимание, что AirTube.mxml тоже спроектирован согласно шаблону Singleton.

Листинг 2.36. AirTube.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" width="800" height="600"
 creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import com.manning.airtube.services.AirTubeService;
         import com.manning.airtube.data.ApplicationData;

         static private var _instance:AirTube;

         private var _service:AirTubeService;

         static public function getInstance():AirTube {
            return _instance;
         }
         private function creationCompleteHandler():void {
            _service = AirTubeService.getInstance();
            _service.key = "YourAPIKey";
            _instance = this;
         }
      ]]>
   </mx:Script>
   <mx:VBox width="100%">
      <mx:Label text="AirTube: Adobe AIR and YouTube" />
      <mx:HBox>
         <mx:Label text="tags:" />
         <mx:TextInput id="tags" text="Adobe AIR" />
         <mx:Button label="Search For Videos" />
      </mx:HBox>
         <mx:TileList id="videoList"
         dataProvider="{ApplicationData.getInstance().videos}"
         width="100%" height="400"
         columnCount="2" horizontalScrollPolicy="off" />
   </mx:VBox>
</mx:WindowedApplication>
```

❶ Без отложенной инициализации

❷ Запуск инициализации

❸ Получить сервис

Задать ключ разработчика

Задать экземпляр Singleton

Компонента для вывода результатов

В большинстве случаев применения шаблона Singleton метод `getInstance()` использует отложенную инициализацию, при которой экземпляр класса создается непосредственно перед использованием.

В нашем случае в этом нет необходимости. Мы можем просто вернуть ссылку на `_instance` ❶, не проверяя, задано ли свойство `_instance`. Так как AirTube.mxml является экземпляром `WindowedApplication` для приложения AirTube, мы знаем, что он всегда будет существовать, когда какой-то код попытается к нему обратиться.

Код AirTube.mxml устроен так, что вызывает метод `creationCompleteHandler()` ❷, когда происходит событие `creationComplete`. В метод `creationCompleteHandler()` мы помещаем код, который нужно выполнить, когда запускается приложение. В этом документе экземпляр

AirTubeService используется часто, поэтому мы просто сохраним на него ссылку ❸. Затем нам нужно задать ключ разработчика для использования с сервисом YouTube. Замените YourAPIKey на свой рабочий ключ API YouTube. Мы должны также присвоить экземпляр this свойству _instance. Это часть шаблона Singleton. Мы знаем, что всегда создается только один экземпляр AirTube.mxml и это происходит автоматически при запуске приложения.

Для отображения результатов поиска фильмов служит компонента TileList. Обратите внимание на привязку в качестве источника данных свойства videos экземпляра ApplicationData. При всяком изменении свойства videos список будет обновлен.

3. Добавьте код, осуществляющий запрос фильмов у сервиса. Для этого нужен обработчик событий кнопки поиска. Внесенные в код изменения показаны в листинге 2.37 жирным шрифтом.

Листинг 2.37. Добавление в AirTube.mxml функций поиска

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" width="800" height="600"
 creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import com.manning.airtube.services.AirTubeService;
         import com.manning.airtube.data.ApplicationData;

         static private var _instance:AirTube;

         private var _service:AirTubeService;

         static public function getInstance():AirTube {
            return _instance;
         }

         private function creationCompleteHandler():void {
            _service = AirTubeService.getInstance();
            _service.key = "YourAPIKey";
            _instance = this;
         }

         private function getVideosByTags():void {
            _service.getVideosByTags(tags.text);
         }
      ]]>
   </mx:Script>
   <mx:VBox width="100%">
      <mx:Label text="AirTube: Adobe AIR and YouTube" />
      <mx:HBox>
         <mx:Label text="tags:" />
         <mx:TextInput id="tags" text="Adobe AIR" />
         <mx:Button label="Search For Videos" click=
         ➥"getVideosByTags();" />
```

```
        </mx:HBox>
        <mx:TileList id="videoList"
           dataProvider="{ApplicationData.getInstance().videos}"
           width="100%" height="400"
           columnCount="2" horizontalScrollPolicy="off" />
    </mx:VBox>
</mx:WindowedApplication>
```

4. Теперь можно протестировать приложение и увидеть что-то вроде изображенного на рис. 2.12 (после выполнения поиска). Как можно видеть, результаты поиска выглядят как обычный текст, едва ли понятный или полезный для большинства пользователей. Мы изменим такое поведение, добавив в список специальный обработчик отображения (item renderer). Для этого откройте новый документ MXML и сохраните его как com/manning/airtube/ui/VideoTileRenderer.mxml относительно каталога исходных файлов вашего проекта.

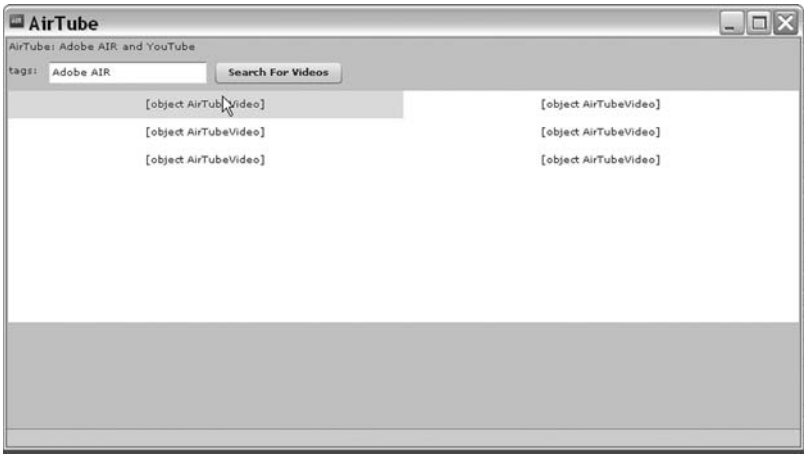


***Рис. 2.12.** Главное окно AirTube до подключения специальной функции вывода результатов поиска*

5. Добавьте в компоненту VideoTileRenderer код из листинга 2.38. Это простой код. В нем используется картинка и метка, чтобы показать миниатюрный кадр и название фильма, полученные функцией вывода из свойства `data`.

Листинг 2.38. VideoTileRenderer.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:HBox xmlns:mx="http://www.adobe.com/2006/mxml" width="200"
➥height="100"
 verticalScrollPolicy="off" horizontalScrollPolicy="off">
   <mx:Image source="{data.video.thumbnailUrl}" />
   <mx:Label text="{data.video.title}" />
</mx:HBox>
```

6. Обновите AirTube.mxml, чтобы использовать правильную визуализацию элементов списка. Для этого достаточно лишь в одной строке кода добавить атрибут `itemRenderer` к тегу `TileList`, как показано в листинге 2.39.

Листинг 2.39. Получение текущей визуализации элементов списка

```
<mx:TileList id="videoList"
        dataProvider="{ApplicationData.getInstance().videos}"
        width="100%" height="400"
        columnCount="2" horizontalScrollPolicy="off"
        itemRenderer="com.manning.airtube.ui.VideoTileRenderer" />
```

Теперь, создав главное окно, мы добавим еще два: видео и HTML.

2.5.7. Добавление окон видео и HTML

Помимо главного, у приложения AirTube есть еще два окна. Одно служит для воспроизведения фильмов, а другое – для просмотра страниц HTML. (В этом случае мы даем пользователю возможность просмотреть веб-страницу YouTube с фильмом.) Чтобы добавить в приложение эти окна, выполните следующие действия:

1. Создайте новый документ MXML и сохраните его как com/manning/airtube/windows/VideoWindow.mxml относительно каталога исходных файлов вашего проекта.
2. Добавьте к компоненте `VideoWindow` код из листинга 2.40.

Листинг 2.40. VideoWindow.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml"
   width="400" height="400"
   type="utility"
   closing="closingHandler(event);"
  creationComplete="creationCompleteHandler();">

   <mx:Script>
      <![CDATA[
         import com.manning.airtube.services.AirTubeService;
         import com.manning.airtube.data.ApplicationData;

         [Bindable]
         private var _applicationData:ApplicationData;

         private function creationCompleteHandler():void {
            _applicationData = ApplicationData.getInstance();
         }

         private function closingHandler(event:Event):void {
            event.preventDefault();
            visible = false;
         }
```

Задание ширины и высоты

Задание типа «вспомогательное»

Обработка события закрытия

Обработка события creationComplete

❶ Не дать окну закрыться

```
            private function togglePlayback():void {
               if(videoDisplay.playing) {
                  videoDisplay.pause();
                  playPauseButton.label = "Play";
               }
               else {
                  videoDisplay.play();
                  playPauseButton.label = "Pause";
               }
            }

         ]]>
      </mx:Script>
      <mx:VBox>
         <mx:Label text=
         ➥"{_applicationData.currentVideo.video.title}" />
         <mx:VideoDisplay id="videoDisplay"
              source="{_applicationData.currentVideo.flvUrl}"
              width="400" height="300" />
         <mx:HBox>
            <mx:Button id="playPauseButton" label="Pause"
                   click="togglePlayback();" />
         </mx:HBox>
      </mx:VBox>
   </mx:Window>
```

Показать название фильма ❷

Подключить показ фильма ❸

Обратите внимание, что мы перехватываем событие закрытия, чтобы не дать выполниться действию по умолчанию ❶, а вместо этого сделать окно невидимым. Кроме того, для компонент метки и видеопоказа задана привязка источника данных к свойствам текущего фильма ❷ ❸.

3. Создайте новый документ MXML и сохраните его как com/manning/airtube/windows/HTMLWindow.mxml относительно каталога исходных файлов вашего проекта.
4. Добавьте в HTMLWindow.mxml код из листинга 2.41. В данный момент это окно пусто. Добавлять в него содержимое мы будем в главе 7. Эта компонента обрабатывает событие закрытия точно так же, как VideoWindow.

Листинг 2.41. HTMLWindow.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
   width="800" height="800" closing="closingHandler(event);">
   <mx:Script>
      <![CDATA[

         private function closingHandler(event:Event):void {
            event.preventDefault();
            visible = false;
         }
```

```
        ]]>
    </mx:Script>
</mx:Window>
```

5. Измените AirTube.mxml, чтобы поместить туда ссылки на экземпляры двух окон, которые мы только что создали, и дать пользователю возможность смотреть выбранный фильм в новом окне. Добавьте туда также код, который закрывает все окна при завершении работы приложения. Соответствующие изменения выделены на листинге 2.42 жирным шрифтом.

Листинг 2.42. Добавление в AirTube.mxml дополнительных окон

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 layout="absolute" width="800" height="600"
 creationComplete="creationCompleteHandler();"
 closing="closingHandler();">
   <mx:Script>
      <![CDATA[
         import com.manning.airtube.services.AirTubeService;
         import com.manning.airtube.data.ApplicationData;

         static private var _instance:AirTube;

         private var _service:AirTubeService;
         private var _videoWindow:VideoWindow;
         private var _htmlWindow:HTMLWindow;
         static public function getInstance():AirTube {
            return _instance;
         }

         private function creationCompleteHandler():void {
            _service = AirTubeService.getInstance();
            _service.key = "YourAPIKey";
            _instance = this;
            _videoWindow = new VideoWindow();
            _htmlWindow = new HTMLWindow();
         }

         private function getVideosByTags():void {
            _service.getVideosByTags(tags.text);
         }

         private function playVideo():void {
            var video:AirTubeVideo =
            ➥videoList.selectedItem as AirTubeVideo;
            _service.configureVideoForPlayback(video);
            if(_videoWindow.nativeWindow == null) {
               _videoWindow.open();
            }
            else {
               _videoWindow.activate();
            }
```

❶ Создание экземпляров окон

❷ Получение выбранного фильма

❸ Получение URL для .flv

❹ Открыть окно просмотра

```
        }
        public function launchHTMLWindow(url:String):void {
          if(_htmlWindow.nativeWindow == null) {
              _htmlWindow.open();
          }
          else {
              _htmlWindow.activate();
          }
        }
        private function closingHandler():void {
              for(var i:Number = 0; i <
              ➥nativeApplication.openedWindows.length; i++) {
                 nativeApplication.openedWindows[i].close();
              }
         }
    ]]>
  </mx:Script>
  <mx:VBox width="100%">
    <mx:Label text="AirTube: Adobe AIR and YouTube" />
    <mx:HBox>
      <mx:Label text="tags:" />
      <mx:TextInput id="tags" text="Adobe AIR" />
      <mx:Button label="Search For Videos"
       click="getVideosByTags();" />
    </mx:HBox>
    <mx:TileList id="videoList"
      dataProvider="{ApplicationData.getInstance().videos}"
      width="100%" height="400"
      columnCount="2" horizontalScrollPolicy="off" />
    <mx:Button label="Play Selected Video" click="playVideo();"
      enabled="{videoList.selectedItem != null}" />
  </mx:VBox>
</mx:WindowedApplication>
```

❺ Открыть окно HTML

❻ Закрыть все окна

Кнопка воспроизведения фильма ❼

Приложение разрешает создать только одно окно просмотра и одно окно HTML. Мы используем одни и те же экземпляры этих окон для каждого фильма и каждой страницы HTML, которые открывает приложение, вначале создав их экземпляры ❶. Эти окна открываются в зависимости от действий пользователя.

Если пользователь хочет воспроизвести фильм, нужно сделать несколько операций. Сначала нужно получить фильм, выбранный из списка ❷. Затем нужно обработать фильм методом `configureVideoForPlayback()`, чтобы получить URL файла .flv ❸. После этого можно открыть экземпляр окна просмотра ❹.

У нас в коде есть метод, который пока нигде не вызывается. Мы определили метод `launchHTMLWindow()` ❺, к которому будем обращаться позднее. Он просто открывает окно HTML.

Как обычно в наших приложениях, нужно обеспечить закрытие всех окон, когда пользователь закрывает главное окно. Этой цели служит метод `closingHandler()` ❻.

Кроме того, мы добавили кнопку для воспроизведения выбранного фильма ❼. При щелчке по кнопке вызывается метод `playVideo()`. Кнопка активна только тогда, когда в списке выделен какой-то фильм.

На этом первая фаза разработки приложения AirTube завершена. В текущем состоянии пользователь может осуществлять поиск фильмов и воспроизводить их в отдельном окне. В следующих главах мы снабдим приложение дополнительными функциями.

2.6. Резюме

В этой главе вы изучили основы работы с приложениями, окнами и меню. Это большой объем информации, которая могла оказаться новой для вас, поскольку изложенные понятия чаще относятся к настольным приложениям, а не сетевым приложениям.

Сделайте остановку и перечитайте заново тот материал, который, как вы чувствуете, требует повторения. Вот перечень некоторых важных положений, которые необходимо было усвоить:

- Приложения и окна представляются программно с помощью объектов `NativeApplication` и `NativeWindow` в чистом ActionScript, и объектов `WindowedApplication` и `Window` во Flex.
- Создание окон, открытие окон и наполнение окон содержанием – это независимые действия, которые вы должны организовать.
- Возможно создание окон неправильной формы.
- Приложения дают возможность выполнять разнообразные функции уровня приложения, такие как определение бездействия пользователя и запуск приложения в полноэкранном режиме.
- AIR дает возможность создавать меню, поддерживаемые операционной системой, и применять их разными способами, в том числе меню окон, меню приложений, меню иконок, контекстные и всплывающие меню.

Если вы готовы, можете переходить к следующей главе, в которой вы научитесь работать с файловой системой и выполнять такие операции, как чтение и запись файлов.

В этой главе:

- Получение содержимого каталогов
- Перемещение и копирование файлов и каталогов
- Чтение содержимого файлов
- Запись в файлы

3

Работа с файловой системой

Глава просто переполнена информацией, поскольку в AIR есть масса средств для работы с файловой системой. Ваши возможности ограничиваются в основном вашим воображением. Нам придется осветить обширный материал, включая:

- Способы обращения к файлам и каталогам
- Получение содержимого каталогов
- Копирование и перемещение файлов и каталогов
- Удаление файлов и каталогов
- Чтение содержимого файлов
- Запись в файлы

Мы подробно разберем все эти темы, приведя множество примеров, которые помогут вам в обучении. Немного труда, и вы сможете управляться с файловой системой, как опытный профессионал.

Прежде чем обратиться к базовым методам работы с файловой системой, мы хотим обсудить важную тему синхронного и асинхронного программирования. Это важный аспект программирования для AIR, особенно существенный при работе с файловой системой.

3.1. Понятие синхронизации

Идеи синхронного и асинхронного программирования имеют большое значение при создании приложений для AIR. Они касаются не только работы с файловой системой, но область работы с файлами – первая из освещаемых в этой книге, для которой важны эти понятия, поэтому мы обсудим их здесь.

Читатель должен быть знаком с синхронным программированием, поскольку это наиболее распространенный и простой вид программирования. При синхронном выполнении кода все операторы выполняются последовательно, и каждый оператор завершается и возвращает результат, прежде чем начнется выполнение следующего оператора. Листинг 3.1 демонстрирует типичный синхронный код.

Листинг 3.1. Пример синхронного кода

```
var devoTracks:Array = new Array("Freedom Of Choice", "Planet Earth",
➥"Cold War", "Don't You Know");
devoTracks.push("Mr. B's Ballroom");
var trackCount:int = devoTracks.length;
for(var i:int = 0; i < trackCount; i++) {
  trace(devoTracks[i]);
}
```

В этом листинге каждый оператор должен завершить свое выполнение, прежде чем может начать выполняться следующая строка кода. Например, в первой строке кода создается массив с начальными значениями, и лишь после этого может выполниться следующий оператор, который добавляет в этот массив еще одну величину. Синхронное программирование настолько обычно и естественно, что именно его вы, скорее всего, имеете в виду, произнося слово «программирование». Хотя большинство операторов ActionScript являются по сути синхронными, тем не менее, есть много операторов, являющихся *асинхронными*.

Асинхронным операторам не обязательно завершать свою работу, чтобы мог выполниться следующий оператор. Такого рода программирование может показаться хаотичным, но оно оказывается оправданным в ситуациях, когда некоторая операция может потребовать длительного времени для своего завершения, потому что иначе у пользователя возникает ощущение, будто приложение остановилось или зависло. Возьмем такой распространенный случай: во время выполнения приложения нужно обработать данные XML, загружаемые из внешнего XML-файла. Эта задача легко решается с помощью класса `flash.net.URLLoader`. Достаточно создать объект `URLLoader` и вызвать метод `load()`, сообщив ему, где найти файл XML. Однако загрузка данных может оказаться долгой – в зависимости от их объема и характеристик сетевого соединения. Если выполнять метод `load()` объекта `URLLoader` синхронно, то приложение может зависнуть вплоть до того момента, когда данные XML не будут загружены полностью. С учетом такой возможности метод `load()` спроектирован для асинхронного выполнения. Это означает, что приложению не нужно ждать, пока `load()` полностью загрузит данные, чтобы начать выполнение следующего оператора. Данная идея продемонстрирована в листинге 3.2.

Листинг 3.2. Асинхронное выполнение операторов

```
private var _loader:URLLoader;

private function startXmlRequest():void {
```

```
        _loader = new URLLoader();
        _loader.addEventListener(Event.COMPLETE, loaderCompleteHandler);
        _loader.load(new URLRequest("data.xml"));
        trace(_loader.data);
    }

    private function loaderCompleteHandler(event:Event):void {
        trace(_loader.data);
    }
```

Мы видим в этом листинге два оператора `trace()`. Тот оператор `trace()`, который расположен сразу после вызова метода `load()`, всегда выполняется раньше, чем оператор `trace()` в методе `loaderCompleteHandler()`. Это связано с тем, что метод `load()` объекта `URLLoader` выполняется асинхронно. В результате оператор `trace()`, следующий за вызовом `load()`, выполняется раньше, чем завершится загрузка данных. Вследствие этого первый оператор `trace()` всегда выводит `null`, так как свойство `data` объекта `URLLoader` будет иметь значение `null`, пока не закончится загрузка данных. Это важный результат асинхронного программирования. Если оператор выполняется асинхронно, его результат нельзя получить сразу после начала его выполнения.

Поскольку результаты асинхронных операций становятся доступны не сразу, асинхронное программирование требует более тщательного планирования, чем чисто синхронное программирование. Например, не нужно пытаться что-то сделать с данными запроса `load()`, прежде чем они действительно будут доступны. Асинхронное программирование может помочь приложению сохранить работоспособность но для этого нужно обеспечить правильную последовательность выполнения кода. Для этого можно воспользоваться событиями и их обработчиками (event listeners). Пример находится в листинге 3.2. Объекты могут посылать события, когда происходит нечто, заслуживающее внимания. Например, когда объект `URLLoader` завершает загрузку данных в запросе `load()`, он рассылает событие `complete`. Можно зарегистрировать у объекта метод, который будет вызываться, когда этот объект генерирует событие определенного типа. Такой метод называют обработчиком события. В листинге 3.2 объект `URLLoader` является источником события, посылающим событие `complete`, а метод `loaderCompleteHandler()` является обработчиком события, который вызывается, когда объект `URLLoader` рассылает событие `complete`. Такой способ гарантирует, что при вызове метода `loaderCompleteHandler()` данные объекта `URLLoader` окажутся загружены и доступны.

Как видите, хотя синхронное программирование и самый распространенный вид программирования, даже в обычном коде ActionScript есть операции, которым внутренне присуща асинхронность. Большинство базовых операций ActionScript являются либо синхронными, либо асинхронными, но не теми и другими одновременно. Однако многие операции AIR существуют в двух вариантах – синхронном и асинхронном. Это дает разработчику больше гибкости при написании кода, но

и возлагает на него больше ответственности в выборе правильной версии для каждого данного случая.

Как правило, синхронные операции более подходят в тех случаях, когда они выполняются быстро. Поскольку синхронная операция, требующая для своего выполнения большого времени, может вызвать впечатление зависания приложения, в таких случаях обычно рекомендуется применять асинхронные версии операций.

В этой главе вы научитесь с помощью класса `flash.filesystem.File` получать список содержимого каталога локальной файловой системы. У этой операции AIR есть как синхронная, так и асинхронная версии. Применение синхронной версии кажется интуитивно оправданным, поскольку при этом возвращается массив с содержимым каталога, которым можно немедленно воспользоваться. Например, в листинге 3.3 извлекается содержимое каталога (из объекта типа `File` под именем `documentsDirectory`), а потом этот результат сразу присваивается свойству `dataProvider` компоненты списка.

Листинг 3.3. Синхронное получение содержимого каталога

```
directoryContentList.dataProvider =
➥documentsDirectory.getDirectoryListing();
```

Если в каталоге находится много файлов и подкаталогов, операция может занять много времени. Из-за этого в плеере Flash остановятся все другие операции. В зависимости от того, что происходит в этот момент в приложении, может возникнуть ситуация, когда пользователь не сможет нажимать на кнопки, остановится воспроизведение видео или анимаций, звук сделается прерывистым, и приложение в целом не будет ни на что реагировать, пока не завершится операция получения содержимого каталога. В одних случаях впечатление зависания программы во время этой операции не вызывает беспокойства, но в других может быть нежелательным. Тогда можно воспользоваться асинхронной версией операции – `getDirectoryListingAsync()`. В асинхронном варианте эта операция, как почти любые асинхронные операции ActionScript, не возвращает значений. Вместо этого вы должны зарегистрировать обработчик события, возникающего при завершении операции. В случае `getDirectoryListingAsync()` объект `File`, для которого вызывается метод, отправит событие `directoryListing`, когда завершится операция получения содержимого каталога. Так как метод сразу не возвращает список содержимого, нужно дождаться возникновения события и только после этого читать данные или что-либо с ними делать. Фрагмент кода в листинге 3.4 иллюстрирует асинхронный вывод содержимого каталога.

Листинг 3.4. Асинхронное получение содержимого каталога

```
private function getDocumentsDirectoryListing():void {
    var documentsDirectory:File = File.documentsDirectory;
      documentsDirectory.addEventListener(FileListEvent.DIRECTORY_LISTING,
```

```
        ➥directoryListingHandler);
        documentsDirectory.getDirectoryListingAsync();
    }

    private function directoryListingHandler(event:FileListEvent):void {
        directoryListingList.dataProvider = event.files;
    }
```

Если сравнить листинги 3.3 и 3.4, то видно, что асинхронное получение результата требует большего объема кода и более сложной логики, чем синхронное получение того же результата. Однако асинхронное программирование позволяет создавать приложения, которые не замирают, даже если отдельные операции выполняются длительное время.

На протяжении этой главы и всей оставшейся книги мы будем отмечать, когда у операции есть синхронная и асинхронная версии, и часто станем приводить примеры для каждой из них. Если вам понадобится вспомнить, чем различаются синхронное и асинхронное программирование, вернитесь к этому разделу.

3.1.1. Отмена асинхронных файловых операций

При вызове асинхронной операции над файловой системой узнать о ее завершении можно благодаря отправке события. Например, при асинхронном запросе содержимого каталога объект `File`, из которого вызван метод, по завершении операции отправляет событие `directoryListing`. Хотя вы можете предполагать, что в какой-то момент операция будет завершена, у вас нет достоверного способа узнать, как долго будет выполняться эта операция. Вполне возможно, что вы (разработчик приложения AIR) или пользователь вашего приложения решите, что операция слишком долгая. Например, получение содержимого каталога, копирование файла или запись в него могут оказаться такими долгими, что пользователь не захочет дожидаться конца. AIR позволяет отменить любую асинхронную операцию с файловой системой путем вызова метода `cancel()` объекта `File`, инициировавшего запрос. Нужно помнить, что хотя вызывать асинхронные методы можно как для объектов `File`, так и для объектов `FileStream`, отменять их нужно с помощью метода `cancel()` объекта `File`.

Примечание

Вызвать замирание вашего приложения могут не только синхронные операции ввода/вывода файлов, но и обычный код нефайлового ввода/вывода. Допустим, ваше приложение читает графический файл, а затем применяет алгоритм распознавания лиц, чтобы выделить изображения лиц для дальнейшей обработки. Файл можно прочесть в память асинхронным способом, но после его загрузки алгоритм распознавания будет выполняться синхронно. В зависимости от конструкции и эффективности алгоритма приложение также может показаться зависшим. Если ваш код может создать впечатление зависания приложения, переработайте его логику, чтобы распределить обработку по разным кадрам или промежуткам времени.

Все операции с файловой системой, о которых вы узнаете в этой главе, требуют знания ссылки на конкретный каталог или файл на машине пользователя. Например, чтобы получить содержимое каталога, вам нужна ссылка на этот каталог. В следующем разделе вы узнаете, как получать ссылки на файлы и каталоги.

3.2. Получение ссылок на файлы и каталоги

Работая на компьютере, вы можете многими способами получить доступ к файлу или каталогу. Чтобы переместиться к нужному файлу или каталогу, можно воспользоваться файловым менеджером, можно использовать ярлык или псевдоним, а также получить доступ из панели запуска, начального меню или через их функции. Точно так же AIR предоставляет множество способов программного доступа к файлам и каталогам. В следующих далее разделах мы увидим, какие существуют способы доступа к файлам и каталогам и в каких случаях следует их применять.

3.2.1. Знакомство с классом File

Для представления файлов и каталогов в локальной файловой системе пользователя в AIR существует класс `flash.filesystem.File`. Ниже вы познакомитесь с разными способами получения ссылок на файлы и каталоги. Каким бы из этих способов вы ни воспользовались, вы будете работать с объектом `File`. Имея объект `File`, вы можете с помощью его методов осуществлять такие операции, как создание, чтение, запись и удаление файлов и каталогов.

Иногда у вас есть объект `File`, но вы не знаете, что он представляет, – файл или каталог. Например, вы получаете массив объектов `File` в результате чтения содержимого каталога, и нужно определить, какие его элементы являются файлами, а какие – каталогами. Определить, что объект `File` является каталогом, можно с помощью свойства `isDirectory`. Если оно имеет значение `false`, то этот объект – файл.

Теперь, после первого знакомства с классом `File`, посмотрим, как с его помощью получать ссылки на стандартные каталоги локальной файловой системы.

3.2.2. Ссылки на стандартные каталоги

В любой операционной системе есть несколько важных стандартных каталогов. Например, в Windows и OS X у пользователя есть каталоги desktop и documents. Поскольку они являются стандартными, вам, вероятно, часто придется обращаться к ним в своих приложениях AIR. Но даже несмотря на их стандартность, задача правильного определения точного пути к ним оказывается нетривиальной. К счастью, AIR поможет нам в ее решении, предоставив простые способы обращения к этим каталогам через статические свойства класса `File`. Они не толь-

ко удобны, но и не зависят от платформы. Это означает, что, даже несмотря на сильное различие путей к каталогам рабочего стола пользователя на машинах под Windows и OS X, AIR позволит совершенно одинаковым образом получить на них ссылку с помощью свойства `File.desktopDirectory`. В таблице 3.1 перечислены эти каталоги и статические свойства, служащие для обращения к ним.

Как вы могли догадаться, все перечисленные в табл. 3.1 статические свойства сами являются объектами `File`. Это значит, что можно вызывать для этих объектов любые методы класса `File`. Например, если вы хотите (синхронно) получить содержимое каталога рабочего стола пользователя, нужно выполнить код, представленный в листинге 3.5.

Таблица 3.1. Стандартные каталоги, не зависящие от платформы

Стандартный каталог	Описание	Свойство
Домашний каталог пользователя	Корневой каталог учетной записи пользователя	`File.userDirectory`
Документы пользователя	Каталог документов, обычно лежащий в домашнем	`File.documentsDirectory`
Рабочий стол	Каталог, представляющий рабочий стол	`File.desktopDirectory`
Хранилище приложения	Уникальный каталог хранилища для каждого установленного приложения	`File.applicationStorageDirectory`
Приложение	Каталог, в который установлено приложение	`File.applicationDirectory`

Листинг 3.5. Чтение содержимого каталога рабочего стола пользователя

```
var listing:Array = File.desktopDirectory.getDirectoryListing();
```

Есть еще один способ добраться до двух из этих специальных каталогов – приложения и хранилища приложений. При создании нового объекта `File` можно передать конструктору параметр, содержащий путь к файлу или каталогу, к которому вы хотите обращаться с помощью этого объекта. Класс `File` поддерживает две особые схемы: `app` и `app-storage`.

Примечание

Вероятно, вам лучше знакомы такие схемы, как `http` и `https` (или даже `file`), поскольку вы могли видеть их в веб-броузерах. Например, в адресе *http://www.manning.com* `http` является схемой. AIR позволяет применять `app` и `app-storage` в качестве схем для объектов `File`. Если вы применяете их как схемы, то после них должны следовать двоеточие и косая черта (слэш).

В следующем примере создается ссылка на каталог приложения:

```
var applicationDirectory:File = new File("app:/");
```

Обратите внимание, что этот код равносилен использованию `File.applicationDirectory`.

Нет сомнений, что ссылки на стандартные каталоги удобны. Но едва ли они могут удовлетворить все потребности любого приложения AIR. Как быть, например, если вам нужна ссылка на файл в каталоге документов, а не на сам этот каталог? Тогда вам нужно сделать еще один шаг, воспользовавшись относительными или абсолютными ссылками. Сначала мы рассмотрим относительные ссылки.

3.2.3. Относительные ссылки

Одна из замечательных особенностей AIR заключается в возможности легкого создания кросс-платформенных приложений. При создании кросс-платформенных приложений получение ссылок может представлять проблему. AIR в определенной мере облегчает ее решение, обеспечивая встроенные ссылки на стандартные каталоги, о которых говорилось в предыдущем разделе. Поэтому, если вам удастся обойтись ссылками, указывающими пути относительно стандартных каталогов, то вы обеспечите кросс-платформенность своих приложений.

Альтернативой относительным ссылкам являются ссылки абсолютные. Рассмотрим пример, в котором противопоставлены эти два способа ссылки на каталоги, и выясним, какие трудности могут возникнуть. Представим себе, что мы пишем приложение, которому нужно записать текстовый файл в подкаталог (назовем его notes) каталога документов пользователя. Сначала посмотрим, как ссылаться на этот каталог в абсолютном виде. Сразу возникает проблема: абсолютный адрес каталога документов пользователя в Windows и OS X различен. Под Windows каталог документов пользователя обычно располагается в *БуквеДиска*:\Documents and Settings*имя_пользователя*\My Documents, где *БуквеДиска* – это присвоенное диску имя из одной буквы (чаще всего C), а *имя_пользователя* – это имя того, кто авторизован в системе в данный момент. В OS X каталогом документов пользователя обычно является /Users/*имя_пользователя*/Documents, где *имя_пользователя* – это имя того, кто авторизован в системе в данный момент. Даже если мы правильно определим операционную систему, нам не известно, какое имя пользователя нужно взять. Даже если нам удастся собрать все данные, необходимые для правильного определения пути к подкаталогу notes в каталоге документов пользователя, для этого придется проделать большую работу, и хотелось бы идти более легким путем.

Более удачный выбор состоит в том, чтобы при всякой возможности пользоваться относительными, а не абсолютными ссылками на файлы и каталоги. Мы уже видели, как получить ссылку на каталог документов пользователя посредством `File.documentsDirectory`. Теперь нам нужно только выяснить, как ссылаться на каталог относительно этого пути. Класс File предоставляет для этого простой метод `resolvePath()`. Можно передать методу `resolvePath()` строку, содержащую относитель-

ный путь к каталогу или файлу, и он разрешит его в подкаталог или файл относительно объекта File, из которого вы вызвали этот метод. В нашем примере можно получить ссылку на подкаталог notes каталога документов с помощью такого кода:

```
var notesDirectory:File = File.documentsDirectory.resolvePath("notes");
```

Метод resolvePath() можно использовать для доступа к подкаталогу или файлу внутри каталога, как приведено в примере. Но с помощью resolvePath() можно, кроме того, получить доступ к подкаталогу или файлу, находящемуся еще глубже в дереве каталогов. Например, следующий код дает путь к файлу reminders.txt этого самого подкаталога notes, лежащего внутри каталога документов пользователя.

```
var reminders:File = File.documentsDirectory.resolvePath(
➥"notes/recent/reminders.txt");
```

Обратите внимание, что разделителем имен каталогов и файлов в пути является прямой слэш (/) – аналогично тому, как это принято в системах Unix. Вы должны использовать в качестве разделителя обычную косую черту. Обратная косая черта имеет в строках ActionScript особый смысл и не будет восприниматься в качестве разделителя пути.

Для ссылки на каталог, находящийся в пути на один уровень выше, можно передать две точки. Например, следующий код получает путь к родительскому каталогу для каталога документов пользователя:

```
var parent:File = File.documentsDirectory.resolvePath("..");
```

Можно не только получать относительные пути с помощью resolvePath(), но и вычислять относительный путь от одного файла или каталога к другому с помощью метода getRelativePath(). Этот метод требует, чтобы ему передали ссылку на объект File, относительный путь к которому нужно получить. Например, следующий код определяет относительный путь от каталога пользователя до каталога документов:

```
var relativePath:String = File.userDirectory.getRelativePath(
➥File.documentsDirectory);
```

Обычно каталог документов является подкаталогом каталога пользователя. Например, в Windows значением relativePath будет My Documents, потому что каталог документов является подкаталогом пользовательского каталога и носит название My Documents.

Если относительный путь не является подкаталогом или файлом, находящимся в подкаталоге объекта File, из которого был вызван метод getRelativePath(), то по умолчанию метод возвращает пустую строку. Но может быть задан необязательный второй параметр, указывающий, можно ли использовать в пути обозначение «две точки». Если этот параметр имеет значение true, то getRelativePath() возвратит значение, даже если путь ведет вне каталога, из которого вызван метод.

Как мы уже подчеркивали, следует по возможности применять относительные ссылки. Они не только позволяют строить более гибкие

и надежные системы, работающие на разных платформах, но и обычно оказываются гораздо проще в применении. Относительные ссылки обеспечат почти любые ваши потребности в отношении работы с файловой системой. Тем не менее, существуют ситуации, когда необходимо использовать абсолютные ссылки на файлы или каталоги. Далее мы посмотрим, как поступать в таких случаях.

3.2.4. Абсолютные ссылки

При необходимости можно ссылаться на каталоги и файлы, указывая их абсолютные пути. Это можно сделать, непосредственно передав полный путь к каталогу или файлу конструктору объекта `File`, например:

```
var documentsAndSettings:File = new File("C:/Documents and Settings/");
```

Из этого примера видно, что даже если путь явно указывает на каталог в системе Windows, в качестве разделителя в нем применяется обычная косая черта. В отличие от метода `resolvePath()` (который требует разделителей в виде обычной косой черты), в пути, передаваемом конструктору `File`, можно использовать обратную косую черту. Обратные и прямые слэши в пути, передаваемом конструктору `File`, интерпретируются одинаково. Однако прямыми слэшами пользоваться несколько проще, т. к. обратная косая черта требует экранирования в виде еще одной обратной косой черты, что приводит к появлению такого кода:

```
var documentsAndSettings:File = new File("C:\\Documents and Settings\\");
```

Когда вам требуется указать файл или каталог с помощью абсолютного пути, нужно знать, какие корневые каталоги есть в системе. В Windows, например, C:\ часто оказывается основным системным диском, но нельзя полагаться на то, что так будет на любой машине. Можно получить массив объектов `File` со ссылками на все корневые каталоги системы с помощью статического метода `File.getRootDirectories()`.

3.2.5. Получение полного пути

Независимо от того, какая у вас есть ссылка на файл или каталог – абсолютная или относительная, вам может понадобиться полный путь в том виде, как он выглядит в системе. Эту информацию предоставит вам свойство `nativePath`, имеющееся у всех объектов `File`. Листинг 3.6 содержит простой тест, запустив который вы увидите системные пути всех статических свойств `File` класса `File`.

Листинг 3.6. Системные пути стандартных каталогов

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    creationComplete="creationCompleteHandler();">
   <mx:Script>
```

```
<![CDATA[
   import flash.filesystem.File

   private function creationCompleteHandler():void {
    print(File.userDirectory.nativePath);
    print(File.documentsDirectory.nativePath);
    print(File.desktopDirectory.nativePath);
print(File.applicationStorageDirectory.nativePath);
    print(File.applicationDirectory.nativePath);
   }

   private function print(string:String):void {
    output.text += ">" + string + "\n";
   }

]]>
</mx:Script>
<mx:TextArea id="output" width="100%" height="100%" />
</mx:WindowedApplication>
```


В этом листинге функция `print()` ❶ добавляет значения `nativePath` для стандартных каталогов к компоненте текстового поля. При этом выводимые значения зависят от следующих факторов: операционной системы, имени пользователя, названия приложения AIR и идентификатора приложения AIR. Вот воображаемый сценарий, который позволит продемонстрировать, какие значения могут быть выведены. В этом сценарии имя пользователя (под которым он зарегистрирован в системе) Christina, идентификатор приложения com.manning.airinaction.ExampleApplication, а само приложение называется Example Application. Тогда, если приложение выполняется под Windows, оно выведет следующие данные:

1. C:\Documents and Settings\Christina
2. C:\Documents and Settings\Christina\My Documents
3. C:\Documents and Settings\Christina\Desktop
4. C:\Documents and Settings\Christina\Application Data\com.manning. airinaction.ExampleApplication. AFA83DFB7118641978BF5E9EE3C49B0A3C82FA13.1\Local Store
5. C:\Program Files\Example Application

При тех же параметрах на компьютере OS X результат будет такой:

1. /Users/Christina
2. /Users/Christina/Documents
3. /Users/Christina/Desktop
4. /Users/Christina/Library/Preferences/com/manning.airinaction/ExampleAppliction. AFA83DFB7118641978BF5E9EE3C49B0A3C82 FA13.1/Local Store
5. /Applications/Example Application.app/Contents/Resources

Хотя мы рассмотрели примеры извлечения системного пути только для стандартных системных каталогов, свойство nativePath можно использовать с любым объектом File, который ссылается на любой файл или каталог в системе пользователя.

3.2.6. Произвольные ссылки

Итак, мы знаем, как создавать ссылки на файлы и каталоги – относительные и абсолютные. Те и другие вполне пригодны, когда приложение AIR может определить файл или каталог, к которым нужно обратиться. Например, если вы знаете, что в хранилище приложения должен быть подкаталог с именем preloadedAssets, то можете получить на него ссылку следующим образом:

```
var preloadedAssets:File = File.applicationDirectory.resolvePath(
➥"preloadedAssets");
```

Однако существует множество ситуаций, когда приложение AIR не может заранее знать, к какому файлу или каталогу придется обратиться. Например, если приложение должно показать содержимое каталога, выбранного пользователем, а не фиксированного подкаталога в каталоге приложения, рассмотренных нами до сих пор приемов окажется недостаточно. Потребуется средство, с помощью которого пользователь сможет задать ссылку. AIR предоставляет четыре метода класса File, с помощью которых пользователь может выбрать файл или каталог. Они перечислены в табл. 3.2.

Все эти методы открывают диалоговое окно, с помощью которого пользователь может перемещаться по файловой системе и выбрать файл (а в одном случае – несколько файлов) или каталог. Открываемые этими методами диалоговые окна весьма схожи между собой, хотя имеют некоторые различия.

Таблица 3.2. Методы класса File, открывающие диалоговое окно

Метод	Описание
browseForDirectory()	Выбор каталога
browseForOpen()	Выбор файла для открытия
browseForOpenMultiple()	Выбор нескольких файлов для открытия
browseForSave()	Выбор места сохранения файла

Выбор каталога

Метод browseForDirectory() открывает диалоговое окно, показанное на рис. 3.1. Окно предлагает пользователю выбрать каталог. В этом окне для выбора доступны только каталоги. Под заголовком окна есть поле для текста. На данном рисунке текст гласит «Select a directory». Этот текст задается с помощью параметра, передаваемого методу browseForDirectory().

Рис. 3.1. *Метод browseForDirectory( ) открывает диалоговое окно подобного вида*

Исходный каталог, отображаемый в окне, определяется тем, на какой каталог ссылается объект `File`, из которого вызван данный метод. На рис. 3.1 показан диалог, который откроется в результате выполнения следующего кода:

```
var documents:File = File.documentsDirectory;
documents.browseForDirectory("Select a directory");
```

Если вызвать `browseForDirectory()` для объекта `File`, указывающего на несуществующий файл или каталог, то выбранным каталогом окна окажется первый из вышерасположенных каталогов, который существует. Если никакая часть пути не указывает на существующий каталог, то выбранным каталогом по умолчанию станет каталог рабочего стола.

Выбор одного или нескольких файлов

Методы `browseForOpen()` и `browseForOpenMultiple()` открывают диалоговое окно, которое предлагает пользователю выбрать не каталог, а файлы. Различие между методами состоит в том, что с помощью `browseForOpen()` можно выбрать только один файл, а с помощью `browseForOpenMultiple()` – один или несколько файлов. Оба окна выглядят одинаково – как на рис. 3.2.

Оба метода требуют задать строку, которая появится в заголовке окна. На рис. 3.2 в заголовке окна помещен текст «Select a file», но может быть задан и любой другой. Вот код, который открывает окно, показанное на рис. 3.2:

```
var desktop:File = File.desktopDirectory;
desktop.browseForOpen("Select a file");
```

Методы `browseForOpen()` и `browseForOpenMultiple()` определяют исходный каталог окна таким же образом, как метод `browseForDirectory()`. На рис. 3.2 начальным каталогом является рабочий стол. Это обусловлено

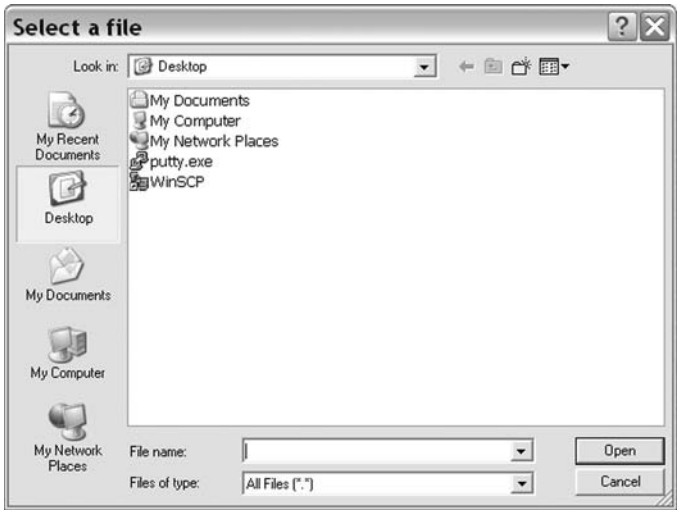


Рис. 3.2. *Методы browseForOpen( ) и browseForOpenMultiple( ) открывают диалоги для выбора пользователем файлов*

тем, что открывший окно код вызвал метод `browseForOpen()` из объекта `File`, который указывает на рабочий стол.

Эти методы позволяют также задать необязательные фильтры, определяющие типы файлов, которые позволено выбирать пользователю. Для этого методам передается второй параметр: массив объектов `flash.net.FileFilter`. (Класс `FileFilter` входит в стандартную библиотеку ActionScript, поэтому мы не будем подробно останавливаться на нем в этой книге). Каждый элемент `FileFilter` создает новый пункт в меню Files of type диалогового окна, дающий пользователю возможность видеть только файлы заданных типов. Следующий код иллюстрирует создание массива фильтров и применение его с методом `browseForOpen()`:

```
var file:File = File.desktopDirectory;
var filters:Array = new Array();
filters.push(new FileFilter("JPEG Images", "*.jpg"));
filters.push(new FileFilter("GIF Images", "*.gif"));
filters.push(new FileFilter("PNG Images", "*.png"));
filters.push(new FileFilter("All Images", "*.jpg;*.gif;*.png"));
file.browseForOpenMultiple("Select a file", filters);
```

В этом примере мы добавили четыре фильтра: JPEG images, GIF images, PNG images и All images. На рис. 3.3 показано, как будет выглядеть в результате диалоговое окно.

Есть еще один способ дать пользователю возможность выбрать файл, который мы теперь и рассмотрим.

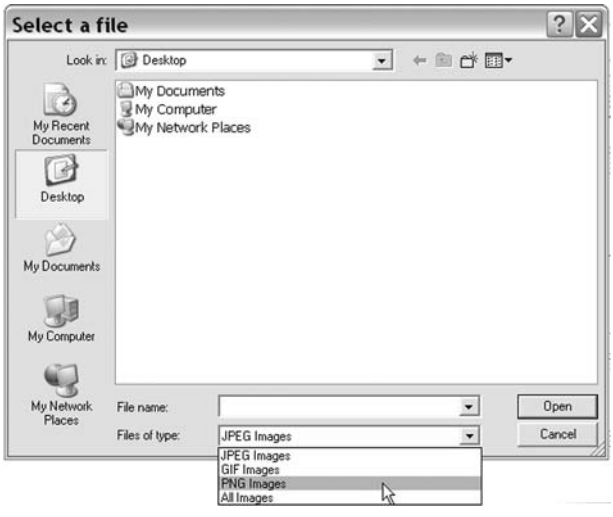


Рис. 3.3. *С помощью фильтров пользователь может выбрать тип файлов, которые будут показаны*

Выбор места сохранения файла

До сих пор мы рассматривали методы выбора каталогов и файлов, предполагая необходимость чтения из файла или каталога. Есть еще одна ситуация, когда нужно дать пользователю возможность перемещения по файловой системе: для выбора места, где он может сохранить файл. Для этого предназначен метод `browseForSave()`.

Как и другие методы навигации, `browseForSave()` требует задания параметра со строкой, показываемой пользователю. Как и методы `browseForOpen()` и `browseForOpenMultiple()`, метод `browseForSave()` показывает эту строку в заголовке окна.

Диалоговое окно Save дает пользователю возможность перемещаться по файловой системе и выбрать существующий файл либо выбрать каталог и ввести имя нового файла. Пример диалогового окна Save приведен на рис. 3.4.

Метод `browseForSave()` определяет начальный открываемый каталог по тем же правилам, что и остальные методы навигации.

Как определить, сделал ли пользователь выбор

Теперь мы умеем вызывать несколько методов перемещения по файловой системе. Но нужно еще уметь определить, какой выбор сделал пользователь в диалоговом окне. Все методы навигации действуют асинхронно. Приложение не прекращает выполнение кода, если открыто диалоговое окно для навигации. Оно должно только обрабатывать

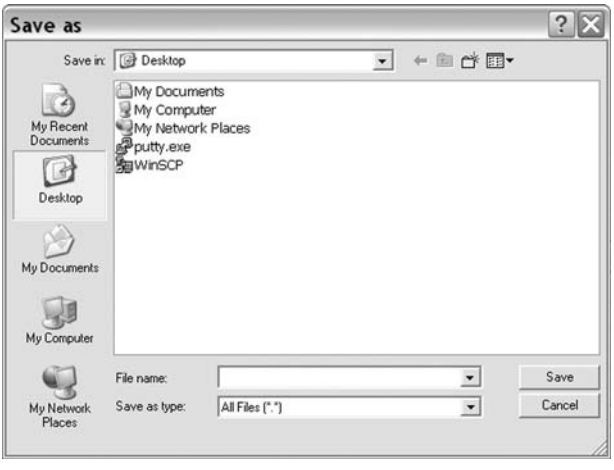


Рис. 3.4. *Диалоговое окно Save дает пользователям возможность выбрать файл для сохранения каких-либо данных из приложения AIR*

определенные события, посылаемые в результате выбора пользователем файла или каталога или отмены операции.

Методы `browseForDirectory()`, `browseForOpen()` и `browseForSave()` посылают событие одного и того же типа, когда пользователь выбирает каталог или файл, и в этом отношении они действуют идентично. Когда пользователь выбирает каталог или файл (т. е. щелкает в окне по кнопкам Open или Save), происходят две вещи:

1. Объект `File`, открывший диалоговое окно, автоматически обновляется, чтобы указывать на выбранный файл или каталог.
2. Этот же объект `File` отправляет событие `select`.

Листинг 3.7 содержит законченный пример, в котором пользователь может выбрать файл, а затем информация о выбранном файле отобразится в текстовой области.

Листинг 3.7. Обработка события select с целью определения выбора пользователем файла

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 creationComplete="creationCompleteHandler();">
    <mx:Script>
    <![CDATA[
       import flash.filesystem.File;

       private function creationCompleteHandler():void {
          var file:File = File.desktopDirectory;
          file.browseForOpen("Select a file");
          file.addEventListener(Event.SELECT, selectEventHandler);
       }
```

```
        private function selectEventHandler(event:Event):void {
           var file:File = event.target as File;
           output.text = "File: " + file.name;
           output.text += "\nPath: " + file.nativePath;
        }

    ]]>
    </mx:Script>
    <mx:TextArea id="output" width="100%" height="100%" />
</mx:WindowedApplication>
```

Метод `browseForOpenMultiple()` **похож на остальные методы навигации, но имеет и некоторые отличия. Когда пользователь выбирает один или несколько файлов в диалоговом окне, открытом с помощью** `browseForOpenMultiple()`**, объект** `File` **отправляет событие** `selectMultiple` **типа** `flash.events.FileListEvent`**. У объекта** `FileListEvent`**, соответствующего действию, есть свойство** `files`**, представляющее собой массив объектов** `File`**, каждый из которых ссылается на один из файлов, выбранных пользователем. Пример приведен в листинге 3.8.**

Листинг 3.8. Метод browseForOpenMultiple( ) может отправлять событие selectMultiple

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 creationComplete="creationCompleteHandler();">
    <mx:Script>
    <![CDATA[
       import flash.filesystem.File;

       private function creationCompleteHandler():void {
           var file:File = File.desktopDirectory;
           file.browseForOpenMultiple("Select files");
           file.addEventListener(FileListEvent.SELECT_MULTIPLE,
                           selectEventHandler);
       }

       private function selectEventHandler(event:FileListEvent):void {
           var file:File;
           output.text = event.files.length + " files selected";
           for(var i:Number = 0; i < event.files.length; i++) {
              file = event.files[i] as File;
              output.text += "\nFile: " + file.name;
              output.text += "\nPath: " + file.nativePath;
              output.text += "\n\n";
           }
       }

    ]]>
    </mx:Script>
    <mx:TextArea id="output" width="100%" height="100%" />
</mx:WindowedApplication>
```

Щелчок пользователя по кнопке Cancel в любом из этих диалоговых окон вызывает одинаковый результат: генерацию события `cancel`.

3.2.7. Красивое отображение путей

В ряде случаев пути к каталогам и файлам на компьютере отображаются приложением AIR не так, как хотелось бы разработчику. Возможны три случая:

- Регистр символов в пути объекта `File` не соответствует регистру, принятому в системе (например, /aPpliCations вместо /Applications).
- Путь объекта File имеет сокращенный вид (например, C:/docume~1 вместо C:/Documents and Settings).
- Путь содержит символическую ссылку, а вы хотели бы показать тот путь, на который указывает символическая ссылка.

Для всех этих ситуаций есть решение: метод `canonicalize()`. Этот метод класса `File` автоматически решает каждую из указанных проблем. Нужно лишь вызвать метод `canonicalize()` для объекта `File` после того, как путь, на который он указывает, получен любым из обсуждавшихся в этой главе способов. Сказанное иллюстрирует листинг 3.9.

Листинг 3.9. Исправление регистра символов пути с помощью canonicalize( )

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
 creationComplete="creationCompleteHandler();">
    <mx:Script>
    <![CDATA[
       import flash.filesystem.File;

       private var _file:File;

       private function creationCompleteHandler():void {
          _file = new File();
          _file.addEventListener(Event.SELECT, selectHandler);
       }

       private function browse():void {
          _file.browseForOpen("Select a File");
       }

       private function selectHandler(event:Event):void {
          output.text = _file.nativePath;
          _file.canonicalize();
          output.text += "\n" + _file.nativePath;
       }

    ]]>
    </mx:Script>
    <mx:Button label="Browse" click="browse();" />
    <mx:TextArea id="output" width="100%" height="100%" />
</mx:WindowedApplication>
```

Этот код дает пользователю возможность выбрать файл ❶. Чтобы нужным образом проверить этот пример, не выбирайте файл, а введите его имя в текстовом окне, притом так, чтобы регистр букв не совпадал с тем, что показывает диалоговое окно. Например, если есть файл sample.txt, задайте его как SaMpLe.TxT в поле для ввода имени. Щелчок по кнопке Open вызывает метод `selectHandler()`, который показывает неисправленный путь ❷, приводит к каноническому виду ❸ и показывает исправленный путь ❹. Исправленный путь показывает имя файла так, как оно выглядит в файловой системе (т. е. sample.txt вместо SaMpLe.TxT).

Приведенный пример показывает тонкое различие, получаемое с помощью метода `canonicalize()`. Теперь возьмем более яркий пример. Мы зададим путь для папки приложения Flex Builder 3 в Mac OS X (/Applications/Adobe Flex Builder 3). Представим, что этот пример выполняется на машине с Mac OS X, и на ней существует такой путь. Если прямо указать этот путь, изменив регистр некоторых букв, все равно получится допустимый объект `File`:

```
var flex:File = new File("/APPlicaTIONs/adoBE flex bUilDER 3");
trace(flex.nativePath);     <-- Выводит «/APPlicaTIONs/adoBE flex bUilDER 3»
```

Этот пример выводит /APPlicaTIONs/adoBE flex bUilDER 3 вместо /Applications/Adobe Flex Builder 3. Чтобы путь соответствовал реальным именам файлов и каталогов, воспользуемся методом `canonicalize()`:

```
var flex:File = new File("/APPlicaTIONs/adoBE flex bUilDER 3");
trace(flex.nativePath);     <-- Выводит «/APPlicaTIONs/adoBE flex bUilDER 3»
flex.canonicalize();
trace(flex.nativePath);     <-- Выводит «/Applications/Adobe Flex Builder 3»
```

После применения `canonicalize()` регистр букв в пути меняется на тот, который задан в файловой системе. Как будет показано ниже, допустимо создание объектов `File`, указывающих на еще не существующие файлы и каталоги (для того чтобы создать их). Поэтому если часть пути объекта `File` указывает на не существующие в файловой системе объекты, `canonicalize()` исправит регистр только в той части, которая существует. Рассмотрим пример, в котором есть ссылка на каталог quantum FLEX 8, отсутствующий в системе. Поскольку каталог Applications действительно существует, метод `canonicalize()` исправит регистр пути в части Applications и сохранит без изменений остальное.

```
var quantumFlex:File = new File("/APPlicaTIONs/ADoBe quantum FLEX 8");
quantumFlex.canonicalize();
trace(quantumFlex.nativePath);     <-- Выводит «/Applications/ADoBE quantum FLEX 8»
```

Как уже говорилось, `canonicalize()` не только корректирует регистр, но и превращает короткие имена Windows в соответствующие им длинные имена (в предположении, что участки пути существуют). Из программных соображений Windows требует, чтобы ко всем файлам и каталогам был доступ в системе обозначений 8.3, т. е. имена файлов и каталогов можно было сократить до 8 символов (плюс 3 символа для

расширения имени файла). Длинное имя – это то, которое обычно показывает вам Windows Explorer, и поскольку в нем может быть больше символов, оно более понятно пользователю. Например, стандартный путь к местонахождению Flex Builder в системе под Windows – это C:\Program Files\Adobe\Flex Builder 3. Сокращенная форма данного пути может иметь вид C:\Progra~1\Adobe\FlexBu~1. Следующий фрагмент кода демонстрирует преобразование короткой формы имени в длинную с помощью `canonicalize()`:

```
var flex:File = new File("C:/Progra~1/Adobe/FlexBu~1");
trace(flex.nativePath);     <-- Выводит «C:\Progra~1\Adobe\FlexBu~1»
flex.canonicalize();
trace(flex.nativePath);     <-- Выводит «C:\Program Files\Adobe\Flex Builder 3»
```

Есть еще одно применение `canonicalize()`, связанное с разрешением символических ссылок (OS X) и ярлыков (Windows). Символическая ссылка почти неотличима от каталога, на который она указывает. Однако свойство `isSymbolicLink` объекта `File` позволяет определить, указывает ли объект `File` на символическую ссылку. Если да, то с помощью `canonicalize()` можно получить путь к тому каталогу, на который указывает эта ссылка.

Пользоваться `canonicalize()` очень удобно, когда приложению не известен точный регистр имен файлов и каталогов и эти пути нужно показать пользователю. Точное отображение регистра не только придает приложению профессиональный вид, но и оберегает пользователя от ошибок.

На этом мы завершаем обсуждение ссылок на файлы и каталоги. Теперь, когда вы знаете, как получать эти ссылки, нужно выяснить, что можно делать с их помощью. Этим мы и займемся в оставшейся части главы, начав с получения листинга содержимого каталога, о чем рассказывается в следующем разделе.

3.3. Вывод содержимого каталога

Допустим, мы создаем приложение, которое поможет пользователю расчистить свой захламленный рабочий стол. Очевидно, для начала нам нужно получить перечень находящихся на нем файлов и каталогов. Только после этого можно начать их сортировку и организацию.

Мы уже знаем, как получить ссылку на каталог рабочего стола с помощью свойства `File.desktopDirectory`. Теперь нам нужно получить перечень содержимого этого каталога. Класс `File` предлагает два удобных способа это осуществить, один из которых синхронный, а другой – асинхронный. Имя синхронного метода `getDirectoryListing()`, а асинхронного – `getDirectoryListingAsync()`. Оба метода возвращают массив объектов `File`, каждый из которых содержит ссылку на файл или каталог, содержащиеся в исходном каталоге. Однако способы, которыми возвращаются объекты `File`, будут разными в зависимости от того, которым из методов вы пользуетесь. Рассмотрим оба метода, начав с синхронного.

3.3.1. Синхронное получение содержимого каталога

Метод `getDirectoryListing()` выполняется синхронно и сразу возвращает массив объектов `File`. Это простейший способ получения перечня содержимого каталога. Следующий пример иллюстрирует получение содержимого каталога рабочего стола пользователя. Поскольку список содержимого доступен сразу, следующий код обходит все элементы массива и показывает их в текстовой области `textArea`:

```
var desktopContents:Array = File.desktopDirectory.getDirectoryListing();
for(var i:Number = 0; i < desktopContents.length; i++) {
   textArea.text += dektopContents[i].nativePath + "\n";
}
```

Конечно, как уже отмечалось, у синхронных операций есть недостатки. Скажем, если на рабочем столе находится слишком много файлов, то приведенный код вызовет замирание приложения на период своего выполнения. Поэтому более удачной может оказаться асинхронная версия этой операции. В следующем разделе рассматривается асинхронное получение перечня содержимого каталога.

3.3.2. Асинхронное получение содержимого каталога

Асинхронно получить содержимое каталога можно с помощью метода `getDirectoryListingAsync()`. При этом, как и для большинства асинхронных операций, код становится объемнее и сложнее, чем для соответствующего синхронного варианта. В этом случае мы не получаем сразу же перечень содержимого каталога. Вместо этого объект `File` посылает событие `directoryListing`, когда операция будет выполнена. Событие `directoryListing` имеет тип `File_ListEvent`, и у объекта события есть свойство `files`, содержащее массив объектов `File` для данного каталога.

Листинг 3.10 содержит законченный пример класса ActionScript, который перечисляет содержимое каталога асинхронным образом.

Листинг 3.10. Пример асинхронного вывода содержимого каталога

```
package com.manning.books.airinaction {

   import flash.display.Sprite;
   import flash.events.FileListEvent;
   import flash.filesystem.File;
   import flash.text.TextField;

   public class Example extends Sprite {

     private var _textField:TextField;

     public function Example() {
       _textField = new TextField();
       _textField.width = stage.stageWidth;
       _textField.height = stage.stageHeight;
       addChild(_textField);
       var desktop:File = File.desktopDirectory;
```

❶ Создать текстовое поле

❷ Получить ссылку на рабочий стол

```
            desktop.addEventListener(FileListEvent.DIRECTORY_LISTING,
                                     desktopListingHandler);
            desktop.getDirectoryListingAsync();
        }
        private function desktopListingHandler(event:FileListEvent):void {
            var files:Array = event.files;
            var file:File;
            for (var i:Number = 0; i < files.length; i++) {
                file = files[i] as File;
                _textField.appendText(">" + file.nativePath + "\n");
            }
        }
    }
}
```

Перехватить directoryListing ❸

Сделать асинхронный запрос

Получить содержимое каталога ❹

Добавить путь в текстовое поле ❺

В этом примере сначала создается текстовое поле ❶ для показа содержимого каталога. Нам нужно содержание каталога рабочего стола, поэтому получаем ссылку на рабочий стол ❷. Поскольку мы применяем асинхронную операцию, нужно зарегистрировать обработчик события `directoryListing` ❸. Теперь можно вызывать метод `getDirectoryListingAsync()`. Когда произойдет событие `directoryListing` и будет вызван его обработчик, получим массив с содержимым каталога из свойства `files` объекта события ❹. Теперь можно перебрать все объекты, содержащиеся в массиве, и вывести для каждого из них системный путь ❺.

До сих пор мы оперировали с существующими в файловой системе объектами. Теперь мы посмотрим, как создавать каталоги в файловой системе.

3.4. Создание каталогов

Создание каталогов может показаться разработчику веб-приложений идеей, взятой из какой-то другой области. Но настольное приложение должно уметь создавать каталоги. Рассмотрим следующий сценарий. Вы написали приложение, с помощью которого пользователь может организовать на своем компьютере все документы Word. Вам нужно, чтобы пользователь мог переместить свои файлы в новую, лучше организованную структуру каталогов. Для этого приложение AIR должно уметь создавать необходимые каталоги.

Для создания нового каталога с помощью AIR нужно выполнить два действия:

1. Создать объект `File`, указывающий на новый несуществующий каталог.
2. Вызвать метод `createDirectory()` этого объекта `File`.

Следующий пример иллюстрирует, как это делается.

```
var recentDocuments:File = File.documentsDirectory("wordFiles/recent");
recentDocuments.createDirectory();
```

Ссылка на несуществующий каталог ❶

❷ Создание каталога

В этом листинге сначала создается объект File, указывающий на подкаталог пользовательского каталога документов ❶. Предполагается, что в каталоге документов нет каталога wordFiles, а следовательно и подкаталога wordFiles. При этом нужно учитывать два важных обстоятельства:

- Объекты File могут ссылаться на несуществующие каталоги (и файлы). Некоторые операции могут не работать, если каталог или файл не существуют (например, нельзя переместить несуществующий каталог), зато другие операции, такие как создание нового каталога, требуют указания несуществующего каталога или файла.
- При создании нового каталога с помощью createDirectory() создаются все необходимые каталоги и подкаталоги внутри пути. В данном случае будут созданы как каталог wordFiles, так и его подкаталог recent.

В данном примере мы создали объект File, который ссылается на каталог, который мы хотели бы создать. После этого достаточно вызвать метод createDirectory() этого объекта ❷, и в файловой системе будет создан каталог.

Если каталог уже существует, createDirectory() просто ничего не станет делать. Благодаря этому createDirectory() является относительно безопасной операцией. Можно не бояться, что при создании каталога с именем, которое уже существует, вы затрете находящиеся в нем данные. Однако в некоторых случаях желательно выяснить, не существует ли уже данный каталог. Например, в последнем примере может оказаться, что у пользователя уже есть каталог wordFiles в каталоге с документами, и он использует его для своих целей, не связанных с вашим приложением AIR. Вместо того чтобы мусорить в уже существующем каталоге wordFiles было бы приличнее, если бы приложение AIR создало новый каталог с уникальным именем. Чтобы проверить, существует ли каталог, достаточно прочесть свойство exists объекта File, указывающего на этот каталог. Сказанное иллюстрируется следующим примером:

```
var count:int = 0;
var recentDocuments:File = File.documentsDirectory("wordFiles");

while(recentDocuments.exists) {
   recentDocuments = File.documentsDirectory("wordFiles" + count);
   count++;
}
recentDocuments = File.documentsDirectory("wordFiles" + count + "/recent");
recentDocuments.createDirectory();
```

❶ Создать начальную ссылку

❷ Проверить, существует ли каталог

❸ Создать новое имя каталога

❹ Увеличить счетчик

❺ Добавить подкаталог recent

В этом листинге сначала создается объект File, который ссылается на каталог wordFiles ❶. Затем в операторе while выполняется проверка существования каталога ❷. Пока каталоги существуют, продолжаем менять имя ❸ с помощью постоянно увеличивающегося числа ❹. Как

только окажется, что имя уникальное, добавим подкаталог recent ❺ и создадим сразу два каталога.

Теперь рассмотрим несколько более полный действующий пример, в котором организация рабочего стола происходит путем перемещения всех файлов в подкаталоги в соответствии с расширением имени файла. Код листинга 3.11 можно использовать как класс документа для приложения AIR на базе Flash.

Листинг 3.11. Класс документа для приложения на базе Flash – организатора рабочего стола

```
package {

   import  flash.display.MovieClip;
   import  flash.events.MouseEvent;
   import  flash.display.Stage;
   import  flash.filesystem.File;
   import  flash.events.FileListEvent;
   import  fl.controls.Button;
   import  fl.controls.TextArea;

   public class DesktopOrganizer extends MovieClip {

      public function DesktopOrganizer() {
         _button.addEventListener(MouseEvent.CLICK, startOrganize);
      }

      private function startOrganize(event:MouseEvent):void {
         var organized:File = File.desktopDirectory.resolvePath(
         ➥"Files Organized By Type");
         if(!organized.exists) {
             organized.createDirectory();
             print("created directory: " + organized.nativePath);
         }
         var desktop:File = File.desktopDirectory
         desktop.addEventListener(FileListEvent.DIRECTORY_LISTING,
         ➥directoryListingHandler);
         desktop.getDirectoryListingAsync();
      }

      private function directoryListingHandler(event:FileListEvent):
      ➥void {
         var files:Array = event.files;
         var file:File;
         var organized:File = File.desktopDirectory.resolvePath(
         ➥"Files Organized By Type");
         var extension:File;
         for(var i:Number = 0; i < files.length; i++) {
            file = files[i] as File;
            if(!file.isDirectory) {
               extension = organized.resolvePath(file.extension);
               if(!extension.exists) {
                  extension.createDirectory();
```

❶ Упорядочение начинается по щелчку кнопки

❷ Получить ссылку на целевой каталог

❸ Проверить существование каталога

❹ Создать каталог

❺ Получить ссылку на рабочий стол

❻ Получить содержимое каталога

❼ Цикл по файлам

❽ Проверить, что это файл

❾ Получить ссылку к каталогу типа

❿ Создать каталог

```
                print("created directory: " +
                  ➥extension.nativePath);
            }
        }
    }
}

private function print(message:String):void {
    textArea.text += message + "\n";
}

}
}
```

В этом примере предполагается, что на сцену файла Flash помещены две компоненты интерфейса пользователя: кнопка `_button` и текстовая область `_textArea`. Кнопка используется для запуска пользователем процедуры организации рабочего стола ❶. После щелчка по кнопке мы сначала получаем ссылку на каталог, в который в итоге будут перемещены все файлы ❷. Этот каталог называется Files Organized By Type и располагается на рабочем столе. Если такого каталога нет ❸, его нужно создать ❹. Затем нужно получить ссылку на рабочий стол пользователя ❺ и список его содержимого ❻. Когда получен список, просматриваем его ❼ и для каждого элемента проверяем, является он файлом или каталогом ❽. Если это не каталог, мы хотим определить путь к подкаталогу, имя которого совпадает с расширением файла ❾. Например, если расширение файла .jpg, нужно будет переместить файл в каталог Files Organized By Type/ jpg. Если такого каталога еще нет ❿, мы его создадим ❿. Правда, мы еще не научились перемещать файлы. Придется немного подождать, а потом снова вернуться к этому примеру и довести его до конца.

Примечание

В приведенном примере каталог Files Organized By Type будет автоматически создан при создании подкаталогов. Поэтому нет особой необходимости явно создавать Files Organized By Type в методе `startOrganize()`. Однако для ясности кода мы все же решили его создать отдельно.

Иногда возникает нужда во временном каталоге. Например, нужен каталог для записи файлов во время работы приложения, но по завершении приложения надобность в нем отпадает. Хорошим тоном будет создать такой каталог в системном каталоге для временного хранения. AIR облегчает эту задачу, предоставляя статический метод `File.createTempDirectory()`. Метод возвращает новый объект `File`, указывающий на этот каталог. При каждом вызове этого метода приложение AIR создает новый каталог с уникальным именем в системном временном каталоге. Следующий фрагмент кода иллюстрирует применение этого метода:

```
var temporary:File = File.createTempDirectory();
trace(temporary.nativePath);
```

❶ Путь к временному каталогу

На машине под Windows приведенный код ❶ может напечатать что-то вроде C:\Documents and Settings*имя_пользователя*\Local Settings\Temp\fla1A.tmp. На машине под OS X код ❶ напечатает что-нибудь вроде /private/var/tmp/folders.2119876841/TemporaryItems/FlashTmp0.

Пока существует этот объект `File`, можно обращаться к данному временному каталогу. Так как `createTempDirectory()` всегда создает новый каталог, вы не сможете повторно обратиться к тому же каталогу, вызвав этот метод во второй раз. Пока вам нужно обращаться к этому каталогу, вам нужна ссылка на объект `File`, возвращаемый методом. Когда надобность во временном каталоге отпала, полезно, чтобы приложение произвело за собой уборку и уничтожило его. Если вы этого не сделаете, удалять каталог придется системе или пользователю. Как удалять каталоги (и файлы), будет показано в следующем разделе.

3.5. Удаление каталогов и файлов

Есть два способа удаления каталогов и файлов. Их можно удалить навсегда или поместить в корзину для мусора. Рассмотрим каждый из этих способов.

Чтобы удалить каталог, достаточно вызвать метод `deleteDirectory()` или `deleteDirectoryAsync()`. Файл удаляется столь же просто, хотя методы другие: `deleteFile()` или `deleteFileAsync()`. Разумеется, этими методами нужно пользоваться с осторожностью. Отменить результат их выполнения нельзя. Разумнее всего применять эти методы при следующих обстоятельствах:

- Вы хотите навсегда удалить каталог или файл, которые создало приложение AIR, и о которых не знает пользователь или они ему не нужны.
- Вы запрашиваете у пользователя разрешение удалить каталог или файл навсегда.

Методы `deleteFile()` и `deleteFileAsync()` идентичны, но первый из них выполняется синхронно, а второй – асинхронно. Относительно того, каким из них лучше пользоваться, совет будет обычным: асинхронный метод позволяет выполняться остальному коду приложения без его зависания, даже если удаляемый системой файл имеет большой размер. Оба метода не требуют параметров.

Методы `deleteDirectory()` и `deleteDirectoryAsync()` идентичны, но первый из них выполняется синхронно, а второй – асинхронно. Если известно, что каталог велик, всегда лучше удалять его асинхронно. По умолчанию оба метода удаляют только пустые каталоги. Если в каталоге что-то лежит, методы генерируют ошибку. Однако можно передать методу необязательный логический параметр со значением `true`, чтобы сообщить о своем желании удалить каталог вместе со всем его содержимым.

В следующем примере сначала создается каталог, а потом он уничтожается:

```
var directory:File = File.createTempDirectory();
directory.deleteDirectory();
```

Перемещение каталога или файла в мусорную корзину отвечает хорошему тону, когда нужно удалить каталог или файл при иных обстоятельствах, чем перечисленные выше. Например, даже если пользователь решит удалить каталог с помощью приложения AIR, обычно лучше просто переместить каталог в корзину, чем явно спрашивать у пользователя, действительно ли он хочет удалить каталог навсегда. Файл или каталог перемещаются в корзину одинаковыми методами. Для перемещения каталога или файла в корзину есть два способа: `moveToTrash()` и `moveToTrashAsync()`. Методы не требуют параметров. Оба метода просто перемещают каталог в корзину, один синхронно, другой – асинхронно. Если каталог велик, обычно лучше воспользоваться асинхронной операцией.

3.6. Копирование и перемещение файлов и каталогов

Копирование и перемещение файлов и каталогов – простые стандартные операции. Для каждой из них в классе `File` определены методы: `copyTo()`, `copyToAsync()`, `moveTo()` и `moveToAsync()`. Эти методы не делают различий между каталогами и файлами.

Копирование и перемещение крайне похожи друг на друга. По сути, в том и другом случае файл или каталог перемещаются в новое место в иерархии файловой структуры. Разница в том, что при копировании экземпляр файла или каталога сохраняется также и в исходном месте. Сходство между двумя операциями влечет и сходство между двумя методами. В любом случае методы требуют одного параметра: объекта `FileReference`, указывающего новый адрес. Все методы позволяют также задать еще один логический параметр, который определяет, можно ли переписывать то содержание, которое может уже находиться по новому адресу.

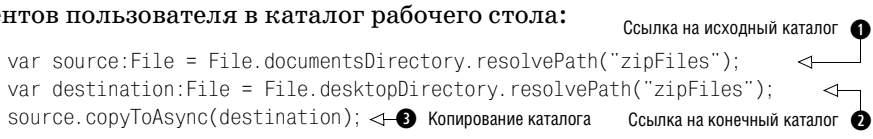
Примечание

Класс `File`, о котором мы много говорили в этой главе, является подклассом `FileReference`. Это означает, что всякий раз, когда требуется объект `FileReference`, можно использовать `File`. Методы копирования и перемещения требуют параметра `FileReference`. В этой главе во всех практических случаях в качестве этого параметра будет использоваться объект `File`.

Как и в других случаях наличия синхронных и асинхронных операций, обычно лучше отдать предпочтение асинхронным методам копирования и перемещения. Если приходится копировать или перемещать

большой файл или каталог, то асинхронные методы работают лучше, поскольку не вызывают замирания приложения AIR во время выполнения операции.

В следующем примере каталог zipFiles копируется из каталога документов пользователя в каталог рабочего стола:



В данном случае предполагается, что в каталоге документов есть каталог под именем zipFiles ❶, и что каталога с именем zipFiles нет в каталоге рабочего стола ❷. Это важный момент. Как мы вскоре убедимся, для того, чтобы этот код правильно сработал, конечного каталога вначале быть не должно. После создания ссылок на исходный и конечный каталоги можно скопировать источник с помощью метода `copyToAsync()` ❸.

Если вы запустите этот код во второй раз, он сгенерирует ошибку, поскольку объект `File` целевого каталога будет указывать на уже существующий каталог. Код в такой форме исходит из того, что конечный каталог не должен существовать в момент начала копирования. Но если известно, что целевой каталог может уже существовать, возникает вопрос, нужно ли переписывать при копировании то, что может уже находиться в нем. Если нужно переписывать, то достаточно передать значение `true` в качестве второго параметра метода `copyToAsync()`:

```
source.copyToAsync(destination, true)
```

По умолчанию этот необязательный параметр имеет значение `false`, но если задать `true`, то `copyToAsync()` сначала удалит конечный файл или каталог, и только потом станет копировать данные. Обратите внимание, что такое поведение отличается от привычного, поскольку удаляются все файлы и каталоги в целевом каталоге независимо от того, есть ли они в источнике.

Примечание

Все, что мы обсуждали применительно к `copyToAsync()`, относится и к другим методам копирования и перемещения.

Все операции копирования и перемещения генерируют ошибку ввода/вывода, если не существует источника или ОС запрещает действие вследствие блокировки файла. Синхронные методы сразу генерируют ошибку, и их нужно заключать в конструкцию try/catch, как в следующем примере:

```
try {
   file.moveTo(destination);
}
catch (error:IOError) {
   trace("an error occurred");
}
```

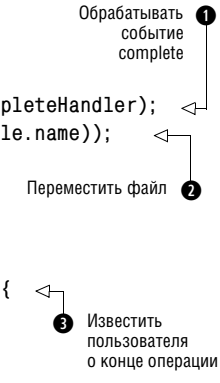
Асинхронные методы сообщают об ошибках с помощью событий. Если есть возможность возникновения такой ошибки, нужно зарегистрировать обработчики этих событий.

Мы сейчас вернемся к более старому примеру из листинга 3.11. В листинге 3.12 показан обновленный код, производящий перемещение файлов и каталогов в зависимости от расширений файлов. Модификации выделены жирным шрифтом.

Листинг 3.12. Перемещение файлов в новые каталоги в соответствии с расширением файла

```
package {

   import flash.display.MovieClip;
   import flash.events.MouseEvent;
   import flash.display.Stage;
   import flash.filesystem.File;
   import flash.events.FileListEvent;
   import flash.events.Event;
   import fl.controls.Button;
   import fl.controls.TextArea;

   public class DesktopOrganizer extends MovieClip {

      public function DesktopOrganizer() {
         _button.addEventListener(MouseEvent.CLICK, startOrganize);
      }

      private function startOrganize(event:MouseEvent):void {
         var organized:File = File.desktopDirectory.resolvePath(
         ➥"Files Organized By Type");
         if(!organized.exists) {
            organized.createDirectory();
            print("created directory: " + organized.nativePath);
         }
         var desktop:File = File.desktopDirectory.resolvePath(
         ➥"To Organize");
         desktop.addEventListener(FileListEvent.DIRECTORY_LISTING,
                              directoryListingHandler);
         desktop.getDirectoryListingAsync();
      }

      private function directoryListingHandler(event:FileListEvent):
      ➥void   {
         var files:Array = event.files;
         var file:File;
         var organized:File = File.desktopDirectory.resolvePath(
         ➥"Files Organized By Type");
         var extension:File;
         for(var i:Number = 0; i < files.length; i++) {
            file = files[i] as File;
            if(!file.isDirectory) {
                extension = organized.resolvePath(file.extension);
                if(!extension.exists) {
```

```
                    extension.createDirectory();
                    print("created directory: " +
                    ➥extension.nativePath);
                }
                file.addEventListener(Event.COMPLETE, completeHandler);
                file.moveToAsync(extension.resolvePath(file.name));
                print("moving file: " + file.name);
            }
        }
    }

    private function completeHandler(event:Event):void {
        print("moved file: " + event.target.name);
    }

    private function print(message:String):void {
        _textArea.text += message + "\n";
    }

  }
}
```



В этом примере мы решили перемещать файлы асинхронно с помощью метода `moveToAsync()` ❷. Мы к тому же так вежливы, что перехватываем событие `complete` ❶ и извещаем пользователя, когда файл оказывается действительно перемещен ❸.

Мы научились работать с каталогами многими способами, а также копировать и перемещать каталоги и файлы. Теперь мы вполне готовы рассмотреть некоторые операции, специфичные для файлов. В следующем разделе мы научимся выполнять такие операции, как чтение, запись и удаление файлов.

3.7. Чтение и запись файлов

Мы уже видели, как получать информацию о файловой системе и изменять ее структуру путем создания новых каталогов, а также перемещения, копирования и удаления файлов и каталогов. Однако некоторые наиболее мощные операции, которые могут производить с файловой системой приложения AIR, включают в себя манипуляции с файлами и их содержимым. С помощью AIR можно совершать с файлами самые разнообразные действия, включая чтение из текстового файла, запись нового графического файла .png, загрузку видеофайла из сети и многое другое. Во всех этих задачах требуется чтение и/или запись данных в файл. Далее эти темы рассматриваются подробно.

3.7.1. Чтение из файлов

Для большинства разработчиков Flash и Flex идея чтения из файлов совсем не нова. Читать данные из текстовых файлов или ресурсов – обычная и довольно тривиальная задача при использовании ActionScript

или MXML. Более того, даже в сетевых приложениях Flash и Flex можно загружать файлы и обрабатывать двоичные данные с помощью классов ActionScript 3 `flash.net.URLStream` или `flash.net.URLLoader`. В чем же тогда состоит особенность AIR? Ответ состоит из трех частей:

- AIR позволяет приложению обращаться как к локальным файлам, так и файлам, находящимся в Интернете.
- AIR позволяет приложению осуществлять чтение из объекта `File`.
- AIR позволяет приложению писать данные в файл, что обычно недоступно для приложений Flash и Flex в сети.

Как уже отмечалось, есть два разных способа чтения данных из файлов с помощью AIR:

- Чтение из интернет-источников
- Чтение из локальных источников

В следующих двух разделах мы рассмотрим оба эти способа.

Чтение из интернет-источников

Чтение из интернет-источников приложения AIR осуществляют так же, как веб-приложения. В контексте темы нашей главы нас интересует только чтение байтов из файла или ресурса. Это означает, что есть два способа загрузки ресурсов и чтения данных: с помощью `URLStream` или с помощью `URLLoader` – двух классов, с которыми читатель должен быть знаком по опыту работы с Flash или Flex. Мы не станем обсуждать подробности работы с этими классами, но даже если вы не встречались с ними, последующего материала будет достаточно для получения необходимой информации.

Класс `URLLoader` загружает файл или ресурс целиком, и только после этого он становится доступен для чтения. Например, если вы загружаете из Интернета с помощью объекта `URLLoader` файл .jpg, файл должен быть загружен в приложение AIR целиком, чтобы можно было прочесть из него хотя бы первый байт. После того как ресурс загружен, объект `URLLoader` посылает событие `complete`, а в процессе загрузки контента объект `URLLoader` посылает события `progress`, позволяющие вашему приложению AIR следить за ходом загрузки.

Класс `URLStream` загружает файл или ресурс и делает байты доступными для чтения в процессе загрузки данных. Например, загружая с помощью объекта `URLStream` из Интернета файл .mp3, можно читать его байты сразу после окончания их загрузки. Объекты `URLStream` посылают события `progress` по мере того, как байты становятся доступными.

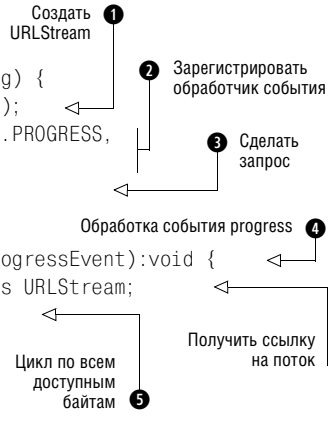
В некоторых контекстах различие между `URLLoader` и `URLStream` напоминает различие между синхронными и асинхронными операциями. Один из таких контекстов возникает при желании воспользоваться объектом `URLLoader` или `URLStream` для чтения и обработки байтов. Так как `URLLoader` не предоставит доступа к данным, пока ресурс не будет загружен целиком, придется ждать, пока не станут доступными все

байты. Если нужно просто записать эти байты в локальный файл, приложению придется ждать загрузки всего файла, а когда на него обрушится вся эта куча байтов, то приложение замрет, пытаясь записать все байты в файл. Напротив, объект `URLStream` делает доступными все байты по мере загрузки файла, благодаря чему приложение может время от времени писать данные на диск небольшими порциями, снижая вероятность появления ощущения зависания приложения.

Листинг 3.13 демонстрирует класс, который загружает интернет-ресурс и по одному выводит байты на консоль с помощью оператора `trace()`.

Листинг 3.13. Загрузка файлов и чтение байтов

```
package {
    import flash.events.ProgressEvent;
    import flash.net.URLRequest;
    import flash.net.URLStream;

    public class FileDownloader {

        public function FileDownloader(url:String) {
            var stream:URLStream = new URLStream();
            stream.addEventListener(ProgressEvent.PROGRESS,
            ➥progressHandler);
            stream.load(new URLRequest(url));
        }

        public function progressHandler(event:ProgressEvent):void {
            var stream:URLStream = event.target as URLStream;
            while(stream.bytesAvailable) {
                trace(stream.readByte());
            }
        }

    }
}
```



Это не очень реалистичный пример, поскольку он просто загружает файл и показывает байты. Но он иллюстрирует принципы запроса интернет-ресурса и чтения байтов, которые уже доступны. Сначала нужно создать объект `URLStream` ❶. Затем нужно зарегистрировать обработчик события `progress` ❷ и запросить загрузку ресурса ❸. Когда произойдет событие `progress` ❹, нужно считать имеющиеся байты ❺ ❻. Есть разные способы чтения данных из файла. Мы обсудим более подробно чтение двоичных данных чуть позднее. Сначала мы посмотрим, как читать данные из локальных ресурсов.

Примечание

Можно также воспользоваться объектом `flash.net.Socket` и читать двоичные данные из сокета соединения. В этой книге мы не станем вникать в детали использования класса `Socket` применительно к файлам. Но можно воспользоваться теми же принципами чтения из объектов `URLStream` или `FileStream`, чтобы применить их к объекту `Socket`.

Чтение локальных ресурсов

Для чтения из локальных ресурсов требуется объект File, с которым читатель уже знаком. Кроме того, требуется объект flash.filesystem.FileStream, который мы пока не обсуждали.

Объект FileStream позволяет выполнять с файлом операции чтения/записи. Необходим объект File, который указывает на файл. После этого посредством объекта FileStream этот файл открывается для чтения или записи с помощью методов open() или openAsync(). Вот основные предварительные шаги для осуществления чтения из файла:

1. Создать объект File, ссылающийся на файл, как в следующем примере:

```
var file:File = File.desktopDirectory.resolvePath("example.jpg");
```

2. Создать новый объект FileStream, используя конструктор, как в следующем примере:

```
var fileStream:FileStream = new FileStream();
```

3. Воспользоваться методами open() или openAsync() объекта FileStream, чтобы открыть файл для чтения. Сначала посмотрим, как пользоваться методом open(). Ему необходимо передать два параметра: ссылку на объект File, который вы хотите открыть, и значение константы flash.filesystem.FileMode.READ, как показано в следующем примере:

```
fileStream.open(file, FileMode.READ);
```

   Метод open() сразу делает файл открытым для чтения, поскольку это синхронная операция. С другой стороны, можно воспользоваться методом openAsync() и открыть файл для асинхронного чтения. Если вы читаете файл асинхронно, получить из него байты можно лишь тогда, когда поток уведомит приложение о готовности байтов путем отправки события progress. По мере того как байты становятся доступными для чтения, объект FileStream отправляет события progress – так же как объект URLStream отправляет события progress по мере доступности байтов. Следующий фрагмент кода показывает, как можно открыть файл для асинхронного чтения, а потом обрабатывать события progress:

```
private function startReading(fileStream:FileStream, file:File):void {
   fileStream.addEventListener(ProgressEvent.PROGRESS, progressHandler);
   fileStream.openAsync(file, FileMode.READ);
}

private function progressHandler(event:ProgressHandler):void {
   // код для чтения байтов
}
```

После того как вы выполните эти действия и откроете файл для чтения (и байты станут доступными), можно читать байты файла с помощью

любых методов чтения объекта FileStream. Например, следующий код читает все доступные байты файла и пишет их на консоль или в окно вывода с помощью команды trace():

```
while(fileStream.bytesAvailable) {
   trace(fileStream.readByte());
}
```

Независимо от того, как вы открыли файл и что прочли из него, как только вы завершили чтение данных, обязательно нужно закрыть доступ к файлу методом close() для того же объекта FileStream.

Теперь, когда мы в целом знаем, как читать из локальных ресурсов и ресурсов Интернета, посмотрим, как работать с двоичными данными.

Что такое двоичные данные

Людям не свойственно думать на языке двоичных данных. Если считать, что машины думают, то они это делают как раз с помощью двоичных данных. Компьютеры предпочитают двоичный формат. Все файлы хранятся в двоичном формате, и только благодаря компьютерным программам эти двоичные данные транслируются в доступный человеку вид, позволяющий людям их понимать. За возможность работать с файлами на низком уровне человеку приходится платить: он вынужден научиться думать на уровне двоичных данных. При чтении данных из объекта URLStream или FileStream мы обязаны решить, что нам делать с двоичными данными, которые может предоставить приложение AIR.

Как уже отмечалось, все файлы представляют собой двоичные данные. Любой файл – лишь последовательность битов, каждый из которых имеет одно из двух значений – 0 или 1. Биты объединяются в байты, обычно являющиеся мельчайшей неделимой структурой данных, с которыми мы работаем в контексте приложений AIR и действий с файлами. Соединяя вместе байты, можно представить любые данные – от простого текста до видеофайлов. Отличие между этими разными типами файлов не в способе хранения данных (это всего лишь последовательности байтов), а в значениях самих байтов. Так как тип данных всегда одинаков (байты), приложение AIR может читать (писать) любой файл. Теоретически, если иметь спецификацию того, какая последовательность байтов составляет видеофайл Flash, можно создать фильм с нуля с помощью одного лишь приложения AIR.

Оба класса – URLStream и FileStream – реализуют интерфейс ActionScript с именем flash.utils.IDataInput. Интерфейс IDataInput требует набора методов для чтения двоичных данных различными способами. Эти методы приведены в табл. 3.3.

Интерфейс IDataInput требует также свойства с именем bytesAvailable. Свойство bytesAvailable возвращает количество байтов, находящихся в буфере объекта (подробнее о чтении буферов см. в следующем разделе), и дает уверенность в том, что вы не пытаетесь прочитать больше байтов, чем доступно в данный момент.

Было бы слишком жестоко обрушить на вас всю эту информацию, не показав, как можно практически применять эти методы. В следующих разделах мы рассмотрим некоторые практические примеры стандартного использования этих методов. Подробно обсуждать нестандартные приемы мы не станем в надежде, что вы самостоятельно сможете получить необходимую информацию.

Чтение строк

В табл. 3.3 приведены три метода для чтения строк из двоичных данных: `readUTF()`, `readUTFBytes()` и `readMultiByte()`. В большинстве приложений метод `readUTF()` гораздо менее интересен, чем остальные два, поэтому мы не станем его обсуждать, а сосредоточим свое внимание на более полезных методах.

Таблица 3.3. Форматы данных, доступные для чтения из объектов, реализующих IdataInput

Тип формата	Формат	Описание	Связанные методы	Связанные типы объектов ActionScript

Необработанные байты	Byte	Один или несколько необработанных байтов	`readByte()`
			`readBytes()`
			`readUnsignedBytes()`
				`Int`
				`ByteArray`

Логический	Boolean	0 для false, иначе true	`readBoolean()`	`Boolean`
Числа	Short	16-разрядное целое	`ReadShort()` `readUnsignedShort()`	`Int` `uint`
	Integer	32-разрядное целое	`ReadInt()` `readUnsignedInt()`	`Int` `uint`
	Float	Число с плавающей точкой одинарной точности	`readFloat()`	`Number`
	Double	Число с плавающей точкой двойной точности	`readDouble()`	`Number`
Строки	Multibyte	Строка с заданным набором символов	`readMultiByte()`	`String`
	UTF-8	Строка в кодировке UTF 8	`ReadUTF()` `readUTFBytes()`	`String`
Объекты	Object	Объекты, сериализованные и десериализованные с применением ActionScript Message Format (AMF)	`readObject()`	Любой объект можно сериализовать с помощью AMF (см. детали в разделе «Чтение объектов»)

Метод readUTFBytes() возвращает строку, содержащую весь текст, хранящийся в последовательности байтов. Необходимо задать один параметр, указывающий нужное вам количество байтов. Не всегда, но часто желательно прочесть все доступные байты, следовательно, с помощью свойства bytesAvailable можно получить значение, передаваемое методу readUTFBytes().

Чтобы проиллюстрировать применение readUTFBytes(), возьмем простой пример. Это код для чтения текстового файла, в котором использованы объект FileStream и метод readUTFBytes(). Код приведен в листинге 3.14. Предполагается, что этот класс используется в качестве класса документа для проекта AIR на базе Flash, в котором есть кнопка _button и текстовое поле _textArea.

Листинг 3.14. Применение метода readUTFBytes( ) объекта FileStream

```
package {

   import  flash.display.MovieClip;
   import  flash.filesystem.File;
   import  flash.filesystem.FileStream;
   import  flash.filesystem.FileMode;
   import  flash.events.Event;
   import  flash.events.MouseEvent;
   import  flash.events.ProgressEvent;

   public class TextFileReader extends MovieClip {

      public function TextFileReader() {
         _button.addEventListener(MouseEvent.CLICK, browseForFile);
      }

      private function browseForFile(event:MouseEvent):void {
         var desktop:File = File.desktopDirectory;
         desktop.addEventListener(Event.SELECT, selectHandler);
         desktop.browseForOpen("Select a text file");
      }

      private function selectHandler(event:Event):void {
         var file:File = event.target as File;
         _textArea.text = "";
         var stream:FileStream = new FileStream();
         stream.addEventListener(ProgressEvent.PROGRESS,
                                 progressHandler);
         stream.addEventListener(Event.COMPLETE, completeHandler);
         stream.openAsync(file, FileMode.READ);
      }

      private function progressHandler(event:ProgressEvent):void {
         var stream:FileStream = event.target as FileStream;
         if(stream.bytesAvailable) {
             _textArea.text += stream.readUTFBytes(
            ➥stream.bytesAvailable);
         }
```

Выбор файла

Очистка текстового поля

Регистрация обработчика

Открытие файла на чтение

Проверка доступности байтов

Вывод байтов в виде строки

```
        }

        private function completeHandler(event:Event):void {
        event.target.close();          <-- Закрыте файлового потока
      }

    }

}
```

Как показывает пример, читать можно любой файл, не обязательно текстовый. Но поскольку в этом коде применяется `readUTFBytes()` для явной интерпретации данных в виде строки, результат будет иметь смысл, только если файл является текстовым.

Метод `readMultiByte()` удобен для чтения текста с использованием определенной кодовой страницы. Если вы не знакомы с понятием кодовых страниц, то знайте, что это способ, с помощью которого машина может узнать о том, какие символы сопоставлены различным значениям байтов. В зависимости от выбранной кодовой страницы одни и те же данные получают разное представление. Например, если рассматривать текстовый файл на машине, где задана кодовая страница для японского языка, он может выглядеть иначе, чем на машине с кодовой страницей для латиницы. Такие методы, как `readUTFBytes()`, используют кодовую страницу, установленную в системе по умолчанию. Если требуется применить нестандартную кодовую страницу, нужно применить метод `readMultiBytes()`. Метод `readMultiBytes()` требует, чтобы ему передали тот же параметр, что и для `readUTFBytes()`, но дополнительно он принимает второй параметр, указывающий кодовую страницу или набор символов, которые должны быть применены. Поддерживаемые наборы символов перечислены на livedocs.adobe.com/flex/3/langref/charset-codes.html. Например, следующий код читает файловый поток, используя расширенный набор японских символов для Unix:

```
var text:String = stream.readMultiByte(stream.bytesAvailable, "euc-jp");\
```

Одного того, что в коде указано на необходимость использования определенного набора символов, недостаточно, чтобы приложение успешно справилось с задачей. Например, если на компьютере нет некоторой кодовой страницы, приложение не сможет интерпретировать байты с помощью этой страницы. В таких случаях приложение AIR пользуется кодовой страницей, установленной в системе по умолчанию.

Чтение объектов

Еще один стандартный способ чтения данных из файла – это чтение их как объектов. Лучше всего, если этот файл был записан с помощью приложения AIR. Дело в том, что приложения AIR могут сериализовать большинство объектов в так называемом формате AMF, записать эти данные в файл, а потом при необходимости прочесть эти данные и восстановить из них объект. В этой главе мы рассмотрим несколько случаев таких операций.

Примечание

AMF активно применяется в Flash Player. `ByteArray`, `LocalConnection`, `NetConnection` и `URLLoader` – перечень лишь некоторых классов, основанных на AMF.

Вот как действуют сериализация и десериализация AMF. Во Flash Player есть встроенная поддержка данных AMF. Когда данные нужно передать вовне, они могут быть переведены в AMF. Обычно сериализация происходит автоматически. Например, если с помощью объекта `NetConnection` осуществляется вызов Flash Remoting, данные автоматически сериализуются и десериализуются с использованием AMF. Аналогично, при записи данных в объект SharedObject они автоматически сохраняются в формате AMF. При чтении данных из объекта SharedObject они десериализуются обратно в объекты ActionScript.

Есть два вида сериализации AMF: AMF0 и AMF3. AMF0 обратно совместим с ActionScript 2 и обычно не применяется в приложениях AIR. AMF3 поддерживает все базовые типы данных ActionScript 3, и именно он обычно применяется в приложениях AIR. AMF3 – это кодировка AMF, устанавливаемая по умолчанию. В кодировке AMF3 все стандартные базовые типы данных ActionScript автоматически сериализуются и десериализуются. В их число входят `String`, `Boolean`, `Number`, `int`, `uint`, `Date`, `Array` и `Object`. Для начала будем предполагать, что вы работаете только с этими базовыми типами данных.

Если ваш ресурс содержит данные AMF, вы можете читать эти данные с помощью метода `readObject()`. Метод `readObject()` возвращает один объект из последовательности байтов ресурса. Метод `readObject()` возвращает данные типа *, поэтому для присваивания результата переменной нужно привести тип. Вообще говоря, если вы читаете объект из файла или другого ресурса, то формат данных и тип объекта вам уже известны. Следующий пример иллюстрирует чтение объектов из файла. В данном примере предполагается, что в файле содержатся объекты `Array`, которые мы дописывали в него один за другим:

```
var array:Array;
while(fileStream.bytesAvailable) {
   array = fileStream.readObject() as Array;
   trace(array.length);
}
```

Если бы можно было читать из файлов только данные стандартных типов, польза была бы не велика. Но вы не ограничены работой только со стандартными типами. На самом деле, из файла можно читать данные любого пользовательского типа, если только в приложении AIR известен и доступен соответствующий класс ActionScript. Для этого требуются два основных действия:

- Написать и скомпилировать в приложении класс ActionScript для вашего типа данных.

- Зарегистрировать класс с псевдонимом.

Первый шаг должен быть вам знаком. Но чтобы гарантировать правильную десериализацию вашего класса, нужно знать еще несколько особенностей AMF:

- По умолчанию AMF сериализует и десериализует только открытые свойства, в том числе геттеры/сеттеры.[1] Поэтому вы должны определить открытые свойства или геттеры/сеттеры для всех свойств, которые вы хотите применять с AMF.
- Во время десериализации AMF значения открытых свойств устанавливаются до вызова конструктора. Поэтому нужно следить, чтобы код инициализации в вашем конструкторе не изменял значений свойств, если они уже заданы.

Со следующим шагом – регистрацией класса с псевдонимом – вы, возможно, не сталкивались. Основная идея следующая: если объект сериализуется в AMF, то возможно, что он будет читаться с помощью другой программы или языка, нежели те, которые его создавали. Поэтому ему нужно дать имя, по которому можно определить тип. Тогда любая программа или язык, знакомые с этим именем, смогут правильно десериализовать данные. Когда мы будем рассматривать запись данных, то подробнее поговорим о создании псевдонимов. Что касается чтения данных, то вам лишь необходимо знать используемый данными псевдоним, закодированный в файле или ресурсе. Если вы его не знаете, придется проконсультироваться с тем, кто записал данные в ресурс.

Есть два способа регистрации класса с псевдонимом. В ActionScript можно воспользоваться методом `flash.net.registerClassAlias()`, а для приложений Flex можно применить тег метаданных `[RemoteClass]`. Оба способа реализуют одну и ту же функцию. Если вы работаете во Flex, то обычно проще всего зарегистрировать класс с псевдонимом с помощью тега `[RemoteClass]`. Для этого нужно поместить тег `[RemoteClass]` перед определением класса и включить аргумент `alias`, который задает псевдоним данных, которые нужно сериализовать. Сказанное иллюстрирует пример:

```
package {
   [RemoteClass(alias="CustomTypeAlias")]
   public class CustomType {
      // Class code goes here
   }
}
```

Ели вы не работаете с Flex, нужно применить `registerClassAlias()`. Этот метод можно вызвать в любом месте приложения, но он должен выполниться прежде, чем вы попытаетесь прочесть данные из файла. Обычно вызов `registerClassAlias()` помещают где-нибудь среди начальных

[1] Методы-аксессоры (англ. getters/setters). Методы, получающие и устанавливающие значение свойства. – *Прим. науч. ред.*

операторов приложения. Первый параметр метода – это псевдоним в виде строки, а второй – ссылка на класс. В следующем примере регистрируется класс `CustomType` с псевдонимом `CustomTypeAlias`:

```
registerClassAlias("CustomTypeAlias", CustomType);
```

Зарегистрировав класс с псевдонимом, можно читать из ресурса объекты этого типа с помощью `readObject()`, и приложение AIR сможет правильно его десериализовать. Если пользовательский объект имеет свойства некоего другого пользовательского типа, надо обеспечить регистрацию и этого другого типа.

Некоторые примеры чтения пользовательских типов из файлов мы рассмотрим немного ниже, когда будем обсуждать запись в файлы. Прежде чем заниматься записью в файлы, посвятим некоторое время тому, чтобы глубже разобраться, как происходит чтение из файлов.

Буферы чтения

Объекты `FileStream` применяют так называемые буферы для чтения. Буфер для чтения можно рассматривать как контейнер с несколькими слотами для байтов. По мере готовности данных для чтения они помещаются в буфер для чтения.

Вызывая один из методов чтения объекта `FileStream`, вы фактически выполняете чтение из буфера для чтения, содержащего копию байтов из файла, а не сам файл.

Вообще говоря, есть два подхода к чтению из файлов: синхронное чтение и асинхронное чтение. Мы уже видели, как применять методы `open()` и `openAsync()` для запуска синхронного и асинхронного вариантов операции чтения. Но мы еще не рассматривали, каковы различия в синхронном и асинхронном чтении файлов при работе с буфером чтения.

Когда вы открываете файл для синхронного чтения, буфер чтения заполняется целиком. Иными словами, все байты файла копируются в буфер для чтения и становятся доступны для чтения. Так как в синхронной операции доступными для чтения становятся сразу все байты, существует, хотя и незначительная, опасность переполнения буфера. Свойство `bytesAvailable` помогает следить, чтобы не происходило запроса байтов за пределами буфера чтения. Например, если в файле 200 байтов (и буфер чтения содержит 200 байтов), не следует пытаться прочесть 202-й байт, которого не существует. Вот почему в примерах использовалась конструкция такого рода:

```
while(fileStream.bytesAvailable) {
   // Читать из буфера
}
```

В синхронной операции чтения `bytesAvailable` всегда сообщает, сколько байтов осталось после текущей позиции чтения в буфере. При чтении из буфера используется иная технология, чем при чтении из массива или в других знакомых вам операциях. Вместо чтения по заданно-

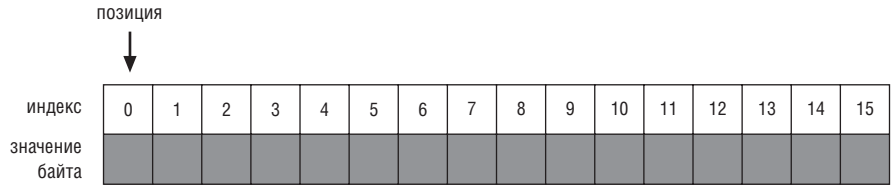


Рис. 3.5. *Буфер чтения с позицией чтения, находящейся в 0*

му индексу методы чтения `FileStream` (и других типов `IDataInput`) всегда читают очередные байты, начиная с текущей позиции чтения. Например, на рис. 3.5. показана принципиальная схема буфера чтения. У каждого из «слотов» для байтов есть свой индекс, который начинается с нуля. (На диаграмме изображено 16 слотов с индексами от 0 до 15.) Байты помещаются в эти слоты, после чего становятся доступны для чтения. На рис. 3.5 показана позиция чтения, находящаяся в начале буфера по индексу 0. В этот момент свойство `bytesAvailable` возвращает 16, потому что впереди текущей позиции чтения находятся 16 байтов.

Если прочесть из буфера четыре байта (например, четырежды вызвав `readByte()`), то позиция чтения переместится на 4, как показано на рис. 3.6. Свойство `bytesAvailable` будет теперь иметь значение не 16, а 12, потому что впереди позиции чтения останется всего 12 байтов.

При синхронном чтении можно переместить позицию чтения в любое место, воспользовавшись свойством `position`. Например, если вы прочли все или часть байтов из объекта `FileStream` и хотите снова прочесть байты из начала файла, нужно установить свойство `position` равным 0.

```
fileStream.position = 0;
```

До сих пор мы обсуждали буферы чтения только для синхронных операций чтения. Асинхронные операции взаимодействуют с буфером иным способом. При открытии файла для асинхронного чтения буфер заполняется не сразу. Вместо этого буфер чтения заполняется последовательно. Каждый раз, когда в буфер добавляются байты, `FileStream` посылает событие `progress`. На рис. 3.7 показана ситуация, которая может возникнуть в начале асинхронной операции, когда в буфер еще не записано никаких байтов.

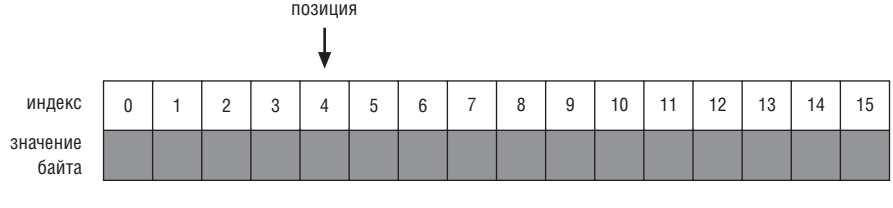


Рис. 3.6. *При чтении из FileStream позиция чтения перемещается по буферу*

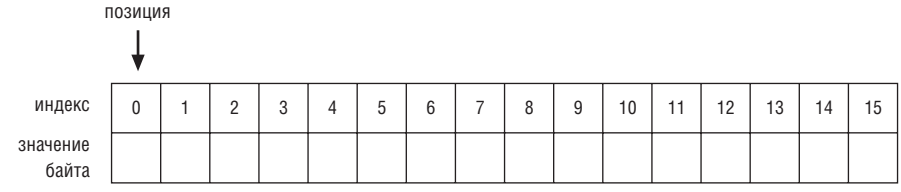


Рис. 3.7. *Белые квадратики указывают, что никаких байтов пока в буфер не считано, поскольку операция асинхронная*

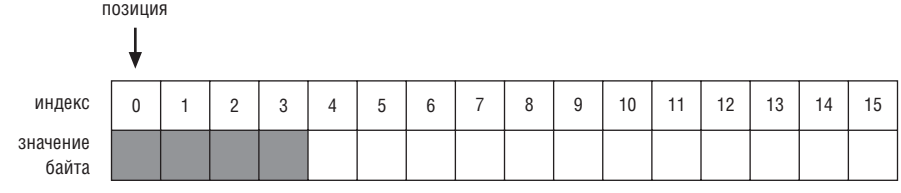


Рис. 3.8. *Буфер чтения, частично заполненный после первого события progress*

На рис. 3.8 показано, как может выглядеть буфер чтения после генерации первого события `progress`. Предполагается, что событие `first` происходит после чтения каждых очередных 4 байтов. Это число выбрано для удобства иллюстрации. На практике события `progress` сигнализируют о передаче значительно больших порций данных.

В этом примере значение `bytesAvailable` будет 4 после первого события `progress`, потому что после позиции чтения имеется только 4 байта. Рисунок 3.9 показывает, как может выглядеть буфер чтения после второго события `progress`, если считать, что загружено еще 4 байта.

В этом случае значение `bytesAvailable` будет 8 после второго события `progress`.

Обратите внимание, что как на рис. 3.8, так и на рис. 3.9 позиция остается в 0. Это связано с тем, что предположительно мы пока не начали читать из буфера. Но, как уже говорилось, вы можете читать все доступные после каждого события `progress` байты. Если бы мы читали

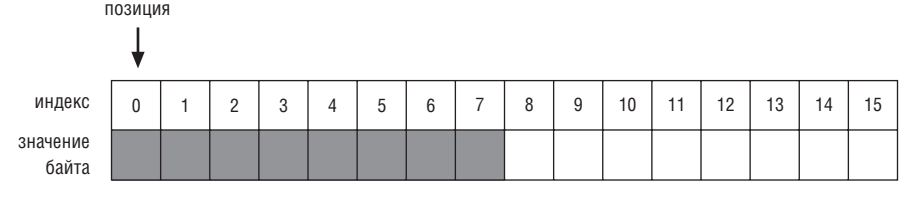


Рис. 3.9. *В буфере чтения стали доступны новые байты после второго события progress*

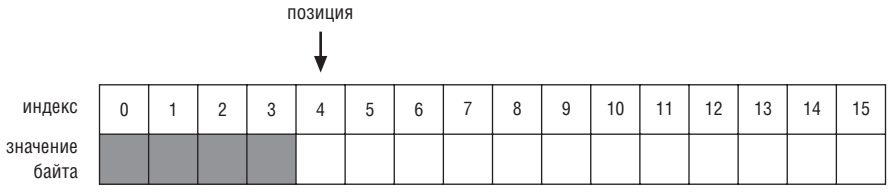


Рис. 3.10. *Чтение доступных байтов перемещает позицию чтения*

байты из буфера чтения после каждого события `progress`, то картинки были бы иными. Рисунок 3.10 показывает, как бы выглядел буфер чтения после первого события `progress`, если бы мы прочли все имеющиеся байты.

Если бы мы стали читать байты сразу, как только они делались доступными, это изменило бы значение `bytesAvailable`. Непосредственно перед чтением байтов значение `bytesAvailable` будет 4 в примере, иллюстрируемом рис. 3.10, но сразу после чтения байтов значением `bytesAvailable` станет 0, потому что после позиции чтения больше не окажется байтов.

Есть еще одно важное различие в работе буфера чтения при синхронном и асинхронном чтении. Как уже отмечалось, при синхронном чтении из файла можно заново прочесть байты из буфера, сбросив свойство `position` в нуль. При асинхронном чтении прочтенные из буфера данные удаляются и становятся недоступными, если их где-нибудь не сохранить.

Закончим на этом обсуждение чтения из файлов. Завершается глава обсуждением записи в файлы.

3.7.2. Запись в файлы

В некотором смысле запись в файл является операцией, обратной чтению. Читая из файла, вы извлекаете данные из него. При записи в файл вы добавляете в него данные. Из-за сходства этих операций вы увидите много аналогий между чтением и записью файлов. В предыдущем разделе был рассмотрен ряд наиболее сложных понятий, касающихся работы с файлами, включая чтение двоичных данных. Многое из того, что вы узнали, также применимо и к записи файлов, и если вам понятны идеи, относящиеся к чтению, то и запись не должна вызвать у вас проблемы.

В нескольких следующих разделах мы обсудим важные понятия, связанные с записью в файлы. Вы также встретите множество примеров, в которых чтение и запись будут объединены.

Выбор режима записи

Вам будет приятно узнать, что открытие файла для записи осуществляется почти так же, как и открытие файла для чтения. В обоих случаях

используется объект FileStream и вызывается метод open() или openAsync(). В обоих случаях методам open() или openAsync() передается ссылка на объект File, которым вы хотите воспользоваться. Но есть три отличия:

- При открытии файла для записи нужно задать параметр режима, который может иметь значение FileMode.WRITE, FileMode.APPEND или FileMode.UPDATE.
- При открытии файла для записи в асинхронном режиме нужно перехватить событие open, и только после этого можно начать запись в файл.
- Чтобы прочесть файл, он должен уже присутствовать в файловой системе, но для записи в файл это не обязательно. Если попытаться открыть файл, которого нет в файловой системе, AIR создаст этот файл и необходимые каталоги.

Второй и третий пункты не требуют особых пояснений, но мы поставили бы читателя в затруднительное положение, не остановившись подробнее на первом пункте. Осуществляя запись в файл, вы располагаете тремя значениями параметра для режима записи FileMode: WRITE, APPEND, и UPDATE. Необходимо следить за тем, чтобы выбрать правильный режим записи:

- WRITE – выберите режим WRITE, если хотите создать совершенно новый файл или заменить существующий файл. В этом режиме существующий файл очищается, и запись данных начинается с самого его начала.
- APPEND – выберите этот режим, если хотите добавить данные в конец существующего файла. Если этого файла еще не существует, в этом режиме он будет создан.
- UPDATE – выберите этот режим, если хотите одновременно писать в файл и читать из него. Этот режим похож на APPEND тем, что не удаляет имеющееся содержимое. Однако в то время как в режиме APPEND позиция записи автоматически перемещается в конец файла, в режиме UPDATE она сначала находится в начале файла.

В следующем примере открывается файл log.txt в каталоге хранилища приложения. Этот код открывает файл в режиме APPEND, поэтому при записи данные будут дописываться в конец файла:

```
var logFile:File = File.applicationStorageDirectory.resolvePath("log.txt");
var stream:FileStream = new FileStream();
stream.addEventListener(Event.OPEN, openHandler);
stream.openAsync(logFile, FileMode.APPEND);
```

Необходимо правильно выбирать режим – синхронный или асинхронный – при открытии файла для записи. При чтении файла основным фактором, определяющим режим, является размер файла, который нужно прочесть. Большой файл обычно следует открывать асинхронно.

Те же правила относятся к записи в файл: если вы собираетесь записать в файл много данных, следует открыть его в асинхронном режиме.

Как и при чтении из файла, после записи в файл также следует закрыть к нему доступ с помощью метода `close()`:

```
fileStream.close();
```

Теперь, когда мы знаем, как открыть файл для записи, посмотрим, как фактически писать в него данные.

Запись данных

В табл. 3.3 были приведены все методы интерфейса `IDataInput` для чтения из объектов, например, объектов `FileStream`. Приятно будет узнать, что методы записи соответствуют методам чтения. В табл. 3.4 перечислены все методы записи для объекта `FileStream`.

Таблица 3.4. Методы IdataOutput для записи данных

Тип формата	Формат	Описание	Связанные методы	Связанные типы объектов ActionScript

Необработанные байты	Byte	Один или несколько необработанных байтов	`writeByte()`
			`writeBytes()`
			`writeUnsignedBytes()`
				`Int`
				`ByteArray`

Логический	Boolean	0 для false, иначе true	`writeBoolean()`	`Boolean`
Числа	Short	16-разрядное целое	`writeShort()` `writeUnsignedShort()`	`Int` `uint`
	Integer	32-разрядное целое	`writeInt()` `writeUnsignedInt()`	`Int` `uint`
	Float	Число с плавающей точкой одинарной точности	`writeFloat()`	`Number`
	Double	Число с плавающей точкой двойной точности	`writeDouble()`	`Number`
Строки	Multi-byte	Строка с заданным набором символов	`writeMultiByte()`	`String`
	UTF-8	Строка в кодировке UTF 8	`writeUTF()` `writeUTFBytes()`	`String`
Объекты	Object	Объекты, сериализованные и десериализованные с применением ActionScript Message Format (AMF)	`writeObject()`	Любой объект можно сериализовать с помощью AMF

Примечание

Перечисленные в табл. 3.4 методы требуются интерфейсом `IdataOutput`. Класс `FileStream` реализует интерфейс `IDataOutput`, как и многие другие классы, например, `ByteArray` и `Socket`.

Большинство методов записи действует схожим образом. Мы не будем детально обсуждать каждый из них. Однако, как и ранее для чтения, мы рассмотрим пару наиболее распространенных методов записи данных в файл: текста и сериализованных объектов.

Запись текста в файл совершенно аналогична чтению текста из файла: для этого применяются методы `writeUTFBytes()` и `writeMultiByte()`. Метод `writeUTFBytes()` позволяет указать в виде параметра строку, которая будет записана в файл. Метод `writeMultiByte()` действует аналогично, за исключением того, что можно также задать кодировку символов. Листинг 3.15 содержит пример записи в файл журнала с помощью `writeUTFBytes()`. Здесь предполагается, что класс используется в качестве документа класса для проекта AIR на базе Flash и что на сцене есть экземпляр кнопки с именем `_button`.

Листинг 3.15. Запись в файл журнала с помощью writeUTFBytes( )

```
package {

    import flash.display.MovieClip;
    import flash.filesystem.File;
    import flash.filesystem.FileStream;
    import flash.filesystem.FileMode;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.events.ProgressEvent;

    public class LogFileWriter extends MovieClip {

       public function LogFileWriter() {
          _button.addEventListener(MouseEvent.CLICK, addLogEntry);
       }

       private function addLogEntry(event:MouseEvent):void {
          var file:File = File.desktopDirectory.resolvePath("log.txt");
          var stream:FileStream = new FileStream();
          stream.open(file, FileMode.APPEND);
          stream.writeUTFBytes("Log entry " + new Date() + "\n");
          stream.close();
       }

    }

}
```

Запись данных в виде сериализованных объектов имеет некоторое сходство с чтением сериализованных объектов. Любые данные, сериа-

лизуемые с помощью AMF, можно записать с помощью метода `writeObject()`. Например, следующий код записывает массив в файл:

```
var array:Array = new Array(1, 2, 3, 4);
fileStream.writeObject(array);
```

В этом примере нет никаких дополнительных действий для записи массива в файл, поскольку `Array` – один из тех типов данных, которые внутренне поддерживаются сериализацией AMF. Если вы хотите записать в файл данные пользовательского типа, нужно зарегистрировать класс с псевдонимом. Процедура регистрации класса с псевдонимом обсуждалась выше в разделе «Чтение объектов».

Теперь, когда мы научились осуществлять запись в файлы, рассмотрим более состоятельный пример, в котором используется вся полученная нами информация.

3.8. Чтение и запись списков воспроизведения музыки

В этом разделе мы воспользуемся всеми полученными в этой главе знаниями, чтобы создать простое приложение, которое позволяет создавать локальные списки воспроизведения mp3-файлов. При этом приложение не будет воспроизводить сами файлы (хотя это нетрудно сделать), поскольку сейчас нас в большей мере интересуют операции с файловой системой. Никто не мешает вам самостоятельно решить задачу дополнения программы функциями воспроизведения mp3.

Приложение для составления списка достаточно элементарно. Оно состоит всего из четырех классов/документов MXML:

- PlaylistMaker.mxml
- ApplicationData.as
- Playlist.as
- PlaylistService.as

В последующих разделах мы построим каждый из них. То, что должно получиться в результате, представлено на рис. 3.11.

В приложении PlaylistMaker реализованы следующие функции:

- Поиск всех файлов mp3 на машине пользователя (начиная с заданного каталога) и вывод их списка
- Добавление и удаление записей в списке
- Сохранение списков воспроизведения
- Загрузка ранее сохраненных списков

Теперь, когда мы знаем, какова структура приложения, как оно выглядит и какие функции выполняет, можем заняться его построением.

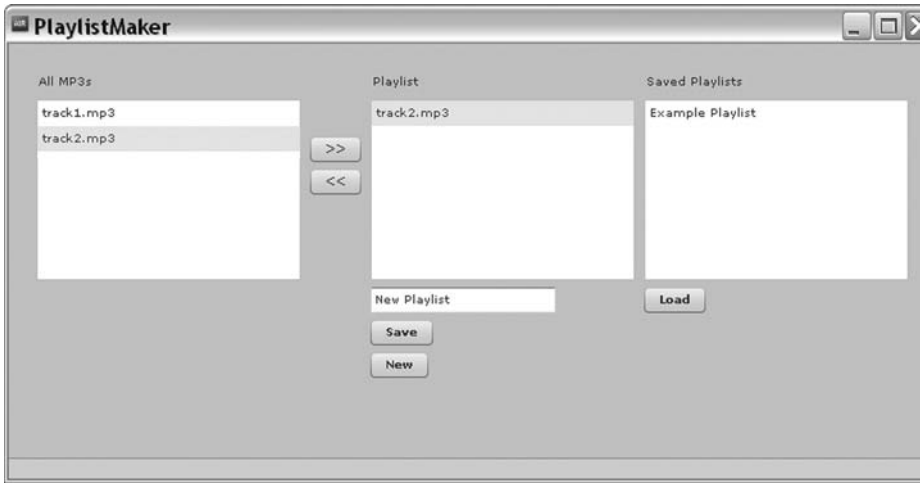


Рис. 3.11. *Приложение PlaylistMaker позволяет создавать списки воспроизведения файлов mp3, находящихся на компьютере*

Прежде всего нужно сконфигурировать проект приложения PlaylistMaker. Если вы работаете с Flex Builder, создайте новый проект AIR с именем PlaylistMaker, и вся необходимая файловая структура, а также файл приложения PlaylistMaker.mxml, будут созданы автоматически. Если вы не пользуетесь Flex Builder, начните конфигурировать проект как обычно и назовите главный файл приложения PlaylistMaker.mxml.

После конфигурирования проекта можно перейти к созданию модели данных и модели локатора для приложения. Этому посвящен следующий раздел.

3.8.1. Создание модели данных

Это достаточно простое приложение, поэтому и модель данных у него простая. Фактически, нам нужен один класс модели данных: Playlist. Объектам Playlist нужны, в сущности, два вида информации: название и коллекция саундтреков, входящих в список воспроизведения. Класс Playlist действительно прост, как видно из листинга 3.16.

Помимо класса Playlist, нам также понадобится создать нечто, выполняющее роль локатора, что позволит централизованно хранить данные приложения. С этой целью мы воспользуемся классом, который назовем `ApplicationData`. В классе `ApplicationData` содержатся данные, используемые тремя частями приложения: все mp3, текущий список воспроизведения и все записанные списки воспроизведения.

Чтобы построить модель данных и локатор модели, выполните следующие шаги:

1. Создайте новый документ класса ActionScript и сохраните его как com/manning/playlistmaker/data/Playlist.as относительно каталога исходных файлов проекта.
2. Поместите в класс `Playlist` код из листинга 3.16.

Листинг 3.16. Класс Playlist

```
package com.manning.playlistmaker.data {
   import flash.events.Event;
   import flash.events.EventDispatcher;
   import flash.filesystem.File;

   [RemoteClass(alias="com.manning.playlistmaker.data.Playlist")]
   public class Playlist extends EventDispatcher {

      private var _list:Array;
      private var _name:String;

      [Bindable(event="listChanged")]
      public function get list():Array {
      return _list;
   }

   public function set list(value:Array):void {
      _list = value;
   }

   [Bindable(event="nameChanged")]
   public function set name(value:String):void {
      _name = value;
   }

   public function get name():String {
      return _name;
   }

   public function Playlist() {
      if(_list == null) {
         _list = new Array();
      }
   }

   public function addTrack(value:File):void {
      _list.push(value);
      dispatchEvent(new Event("listChanged"));
   }

   public function addTracks(value:Array):void {
      _list = _list.concat(value);
      dispatchEvent(new Event("listChanged"));
   }

   public function removeTrack(value:File):void {
      for(var i:Number = 0; i < _list.length; i++) {
         if(_list[i].nativePath == value.nativePath) {
            _list.splice(i, 1);
```

Сделать класс сериализуемым ❶

Записать треки списка

Имя списка

❷ Сделать свойства связываемыми

Создать массив, только если null

❸ Добавить трек в список

Добавить только название

❹ Уведомить связанные свойства

❺ Добавить несколько треков

❻ Удалить трек

```
                break;
            }
        }
        dispatchEvent(new Event("listChanged"));
    }

  }
}
```

Несмотря на свою простоту, класс `Playlist` содержит некоторые детали, требующие обсуждения. Во-первых, обратите внимание, что класс начинается с метатега `[RemoteClass]` ❶. Это нужно для того, чтобы при записи объектов `Playlist` на диск иметь возможность сериализовать и десериализовать объекты. Если помните, тег метаданных `[RemoteClass]` сообщает приложению, как отобразить сериализованные данные снова в класс. Обратите также внимание, что свойства `name` и `list` могут привязываться к данным ❷. Это связано с тем, что мы хотим потом подключить компоненты интерфейса, чтобы вывести содержимое списка воспроизведения. Поскольку мы хотим иметь возможность связывания данных свойства `list`, массива, то нужно добавить методы доступа для добавления и удаления треков ❸ ❺ ❻. Как можно видеть, все эти методы посылают событие `listChanged` ❹, которое вызывает изменение связанных данных.

3. Создайте новый класс документа ActionScript и сохраните его как com/manning/playlistmaker/data/ApplicationData.as относительно каталога исходных текстов проекта.
4. Поместите в класс `ApplicationData` код из листинга 3.17.

Листинг 3.17. Класс ApplicationData

```
package com.manning.playlistmaker.data {

    import flash.events.Event;
    import flash.events.EventDispatcher;
    import flash.filesystem.File;

    public class ApplicationData extends EventDispatcher {

        static private var _instance:ApplicationData;

        private var _mp3s:Playlist;
        private var _playlist:Playlist;
        private var _savedPlaylists:Array;

        [Bindable(event="mp3sChanged")]
        public function get mp3s():Playlist {
            return _mp3s;
        }

        [Bindable(event="playlistChanged")]
        public function set playlist(value:Playlist):void {
            _playlist = value;
            dispatchEvent(new Event("playlistChanged"));
```

❶ Экземпляр Singleton

❷ Все mp3-системы

❸ Текущий список

❹ Сохраненные списки

```
        }

        public function get playlist():Playlist {
           return _playlist;
        }

        [Bindable(event="savedPlaylistsChanged")]
        public function set savedPlaylists(value:Array):void {
            _savedPlaylists = value;
            dispatchEvent(new Event("savedPlaylistsChanged"));
        }

        public function get savedPlaylists():Array {
           return _savedPlaylists;
        }

        public function ApplicationData() {
           _mp3s = new Playlist();
           _playlist = new Playlist();
           _savedPlaylists = new Array();
        }

        static public function getInstance():ApplicationData {
           if(_instance == null) {
              _instance = new ApplicationData();
           }
           return _instance;
        }

        public function addPlaylist(playlist:Playlist):void {
           if(_savedPlaylists.indexOf(playlist) == -1) {
              _savedPlaylists.push(playlist);
              dispatchEvent(new Event("savedPlaylistsChanged"));
           }
        }

     }
  }
```

Метод доступа к Singleton ❺

Добавить список к сохраненным ❻

Класс `ApplicationData` – простой. Как видите, он реализует шаблон проектирования Singleton ❶ ❺. Класс предоставляет доступ к информации трех видов: массиву всех mp3 в системе ❷, текущему списку воспроизведения ❸ и массиву сохраненных списков воспроизведения ❹. Дополнительно в классе определен метод для добавления списка воспроизведения к уже сохраненным ❻. Прежде чем добавить список, метод проверяет, нет ли его среди уже сохраненных.

Это все, что касается модели данных и локатора для данного приложения. Далее приступим к созданию сервиса/контроллера для приложения.

3.8.2. Создание контроллера

Контроллером для этого приложения PlaylistMaker будет класс, который мы назовем `PlaylistService`. На него возложены две главные

функции: получение списка mp3, имеющихся в системе, с сохранением списков на диске и получение их с диска. Чтобы создать контроллер, выполните следующие действия:

1. Создайте новый документ класса ActionScript и сохраните его как com/manning/playlistmaker/data/PlaylistService.as относительно каталога исходных файлов проекта.
2. Поместите в класс PlaylistService следующий код. Этот код образует структуру класса. Мы наполним методы содержимым на следующих этапах.

```
package com.manning.playlistmaker.services {
   import flash.filesystem.File;

   public class PlaylistService {

      public function PlaylistService() {
      }

      public function getMp3s(parentDirectory:File):void {
      }

     private function locateMp3sInDirectory(parentDirectory:File):void {
      }

      private function directoryListingHandler(event:FileListEvent):
      ➥void  {
      }

      public function savePlaylists():void {
      }

      public function loadSavedPlaylists():void {
      }

   }
}
```

3. Напишите код методов getMp3s(), locateMp3sInDirectory() и directoryListingHandler(). В совокупности они позволяют получить все файлы mp3 из родительского каталога.

```
package com.manning.playlistmaker.services {
   import flash.filesystem.File;
   import flash.events.FileListEvent;
   import com.manning.playlistmaker.data.ApplicationData;

   public class PlaylistService {

      public function PlaylistService() {
      }

      public function getMp3s(parentDirectory:File):void {

         locateMp3sInDirectory(parentDirectory);
      }
```

❶ Получить файлы mp3

```
        private function locateMp3sInDirectory(parentDirectory:File):void {
            parentDirectory.addEventListener(
            ➥FileListEvent.DIRECTORY_LISTING, directoryListingHandler);
            parentDirectory.getDirectoryListingAsync();
        }

        private function directoryListingHandler(event:FileListEvent):
        ➥void {
            var files:Array = event.files;
            var mp3s:Array = new Array();
            var file:File;
            for(var i:Number = 0; i < files.length; i++) {
                file = files[i] as File;
                if(file.isDirectory) {
                    locateMp3sInDirectory(file);
                }
                else if(file.extension == "mp3") {
                    mp3s.push(file);
                }
            }
            if(mp3s.length > 0) {
                ApplicationData.getInstance().mp3s.addTracks(mp3s);
            }
        }

        public function savePlaylists():void {
        }

        public function loadSavedPlaylists():void {
        }

    }
}
```

❷ Получить содержимое каталога

❸ Просмотреть содержимое

❹ Выполнить нужные действия

❺ Добавить в модель данных

Метод `getMp3s()` просто вызывает закрытый метод `locateMp3sInDirectory()` ❶, который асинхронно получает содержимое каталога ❷. Когда список содержимого готов, обработчик последовательно перебирает все содержимое ❸ и выполняет для каждого объекта должное действие ❹. Если элемент является каталогом, рекурсивно применяем к нему `locateMp3sInDirectory()`. В противном случае, если файл имеет расширение .mp3, добавляем его в массив, который позднее добавляем к модели данных ❺.

4. Напишите код метода `savePlaylists()`, как показано ниже:

```
package com.manning.playlistmaker.services {
    import flash.filesystem.File;
    import flash.events.FileListEvent;
    import com.manning.playlistmaker.data.ApplicationData;
    import flash.filesystem.FileMode;
    import flash.filesystem.FileStream;
    import com.manning.playlistmaker.data.Playlist;

    public class PlaylistService {
```

```
        public function PlaylistService() {
        }

        public function getMp3s(parentDirectory:File):void {
            locateMp3sInDirectory(parentDirectory);
        }

        private function locateMp3sInDirectory(parentDirectory:File):void {
            parentDirectory.addEventListener(
            ➥FileListEvent.DIRECTORY_LISTING, directoryListingHandler);
            parentDirectory.getDirectoryListingAsync();
        }

        private function directoryListingHandler(event:FileListEvent):
       ➥void   {
            var files:Array = event.files;
            var mp3s:Array = new Array();
            var file:File;
            for(var i:Number = 0; i < files.length; i++) {
                file = files[i] as File;
                if(file.isDirectory) {
                    locateMp3sInDirectory(file);
                }
                else if(file.extension == "mp3") {
                    mp3s.push(file);
                }
            }
            if(mp3s.length > 0) {
                ApplicationData.getInstance().mp3s.addTracks(mp3s);
            }
        }

        public function savePlaylists():void {
            var file:File =
           ➥File.applicationStorageDirectory.resolvePath(
           ➥"savedPlaylists.data");
            var stream:FileStream = new FileStream();
            stream.open(file, FileMode.WRITE);
            var applicationData:ApplicationData =
           ➥ApplicationData.getInstance();
            applicationData.addPlaylist(applicationData.playlist);
            stream.writeObject(applicationData.savedPlaylists);
            stream.close();
        }

        public function loadSavedPlaylists():void {
        }

    }
}
```


При сохранении данных мы создаем ссылку на файл ❶, открываем его в режиме для записи ❷ и пишем данные в файл ❹. В данном случае мы пишем все списки в файл. Кроме того, нужно добавить текущий список

воспроизведения в массив сохраненных списков в модели данных ❸, прежде чем осуществлять запись на диск.

5. Напишите код метода, который загружает сохраненные списки. Листинг 3.18 демонстрирует полный класс вместе с этим методом.

Листинг 3.18. Класс PlaylistService

```
package com.manning.playlistmaker.services {
   import flash.filesystem.File;
   import flash.events.FileListEvent;
   import com.manning.playlistmaker.data.ApplicationData;
   import flash.filesystem.FileMode;
   import flash.filesystem.FileStream;
   import com.manning.playlistmaker.data.Playlist;

   public class PlaylistService {

      public function PlaylistService() {
      }

      public function getMp3s(parentDirectory:File):void {
         locateMp3sInDirectory(parentDirectory);
      }

      private function locateMp3sInDirectory(parentDirectory:File):void {
         parentDirectory.addEventListener(
         ➥FileListEvent.DIRECTORY_LISTING, directoryListingHandler);
         parentDirectory.getDirectoryListingAsync();
      }

      private function directoryListingHandler(event:FileListEvent):
      ➥void    {
         var files:Array = event.files;
         var mp3s:Array = new Array();
         var file:File;
         for(var i:Number = 0; i < files.length; i++) {
            file = files[i] as File;
            if(file.isDirectory) {
               locateMp3sInDirectory(file);
            }
            else if(file.extension == "mp3") {
               mp3s.push(file);
            }
         }
         if(mp3s.length > 0) {
            ApplicationData.getInstance().mp3s.addTracks(mp3s);
         }
      }

      public function savePlaylists():void {
         var file:File =

➥File.applicationStorageDirectory.resolvePath("savedPlaylists.data");
         var stream:FileStream = new FileStream();
```

```
            stream.open(file, FileMode.WRITE);
            var applicationData:ApplicationData =
            ➥ApplicationData.getInstance();
            applicationData.addPlaylist(applicationData.playlist);
            stream.writeObject(applicationData.savedPlaylists);
            stream.close();
        }

        public function loadSavedPlaylists():void {
            var file:File =
            ➥File.applicationStorageDirectory.resolvePath(
            ➥"savedPlaylists.data");
            if(file.exists) {
                var stream:FileStream = new FileStream();
                stream.open(file, FileMode.READ);
                var playlists:Array = new Array();
                while(stream.bytesAvailable) {
                    playlists = stream.readObject() as Array;
                }
                stream.close();

                ApplicationData.getInstance().savedPlaylists = playlists;
            }
        }

    }
}
```

❶ Ссылка на файл хранилища

❷ Если файл существует

❸ Открыть для чтения

❹ Читать данные из файла

❺ Присвоить массив savedPlaylists

Читая сохраненные списки, мы читаем из того же файла, в который писали данные ❶. Прежде чем читать данные из файла, проверим, что он существует ❷, иначе возникнет ошибка. Затем откроем файл в режиме для чтения ❸, прочтем данные ❹ и запишем данные в модель данных ❺.

На этом контроллер завершен. Остается только создать для приложения интерфейс пользователя.

3.8.3. Создание интерфейса пользователя

Интерфейсом пользователя для приложения PlaylistMaker является PlaylistMaker.mxml, который вы уже создали при конфигурировании проекта. Осталось добавить в этот документ необходимый код, чтобы он выглядел, как на рис. 3.11. Для этого откройте PlaylistMaker.mxml и добавьте туда код из листинга 3.19.

Листинг 3.19. Документ PlaylistMaker

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="creationCompleteHandler();" width="800">
    <mx:Script>
    <![CDATA[
        import com.manning.playlistmaker.data.Playlist;
```

Зарегистрировать обработчик creationComplete

```
import com.manning.playlistmaker.services.PlaylistService;
import com.manning.playlistmaker.data.ApplicationData;
import flash.filesystem.File;

[Bindable]
private var _applicationData:ApplicationData;
private var _service:PlaylistService;

private function creationCompleteHandler():void {
   _applicationData = ApplicationData.getInstance();
   _service = new PlaylistService();
   _service.getMp3s(File.documentsDirectory);
   _service.loadSavedPlaylists();
}

private function addToPlaylist():void {
   _applicationData.playlist.addTrack(
   ➥mp3list.selectedItem as File);
}

private function removeFromPlaylist():void {
   _applicationData.playlist.removeTrack(
   ➥playlist.selectedItem as File);
}

private function newPlaylist():void {
   _applicationData.playlist = new Playlist();
}

private function loadPlaylist():void {
   _applicationData.playlist =
   ➥savedPlaylists.selectedItem as Playlist;
}

private function savePlaylist():void {
   var playlist:Playlist = _applicationData.playlist;
   playlist.name = playlistName.text;
   _service.savePlaylists();
}
]]>
</mx:Script>
<mx:HBox width="100%">
   <mx:VBox width="33%">
      <mx:Label text="All MP3s" />
      <mx:List id="mp3list"
             dataProvider="{_applicationData.mp3s.list}"
             labelField="name" width="100%" height="100%" />
   </mx:VBox>
   <mx:VBox>
      <mx:Spacer height="50" />
      <mx:Button label="&gt;&gt;" click="addToPlaylist();"
         enabled="{mp3list.selectedItem != null}" />
      <mx:Button label="&lt;&lt;" click="removeFromPlaylist();"
```

Получить ссылку на ApplicationData

Создать экземпляр сервиса

❶ Запросить mp3-системы

❷ Сохранить списки воспроизведения

Задать имя списка

Сохранить списки

```
            enabled="{playlist.selectedItem != null}" />
      </mx:VBox>
      <mx:VBox width="33%">
         <mx:Label text="Playlist" />
         <mx:List id="playlist"
            dataProvider="{_applicationData.playlist.list}"
            labelField="name"
            width="100%" height="100%" />
         <mx:TextInput id="playlistName"
   text="{_applicationData.playlist.name}" />
         <mx:Button label="Save" click="savePlaylist();" />
         <mx:Button label="New" click="newPlaylist();" />
      </mx:VBox>
      <mx:VBox width="33%">
         <mx:Label text="Saved Playlists" />
         <mx:List id="savedPlaylists"
            dataProvider="{_applicationData.savedPlaylists}"
            labelField="name" width="100%" height="100%" />
         <mx:Button label="Load" click="loadPlaylist();" />
      </mx:VBox>
   </mx:HBox>
</mx:WindowedApplication>
```

В документе PlaylistMaker.mxml нет ничего необычного. В основном он состоит из набора компонент UI, подключенных (через привязку к источникам данных) к свойствам ApplicationData. Помимо этого он просто делает вначале несколько запросов к сервису ❶ ❷ и отвечает на действия пользователя путем обновления значений в `ApplicationData`. Обратите внимание, что при запросе mp3 у системы мы передаем сервису родительский каталог документов пользователя ❶. Это означает, что файлы mp3 будут браться только из каталога документов. Если вы хотите взять эти файлы из других мест, задайте другой начальный каталог.

Это все, что касается PlaylistMaker. Можете запустить его и посмотреть, как он работает.

Может сложиться впечатление, что едва ли остались для обсуждения вопросы, касающиеся файлов и локального хранения данных. Мы уже и так изложили большой объем сведений. Не беспокойтесь. Материала по этой теме осталось немного, но есть еще один важный вопрос: безопасность хранения данных. К нему мы теперь и перейдем.

3.9. Безопасное хранение данных

Мы рассмотрели чтение и запись данных с помощью файлов. Эти данные часто не только легко доступны, но и могут читаться человеком. Для большей части данных беспокоиться тут не о чем. В предыдущем разделе мы записали списки воспроизведения в файлы, и ничто не помешало сохранить эти данные в легкодоступных файлах, что означает возможность открыть и прочесть их любому пользователю. Вероятно,

вас не обеспокоит, если составленные вами списки смогут прочесть ваша сестра, ребенок или даже посторонний человек. Но не любые данные допускают такое отношение. Рассмотрим следующий пример. Ради удобства вы решили дать приложению возможность сохранить данные о счетах пользователя. В результате ему не придется при каждой новой покупке вводить одни и те же данные. Поначалу идея записать эти данные в файл может показаться неплохой, но, в отличие от списка mp3-записей, финансовые данные вряд ли окажутся той информацией, которой пользователь готов поделиться с другими. Хранить ее в обычных файлах было бы рискованно. Информация становится доступной не только другим пользователям компьютера, но и другим программам, которые на нем выполняются.

Запись в обычные файлы – не самая лучшая идея в таких ситуациях, как описанная, поэтому AIR предлагает решение для этой проблемы. Класс `flash.data.EncryptedLocalStore` обеспечивает доступ к находящейся на компьютере защищенной области хранения данных. Каждый пользователь компьютера получает собственную защищенную область для каждого приложения AIR. Это означает, что если вы совершаете покупки с помощью своего приложения, то вам предоставляется область защищенного хранения данных, но если на том же компьютере тем же приложением пользуется ваша сестра, то у нее будет своя отдельная область защищенного хранения данных. Класс `EncryptedLocalStore` незаметно от вас позаботится обо всей логике, определив, какие области хранения должны использоваться. Вам достаточно только написать код, который запишет данные в хранилище, прочтет их оттуда или удалит.

Все данные, записываемые в область защищенного хранения данных с помощью `EncryptedLocalStore`, зашифрованы посредством 128-разрядного алгоритма AES-CBC. Все заботы о шифровании и дешифровании берет на себя `EncryptedLocalStore`. Вам нужно только вызвать нужные методы и передать им нужные параметры. Методы мы рассмотрим несколько ниже, а сначала взглянем на то, как хранятся данные.

Каждый фрагмент данных в зашифрованном хранилище идентифицируется уникальным ключом. Ключ представляет собой строку, с помощью которой можно извлечь данные. Например, если вы хотите записать пароль для почтового сервера, можно воспользоваться таким ключом, как `emailServerPassword`. Ключи выбираются произвольно, но обычно разумно пользоваться именами, по которым ясно назначение. Каждый ключ указывает на фрагмент данных, хранящихся с помощью `EncryptedLocalStore`, и этот фрагмент хранится в виде объекта `flash.utils.ByteArray`. Если вам не знаком класс `ByteArray`, не пугайтесь. Он реализует интерфейсы `IDataInput` и `IDataOutput` – те же, которые реализованы в `FileStream`. Это означает, что операции чтения и записи для объекта `ByteArray` можно проводить так же, как для объекта `FileStream`. Например, следующий код создает объект `ByteArray` и записывает в него массив строк с помощью метода `writeObject()`:

```
var array:Array = new Array("a", "b", "c", "d");
var byteArray:ByteArray = new ByteArray();
byteArray.writeObject(array);
```

Если вам нужно записать данные в хранилище, достаточно вызвать статический метод `EncryptedLocalStore.setItem()`. Статический метод `setItem()` требует двух параметров: ключа и данных (в виде объекта `ByteArray`). В следующем примере записывается значение пароля с помощью `setItem()`:

```
var byteArray:ByteArray = new ByteArray();
byteArray.writeUTFBytes("j8ml08*1");
EncryptedLocalStore.setItem("emailServerPassword", byteArray);
```

Записав данные в хранилище, вы, вероятно, в какой-то момент захотите получить их обратно. Для этого можно применить метод `getItem()`. Метод `getItem()` требует, чтобы вы указали ключ для данных, которые хотите получить. После этого он возвращает объект `ByteArray` с данными. В следующем примере извлекается пароль почтового сервера:

```
var byteArray:ByteArray =
➥EncryptedLocalStore.getItem("emailServerPassword");
var password:String= byteArray.readUTFBytes(byteArray.length);
```

Как быть, если нужно удалить данные из хранилища? Нет проблем. `EncryptedLocalStore` предоставляет для этого два статических метода: `removeItem()` и `reset()`. Метод `removeItem()` удаляет объект, определяемый ключом. Например, следующий код удаляет из хранилища пароль почтового сервера:

```
EncryptedLocalStore.removeItem("emailServerPassword");
```

Метод `reset()` удаляет из хранилища все данные:

```
EncryptedLocalStore.reset();
```

Мы собрали все основные теоретические сведения, поэтому нам осталось в этой главе сделать только одно – дополнить наше приложение AirTube, применив к нему часть полученных знаний о работе с файловой системой.

3.10. Запись в файлы в AirTube

Если помните, в главе 2 мы начали строить приложение AirTube, которое дает возможность поиска в YouTube и воспроизведения фильмов. В той главе мы значительно преуспели в создании этого приложения. Но нам еще предстоит реализовать одну из главных функций приложения: загрузку фильмов для воспроизведения в автономном режиме. Мы неспроста не создали эту функцию во второй главе: мы еще не знали, как это сделать. Но с нашими нынешними знаниями мы готовы взяться за эту работу.

Несмотря на знакомство с теорией, мы еще не видели ни одного практического примера. До сих пор в этой главе мы рассматривали практические случаи записи и чтения локальных файлов, но не было примера чтения интернет-ресурса и записи его в локальный файл. Этим мы сейчас и займемся. Нам нужно загрузить из Интернета файл .flv и сохранить его в локальном файле на компьютере пользователя. С помощью той же процедуры мы загрузим миниатюрное изображение для фильма.

Чтобы реализовать в приложении AirTube эту новую функцию, выполните следующие шаги:

1. Откройте класс `ApplicationData` для проекта AirTube и измените код, добавив в него свойство `downloadProgress`, как в листинге 3.20. (Изменения показаны жирным шрифтом.) С помощью этого свойства мы будем следить за ходом загрузки фильма. Само по себе оно делает немного. Но мы будем обновлять его значение из класса сервиса, как будет показано в ближайшем будущем.

Листинг 3.20. Добавление свойства downloadProgress в класс ApplicationData

```
package com.manning.airtube.data {

   import flash.events.Event;
   import flash.events.EventDispatcher;

   public class ApplicationData extends EventDispatcher {

       static private var _instance:ApplicationData;

       private var _videos:Array;
       private var _currentVideo:AirTubeVideo;
       private var _downloadProgress:Number;

        [Bindable(event="videosChanged")]
       public function set videos(value:Array):void {
           _videos = value;
           dispatchEvent(new Event("videosChanged"));
       }

       public function get videos():Array {
           return _videos;
       }

        [Bindable(event="currentVideoChanged")]
       public function set currentVideo(value:AirTubeVideo):void {
           _currentVideo = value;
           dispatchEvent(new Event("currentVideoChanged"));
       }

       public function get currentVideo():AirTubeVideo {
           return _currentVideo;
       }

        [Bindable(event="downloadProgressChanged")]
```

```
        public function set downloadProgress(value:Number):void {
            _downloadProgress = value;
            dispatchEvent(new Event("downloadProgressChanged"));
        }

        public function get downloadProgress():Number {
            return _downloadProgress;
        }

        public function ApplicationData() {
        }

        static public function getInstance():ApplicationData {
       if(_instance == null) {
          _instance = new ApplicationData();
       }
       return _instance;
     }

   }
}
```

2. Откройте класс `AirTubeService` и добавьте в него код из листинга 3.21. Изменения выделены жирным шрифтом. Мы добавляем открытый метод `saveToOffline()`, который инициирует загрузку файлов фильмов и миниатюр, а затем добавляем необходимые методы-обработчики.

Листинг 3.21. Добавление saveToOffline( ) в класс AirTubeService

```
package com.manning.airtube.services {

   import   com.adobe.webapis.youtube.YouTubeService;
   import   com.adobe.webapis.youtube.events.YouTubeServiceEvent;
   import   com.manning.airtube.data.AirTubeVideo;
   import   com.manning.airtube.data.ApplicationData;
   import   com.manning.airtube.utilities.YouTubeFlvUrlRetriever;

   import   flash.events.Event;
   import   flash.events.ProgressEvent;
   import   flash.filesystem.File;
   import   flash.filesystem.FileMode;
   import   flash.filesystem.FileStream;
   import   flash.net.URLRequest;
   import   flash.net.URLStream;
   import   flash.utils.ByteArray;

   public class AirTubeService {

      static private var _instance:AirTubeService;

      private  var _proxied:YouTubeService;
      private  var _flvFile:File;
      private  var _imageFile:File;
      private  var _downloadingVideo:AirTubeVideo;
```

```
public function set key(value:String):void {
   _proxied.apiKey = value;
}

public function AirTubeService() {
   _proxied = new YouTubeService();
   _proxied.addEventListener(
   ➥YouTubeServiceEvent.VIDEOS_LIST_BY_TAG,
   ➥getVideosByTagsResultHandler);
}

static public function getInstance():AirTubeService {
   if(_instance == null) {
       _instance = new AirTubeService();
   }
   return _instance;
}

public function getVideosByTags(tags:String):void {
   if(_proxied.apiKey.length == 0) {
      throw Error("YouTube API key not set");
   }
   _proxied.videos.listByTag(tags);
}

private function getVideosByTagsResultHandler(
➥event:YouTubeServiceEvent):void    {
   var videos:Array = event.data.videoList as Array;
   for(var i:Number = 0; i < videos.length; i++) {
      videos[i] = new AirTubeVideo(videos[i]);
   }
   ApplicationData.getInstance().videos = videos;
}

public function configureVideoForPlayback(video:AirTubeVideo):
➥void  {
   ApplicationData.getInstance().currentVideo = video;
   if(video.flvUrl == null) {
      new YouTubeFlvUrlRetriever().getUrl(video);
   }
}

public function saveToOffline(video:AirTubeVideo):void {
   _downloadingVideo = video;

   _flvFile = File.applicationStorageDirectory.resolvePath(
   ➥"videos/" + video.video.id + ".flv");
   var videoLoader:URLStream = new URLStream();
   videoLoader.load(new URLRequest(video.flvUrl));
   videoLoader.addEventListener(Event.COMPLETE,
                  videoDownloadCompleteHandler);
   videoLoader.addEventListener(ProgressEvent.PROGRESS,
   ➥videoDownloadProgressHandler);
```

❶ Переход к сохраняемому видео

❷ Создать путь для сохранения видео

```
            _imageFile = File.applicationStorageDirectory.resolvePath(
            ➥"thumbnails/" + video.video.id + ".jpg");
            var imageLoader:URLStream = new URLStream();
            imageLoader.load(new URLRequest(video.video.thumbnailUrl));
            imageLoader.addEventListener(ProgressEvent.PROGRESS,
            ➥imageDownloadProgressHandler);
        }

        private function videoDownloadProgressHandler(event:
        ➥ProgressEvent):void {
            var loader:URLStream = event.target as URLStream;
            var bytes:ByteArray = new ByteArray();
            loader.readBytes(bytes);
            var writer:FileStream = new FileStream();
            writer.open(_flvFile, FileMode.APPEND);
            writer.writeBytes(bytes);
            writer.close();
            var ratio:Number = event.bytesLoaded / event.bytesTotal;
            ApplicationData.getInstance().downloadProgress = ratio;
        }

        private function videoDownloadCompleteHandler(event:Event):void {
            _downloadingVideo.offline = true;
            ApplicationData.getInstance().downloadProgress = 0;
        }

        private function imageDownloadProgressHandler(event:
        ➥ProgressEvent):void  {
            var loader:URLStream = event.target as URLStream;
            var bytes:ByteArray = new ByteArray();
            loader.readBytes(bytes);
            var writer:FileStream = new FileStream();
            writer.open(_imageFile, FileMode.APPEND);
            writer.writeBytes(bytes);
            writer.close();
         }
    }
}
```

❸ Создать путь для сохранения миниатюры

❹ Прочитать доступные байты

❺ Записать в файл видео

❻ Прочитать доступные байты

❼ Записать в файл миниатюры

Метод `saveToOffline()` ❶ с помощью объектов `URLStream` начинает загрузку видеофайлов и миниатюр и создает пути для сохраняемых файлов ❷ ❸, используя ID видео для создания уникальных имен файлов. В процессе загрузки видео и миниатюр события progress обрабатываются методами `videoDownloadProgressHandler()` и `imageDownloadProgressHandler()` соответственно. Каждый из этих методов делает по существу одно и то же: с помощью метода `readBytes()` объекта `URLStream` читает все доступные байты ❹ ❻ и записывает их в конец целевого файла ❺ ❼.

3. Внесите в окно видео незначительные изменения. Для этого откройте VideoWindow.mxml и добавьте код, выделенный жирным шрифтом в листинге 3.22.

Листинг 3.22. Модификация VideoWindow.mxml to для поддержки загрузки видео

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" width="400"
 height="400" type="utility" closing="closingHandler(event);"
 creationComplete="creationCompleteHandler();">
   <mx:Script>
       <![CDATA[
          import com.manning.airtube.services.AirTubeService;
          import com.manning.airtube.data.ApplicationData;

           [Bindable]
          private var _applicationData:ApplicationData;

          private function creationCompleteHandler():void {
              _applicationData = ApplicationData.getInstance();
          }

          private function closingHandler(event:Event):void {
              event.preventDefault();
              visible = false;
          }

          private function saveOffline():void {
                 AirTubeService.getInstance().saveToOffline(
                 ➥_applicationData.currentVideo);
          }

       ]]>
   </mx:Script>
   <mx:VBox>
      <mx:Label text="{_applicationData.currentVideo.video.title}" />
      <mx:VideoDisplay id="videoDisplay"
         width="400" height="300" />
      <mx:HBox id="progressContainer" width="100%"
           visible="{_applicationData.downloadProgress > 0}"
           includeInLayout="{progressContainer.visible}">#1
         <mx:Label text="download progress" />
         <mx:HSlider id="progressIndicator" enabled="false" width="100%"
                minimum="0" maximum="1"
                value="{_applicationData.downloadProgress}" />
      </mx:HBox>
      <mx:HBox>
         <mx:Button id="playPauseButton" label="Pause"
            click="togglePlayback();" />
         <mx:Button id="saveOfflineButton" label="Save Offline"
                visible="{!_applicationData.currentVideo.offline}"
                enabled="{!(_applicationData.downloadProgress > 0)}"
                click="saveOffline();" />
      </mx:HBox>
   </mx:VBox>
</mx:Window>
```

Скрыть, если не идет загрузка

Показать ход загрузки ❶

Скрыть, если фильм уже загружен

Отключить во время загрузки

❷ Щелкнуть, чтобы сохранить

Изменения в коде VideoWindow.mxml достаточно ограничены. Мы лишь добавили ползунок, показывающий прогресс загрузки ❶, и кнопку для сохранения видео ❷. Компоненты привязаны к источникам данных – свойствам `ApplicationData`, и когда пользователь щелкнет, чтобы сохранить фильм, мы вызовем метод сервиса, который запишет данные.

На этом данная стадия приложения AirTube завершена. Конечно, у нас пока нет средства просмотра фильмов, которые пользователь сохранил локально. Для этого мы воспользуемся локальной базой данных, о которой будет рассказано в главе 5. Если запустить AirTube в нынешнем виде, мы увидим кнопку для сохранения фильма с целью проигрывания его в автономном режиме; щелкнув по ней, вы увидите, как меняет свой вид индикатор загрузки. Кроме того, посмотрев в каталог хранилища приложения AirTube, вы увидите в нем сохраненные файлы .flv и .jpg.

3.11. Резюме

В этой главе мы получили массу сведений о работе с файловой системой из приложений AIR. Одно из основных требований для работы с файловой системой – это умение ссылаться на файлы и каталоги, с чего мы и начали эту главу. Далее мы изучили основные методы работы с каталогами, такие как чтение их содержимого и создание новых каталогов. Затем мы рассмотрели копирование, перемещение и удаление как каталогов, так и файлов. Потом мы перешли к самой сложной теме главы: чтению из файлов и записи в них. Эта тема привела нас к вопросам, которые могли оказаться для вас неожиданными, – когда мы изучали интерфейсы для чтения и записи двоичных данных.

Мы осветили большой объем материала, и пора двинуться дальше. В главе 4 вы узнаете об операциях перетаскивания, а также копирования и вставки.

В этой главе:

- Копирование данных из приложений AIR
- Вставка данных в приложения AIR
- Перетаскивание данных из приложений AIR
- Перетаскивание данных в приложения AIR

4

Копирование и вставка. Перетаскивание

Копирование, вставка и перетаскивание – стандартные способы взаимодействия пользователей с приложениями, поэтому удобно было бы снабдить этими функциями приложения AIR. Например, если вы создаете приложение, которое показывает графические изображения, находящиеся в сети, многие пользователи положительно отнеслись бы к возможности сохранить изображение, перетащив его из приложения AIR на рабочий стол.

Важно понимать, что в приложениях AIR операции копирования и вставки, а также перетаскивания осуществляются на системном уровне. Возможно, вы работали с такими операциями в приложениях Flex или Flash, но нужно понимать существующую разницу. При создании приложений Flex или Flash для работы в сети все ваши операции перетаскивания и большинство операций копирования и вставки происходят внутри одного экземпляра Flash-плеера. Например, вы не сможете перетащить изображение из приложения Flash, выполняемого в броузере, и сохранить его на рабочем столе. AIR позволяет вывести эти операции за рамки Flash-плеера. Ваши пользователи смогут с помощью этих операций передавать данные как внутри AIR приложения, так и между ним и окружающей средой.

Операции перетаскивания и копирования/вставки просты во многих отношениях. Но эта простота не умаляет их важности. Хорошая реализация их в приложениях AIR предоставляет пользователям возможность более интуитивной и комфортной работы.

Несмотря на различие между копированием/вставкой и перетаскиванием, между этими операциями есть много общего как в теоретическом,

так и в практическом отношениях. По этой причине мы объединили обе темы в этой главе. Сначала рассмотрим общие черты этих операций. Затем более подробно изучим каждую из них.

4.1. Использование буфера обмена для передачи данных

Перетаскивание и копирование со вставкой схожи по сути. В обеих операциях происходит перемещение/копирование данных из одной области в другую. Отличие только во взаимодействии пользователя с данными, которые перемещаются/копируется.

При перетаскивании (drag-and-drop) пользователь щелкает по элементу интерфейса (например, графическому или текстовому файлу), перетаскивает его в новое место и отпускает кнопку мыши, вызывая завершение операции, при котором происходит перемещение или копирование данных (картинки или текстового файла).

Копирование и вставка требуют, чтобы пользователь выделил элемент интерфейса с помощью пункта меню, комбинации клавиш или кнопки, после чего он может перенести или скопировать элемент в новое место с помощью пункта меню, комбинации клавиш или кнопки. В обоих случаях должен существовать промежуточный носитель – место, где данные временно хранятся во время переноса или копирования. В AIR этот промежуточный носитель называется буфером (clipboard), и в последующих разделах мы более подробно изучим, что он собой представляет и как с ним работать.

4.1.1. Что такое буфер обмена?

Буфер обмена (clipboard) – метафора, которой пользуются при перемещении или копировании данных в компьютерах. Вероятно, вы знакомы с системным буфером, используемым вашей операционной системой для выполнения копирования и вставки. Например, когда вы хотите скопировать текст из своей почтовой программы и перенести его в документ Word, вы тем самым пользуетесь буфером обмена.

В этой главе мы будем иметь дело с буферами обмена двух разных типов. Один тип – это только что упомянутый системный буфер. Мы увидим, как можно записывать данные в системный буфер, чтобы использовать их с другими приложениями. Мы будем также работать с буферами обмена, специфичными для приложений AIR. В любом случае, AIR одинаково обращается с буферами обоих типов, и вы сможете получить к ним доступ через экземпляры класса `flash.desktop.Clipboard`.

Что можно делать с объектами `Clipboard`? Это относительно простые объекты. С ними можно делать три вещи: добавлять в них данные, читать эти данные и удалять данные. В данной главе мы детально разберем

различные способы выполнения этих задач, при этом, вообще говоря, не выходя за пределы перечисленных действий с объектами Clipboard.

Если вам нужен для работы объект Clipboard, то прежде всего нужно создать на него ссылку. Мы будем работать как со специфическими буферами обмена AIR, так и с системными. Поэтому нужно знать, как получать ссылки на те и другие. Ссылку на системный буфер обмена можно получить через статическое свойство Clipboard.generalClipboard.

```
var systemClipboard:Clipboard = Clipboard.generalClipboard;
```

Кроме того, мы часто будем использовать буферы обмена, поддерживаемые AIR. В таких случаях потребуется конструктор Clipboard, который не требует параметров. Вот код, создающий новый объект Clipboard:

```
var clipboard:Clipboard = new Clipboard();
```

Следующий шаг – добавление и извлечение данных. Однако сначала нужно разобраться с форматами, которые можно использовать с буфером обмена.

4.1.2. Форматы данных буфера обмена

Данные, помещаемые в буфер обмена и извлекаемые из него, могут быть различных форматов. Идея, возможно, вам знакома, даже если вы никогда об этом не задумывались. Работая с системным буфером, вы иногда копируете и вставляете текст, а иногда – копируете и вставляете файлы. Текст и файлы имеют разный формат, но в системный буфер можно записать и то, и другое. Сказанное справедливо и при работе с AIR. Отсюда следует, что вы должны уметь указать формат, в котором хотите сохранить или получить данные. Многие из методов Clipboard требуют указания формата, в том числе методы записи и чтения данных из буфера. Например, в следующем разделе мы рассмотрим метод setData(), применяемый для записи данных в буфер. Он всегда требует, чтобы были заданы формат и данные, подлежащие записи. Следующий код записывает в буфер текст, задавая при этом текстовый формат:

```
clipboard.setData(ClipboardFormats.TEXT_FORMAT, "example text");
```

Как видно из этого примера, для задания самых распространенных форматов можно пользоваться константами flash.desktop.ClipboardFormats. Эти константы и их эквиваленты в ActionScript приведены в табл. 4.1. Эти форматы воспринимаются почти всеми другими приложениями. Это означает, что можно из приложения AIR поместить данные в буфер обмена в формате растровой графики или текста и с большой вероятностью предполагать, что пользователь сможет скопировать эти данные в другое приложение на своем компьютере (например, Word). Справедливо и обратное: это форматы, которые может записывать в буфер обмена большинство приложений, и потому они будут доступны для чтения приложению AIR.

Таблица 4.1. Константы ClipboardFormats и их эквиваленты в ActionScript

Константа	Эквивалент ActionScript
BITMAP_FORMAT	BitmapData
FILE_LIST_FORMAT	Массив объектов File
HTML_FORMAT	Строка в формате HTML
TEXT_FORMAT	String
URL_FORMAT	String

Помимо стандартных форматов, перечисленных в табл. 4.1, можно передавать объекты по ссылке для использования в том же самом приложении или в сериализованном формате, который может быть понятен другим приложениям. Допустим, например, что у вас есть несколько фрагментов информации, таких как имя, адрес, город, штат/область и почтовый индекс, которые вы хотите связать вместе, создав объект пользовательского класса Address. С помощью стандартных форматов можно записать в буфер каждый отдельный элемент этой информации, но не их совокупность. Но вы можете воспользоваться своим форматом с особым именем. В данном примере в качестве имени формата подойдет address. Все, что теперь требуется, это чтобы приложение, читающее данные из буфера, знало имя формата данных (address) и умело десериализовать эти данные. Выполняя сериализацию данных, можно воспользоваться собственным методом (например, использовать строку XML) или встроенной поддержкой AMF в AIR. (Подробнее об AMF см. главу 3.) В следующем примере предполагается, что userAddress является объектом класса Address (в нашем сценарии Address – пользовательский класс), а сериализация объекта происходит автоматически с помощью AMF:

```
clipboard.setData("address", userAddress);
```

Теперь мы знаем, что такое буфер обмена и какие форматы данных можно записывать или читать из буферов. Теперь логично заняться чтением и записью данных в буфер обмена.

4.1.3. Чтение и запись данных

Буфер обмена – достаточно пассивный объект. Он не ведет активных действий. Это просто область, в которую можно записывать данные, а потом считывать их оттуда. Например, если вы хотите сделать мгновенный снимок содержимого окна AIR и обеспечить вставку его в документ Word, нужно записать данные этой картинки в буфер обмена. С другой стороны, если вы скопировали файл с рабочего стола и хотите вставить его в приложение AIR, добавив к списку файлов, нужно прочесть информацию об этом файле из буфера обмена.

Записать данные в буфер можно методом setData(), как уже было показано в предыдущем разделе. Метод setData() требует задания двух

параметров: формата и данных, которые нужно занести в буфер. В следующем примере считывается текущее значение текстового поля ввода и записывается в буфер с использованием текстового формата:

```
clipboard.setData(ClipboardFormats.TEXT_FORMAT, textInput.text);
```

Можно одновременно добавить в буфер данные в нескольких форматах. Например, если пользователь копирует графический образ из вашего приложения AIR, можно предоставить ему возможность потом вставить этот образ в графический редактор или на рабочий стол в виде графического файла. В этом случае вам потребуется поместить в буфер данные в двух форматах: `BITMAP_FORMAT` и `FILE_LIST_FORMAT`. Каждый формат требует данных своего типа, который сможет понять получатель данных. Данные в формате `BITMAP_FORMAT` должны быть объектом `BitmapData`, а для формата `FILE_LIST_FORMAT` нужен массив из одного или нескольких объектов `File`. Ниже в этой главе мы рассмотрим примеры, иллюстрирующие такого рода сценарии.

Прочесть данные из буфера обмена можно с помощью метода `getData()`. Методу `getData()` нужен лишь один параметр: формат считываемых данных. В следующем примере данные из буфера обмена извлекаются в текстовом формате:

```
var text:String = clipboard.getData(ClipboardFormats.TEXT_FORMAT)
➥as String;
```

Обратите внимание, что в этом примере значение, возвращаемое `getData()`, приводится к типу `String`. Это связано с тем, что `getData()` всегда возвращает данные произвольного типа. Если попытаться использовать это значение в контексте, требующем более конкретного типа (например, присвоить его типизированной переменной), необходимо провести соответствующее приведение типа. Разумеется, вы сами должны определить, к какому типу выполнить приведение. Для форматов текста, URL и HTML нужно осуществить приведение к типу `String`. Для формата растровой графики нужно приводить к типу `BitmapData`. А для формата списка файлов нужно приводить к типу `Array` (и знать, что каждый элемент массива представляет собой объект `File`). Для пользовательских форматов нужно знать исходный формат данных, с которым они записывались в буфер обмена.

Если вы не уверены, есть ли в буфере данные определенного формата, можно воспользоваться методом `hasFormat()`. Вы указываете этому методу формат, а он возвращает булево значение `true`, если в буфере есть данные в таком формате, и `false` – в противном случае.

4.1.4. Удаление данных из буфера обмена

Может возникнуть необходимость не только в занесении данных в буфер обмена и получении их оттуда, но также в уничтожении данных, находящихся в буфере. Например, записывая данные в системный буфер обмена в определенном формате, вы не можете гарантировать, что

в нем уже нет каких-либо других данных в других форматах. Когда пользователь запросит данные из буфера обмена, результаты окажутся непредсказуемы, поскольку у разных приложений могут быть свои представления о порядке загрузки форматов. Например, Word всегда вставляет из буфера текст, если он там есть, игнорируя все другие находящиеся там форматы. Может случиться так, что ваше приложение AIR запишет в системный буфер данные в формате растровой графики, а когда пользователь вставит данные в документ Word, они окажутся текстом, скопированным в буфер ранее другой программой. Следовательно, необходимы методы, которые удаляют данные из буфера обмена.

Примечание

Именно из-за возможности возникновения указанной ситуации следует всегда очищать системный буфер перед тем, как записывать в него данные.

Метод `clearData()` позволяет убрать из буфера обмена данные в определенном формате. Например, удалить из буфера данные в формате URL можно с помощью следующего кода:

```
clipboard.clearData(ClipboardFormats.URL_FORMAT);
```

С другой стороны, иногда нужно просто удалить все данные из буфера, независимо от формата. Тогда можно прибегнуть к методу `clear()`:

```
clipboard.clear();
```

Как видите, методу `clear()` не нужны никакие параметры. Он просто удаляет все данные.

Теперь вы знаете, как записывать, считывать и удалять данные. Но мы еще не сказали о том, как хранятся данные в буфере обмена и какое это может оказать влияние на доступность данных внутри приложения AIR и за его пределами. В следующем разделе мы обсудим, как режим передачи может повлиять на способ хранения данных.

4.1.5. Режимы передачи

Есть разные ситуации, в которых вам может понадобиться буфер обмена. Вот три главных:

- Передача данных внутри одного приложения AIR
- Передача данных между приложениями AIR
- Передача данных между приложением AIR и операционной системой или другим, не-AIR приложением

Второй и третий случаи требуют создания копии данных. Например, если вы хотите скопировать текст из приложения AIR в документ Word, вы должны помнить, что Word не в состоянии прочесть текст прямо из приложения AIR. Вместо этого нужно сделать копию текста. То же относится и к обмену между двумя приложениями AIR. Если вы создали собственный тип `Address`, который доступен в двух приложе-

ниях AIR, и хотите передать объект Address из одного приложения AIR в другое, вы должны изготовить копию объекта Address. Второе приложение AIR не может получить доступ к исходному объекту Address, чтобы прочесть его данные.

Иное дело в первом сценарии. Если вы хотите передать данные внутри одного приложения AIR, возможно обращение к данным по ссылке, а не по значению. По умолчанию при записи данных в буфер обмена они всегда сериализуются, т. е. в буфер помещается как копия данных, так и ссылка. Если вы хотите сами решать, нужно ли сериализовать данные, помещаемые в буфер, можно воспользоваться третьим, булевым параметром метода setData(). Значение true (по умолчанию) означает, что записаны будут и копия данных, и ссылка, а значение false влечет запись только ссылки. Этот параметр нужен только тогда, когда вы не хотите, чтобы данные сериализировались и копировались.

Примечание

Даже если задать в качестве третьего параметра setData() значение false, все стандартные форматы остаются доступными для не-AIR приложений. Значение false оказывает влияние только на пользовательские форматы, используемые внутри приложений AIR.

Аналогично, можно указать желаемый способ получения данных при вызове getData(). Для этого используется необязательный второй параметр, который определяет режим передачи. Существует четыре режима передачи и для каждого из них есть константа в классе flash.desktop.ClipboardTransferMode. Константы следующие:

- ORIGINAL_PREFERRED – в этом режиме возвращается ссылка на оригинал, если она существует. Если ссылки нет, возвращается копия данных. Помните, что ссылки доступны только внутри того приложения AIR, которое породило данные. Если указан данный режим передачи, а источником данных явилось другое приложение, всегда будет возвращена копия данных. Этот режим установлен по умолчанию.
- ORIGINAL_ONLY – в этом режиме возвращается ссылка на оригинал, если она существует. Если ссылки нет, возвращается null.
- CLONE_PREFERRED – в этом режиме возвращается копия данных, если она есть. В противном случае, возвращается ссылка.
- CLONE_ONLY – в этом режиме возвращается только копия данных, если она есть. В противном случае возвращается null.

Нам нужно рассмотреть еще одну тему, прежде чем браться за фактические действия с буферами обмена. Мы сейчас обсудим, чем руководствоваться, выбирая способ и время записи данных в буфер обмена.

4.1.6. Отложенный вывод

Обычно мы пишем данные в буфер сразу, как только получаем запрос. Например, если пользователь выберет в меню опцию копирования

текста из текстового поля, можно предположить, что приложение немедленно поместит данные в буфер. Однако бывают ситуации, когда лучше отложить запись реальных данных в буфер. Вместо этого полезнее сделать пометку о том, где найти необходимые данные, в ожидании момента, когда пользователь захочет извлечь эти данные из буфера. Есть два основных сценария для такой задержки:

- Объем данных велик или для записи данных в буфер требуется большой объем вычислений.
- Между тем моментом, когда данные копируются, и моментом, когда они понадобятся, данные могут измениться, а предпочтительнее иметь самые свежие данные.

Конечно, это крайние ситуации, но полезно знать, что если вам потребуется отложенная передача данных, то AIR поможет вам ее организовать. Если вам нужна отложенная пересылка, не следует применять `setData()`. Вместо этого воспользуйтесь `setDataHandler()`. Метод `setDataHandler()` тоже требует, чтобы вы задали формат. Однако вместо того чтобы передать данные, вы передаете ссылку на метод обработки. Когда пользователь запрашивает данные соответствующего формата из буфера обмена, AIR вызывает метод-обработчик и использует то значение, которое он возвратит. Из этого следует, что вы должны обеспечить соответствие типа значения, возвращаемого обработчиком, формату данных. Рассмотрим такой пример:

```
clipboard.setDataHandler(ClipboardFormats.TEXT_FORMAT, getText);
```

В этом примере задан текстовый формат, из чего следует, что метод-обработчик (`getText()`) должен возвратить `String`, например:

```
private function getText():String {
   return textArea.text;
}
```

Если вы хотите отложить передачу, воспользовавшись `setDataHandler()`, то должны следить за тем, чтобы не записать данные с помощью `setData()`, потому что у `setData()` имеется приоритет над `setDataHandler()`. Если вы уже записали данные в буфер с помощью `setData()` и хотите воспользоваться отложенной записью для того же самого формата, вызовите для этого формата `clearData()`, прежде чем вызывать `setDataHandler()`.

Метод обработки, задаваемый `setDataHandler()`, вызывается только после запроса пользователем данных (например, для вставки из буфера в документ Word). Но AIR вызовет метод обработки только один раз. При последующих запросах данных из буфера будут извлекаться все те же данные, которые были получены, когда метод был вызван впервые.

После того как мы осветили все необходимые предварительные темы, касающиеся буферов обмена, можно приступать к написанию кода, использующего буферы обмена. Затем мы обсудим операции копирования и вставки, а потом расскажем об операциях перетаскивания.

4.2. Копирование и вставка

Идея копирования и вставки, несомненно, знакома читателю. Например, чтобы создать копию какого-либо файла на своем компьютере, вы часто выбираете файл, копируете его с помощью системных комбинаций клавиш или контекстного меню, а затем вставляете копию файла в другое место файловой системы с помощью системных комбинаций клавиш или контекстного меню. Аналогичным образом вы копируете текст из почтовой программы в текстовый редактор или с веб-страницы в почтовую программу. Поскольку операции копирования и вставки являются столь фундаментальными при работе на компьютере, желательно обеспечить поддержку этой функции в создаваемых вами приложениях AIR. На протяжении нескольких последующих разделов мы рассмотрим, как организовать копирование и вставку в собственных приложениях AIR.

4.2.1. Выбор буфера обмена

Если вы хотите обеспечить поддержку копирования и вставки, почти всегда для этой цели следует использовать системный буфер обмена. Вспомните, есть два типа буферов обмена, с которыми могут работать приложения AIR: системный буфер и специальные буферы AIR. Те и другие работают одинаково, но только системный буфер обмена позволяет обмениваться через него данными с другими, не-AIR приложениями. Обычно пользователь рассчитывает, что если он что-то скопировал, то сможет вставить это в любом месте на той же машине, будь это то же самое приложение или другое. По этой причине предпочтительнее использовать системный буфер обмена.

Вспомним по нашему предшествующему обсуждению, что ссылку на системный буфер обмена можно получить с помощью свойства `Clipboard.generalClipboard`:

```
var clipboard:Clipboard = Clipboard.generalClipboard;
```

Кроме того, перед копированием данных в системный буфер нужно сначала все из него удалить. Как уже отмечалось, удалить все данные из буфера можно с помощью метода `clear()`:

```
clipboard.clear();
```

После всего этого можно копировать контент в буфер обмена.

4.2.2. Копирование контента

Для копирования контента не требуется ничего помимо записи данных в буфер с помощью ранее описанных приемов. Рассмотрим пример. Мы создадим две текстовые области, в одной из которых будет начальный текст, и дадим пользователю возможность копировать и вставлять выделенный текст, перенося его между двумя областями, пользуясь ме-

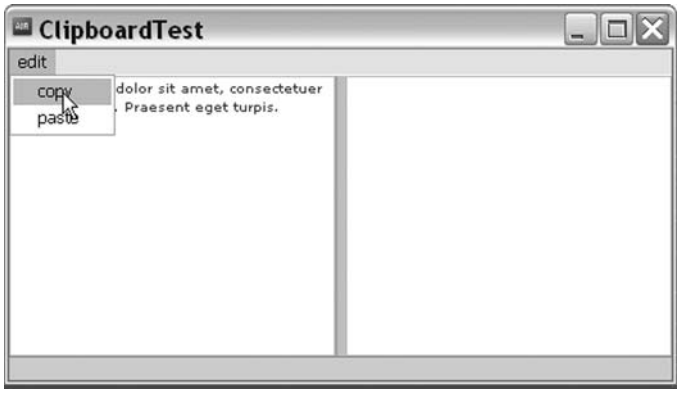


Рис. 4.1. *В этом приложении пользователь может с помощью меню выполнять копирование и вставку между двумя текстовыми областями*

ню окна или приложения. (Подробнее о меню см. главу 2.) На рис. 4.1 показан вид готового приложения.

Прежде всего, нужно создать базовую структуру приложения. Соответствующий код приведен в листинге 4.1.

Листинг 4.1. Задание базовой структуры приложения для копирования и вставки

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
       <![CDATA[

          private var _focused:TextArea;
          private var _clipboard:Clipboard;
          private function creationCompleteHandler():void {
             clipboard = Clipboard.generalClipboard;
          }

          private function copyText(event:Event):void {
          }

          private function pasteText(event:Event):void {
          }
       ]]>

    </mx:Script>
    <mx:HBox width="100%" height="100%">
       <mx:TextArea id="textAreaA" width="50%" height="100%"
          focusIn="_focused = textAreaA;">
          <mx:text>
```

Регистрация обработчика события

Выбранная текстовая область

Ссылка на буфер обмена

Назначить системный буфер

Пустой метод обработки копирования

Пустой метод обработки вставки

Поместить текстовую область в фокус

```
        Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
        </mx:text>
      </mx:TextArea>
      <mx:TextArea id="textAreaB" width="50%" height="100%"
        focusIn="_focused = textAreaB;" />
  </mx:HBox>
</mx:WindowedApplication>
```

Поместить текстовую область в фокус

Теперь можем добавить меню. Листинг 4.2 показывает код, который добавляет в меню окна или приложения подменю Edit. В меню Edit два пункта: Copy и Paste.

Листинг 4.2. Добавление меню в приложение

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
       <![CDATA[

          private var _focused:TextArea;
          private var _clipboard:Clipboard;

          private function creationCompleteHandler():void {
             _clipboard = Clipboard.generalClipboard;
             var menu:NativeMenu = new NativeMenu();
             var editMenu:NativeMenu = new NativeMenu();
             var copyItem:NativeMenuItem = new NativeMenuItem("copy");
             copyItem.addEventListener(
             ➥Event.SELECT, copyText);
             var pasteItem:NativeMenuItem =
             ➥new NativeMenuItem("paste");
             pasteItem.addEventListener(Event.SELECT, pasteText);
             editMenu.addItem(copyItem);
             editMenu.addItem(pasteItem);
             menu.addSubmenu(editMenu, "edit");
             if(NativeApplication.supportsMenu) {
                nativeApplication.menu = menu;
             }
             else if(NativeWindow.supportsMenu) {
                nativeWindow.menu = menu;
             }
          }

          private function copyText(event:Event):void {
          }

          private function pasteText(event:Event):void {
          }

       ]]>
    </mx:Script>
    <mx:HBox width="100%" height="100%">
       <mx:TextArea id="textAreaA" width="50%" height="100%"
```

Создать пункт copy

Зарегистрировать обработчик события copy

Создать пункт paste

Зарегистрировать обработчик события paste

Добавить подменю edit

Добавить меню в окно или приложение

```
            focusIn="_focused = textAreaA;">
            <mx:text>
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
            </mx:text>
         </mx:TextArea>
         <mx:TextArea id="textAreaB" width="50%" height="100%"
            focusIn="_focused = textAreaB;" />
      </mx:HBox>
   </mx:WindowedApplication>
```

Теперь нужно добавить определение метода `copyText()`**. Этот метод должен определять, какой текст выделен в выбранной текстовой области, и писать этот текст в буфер. Его код приведен в листинге 4.3.**

Листинг 4.3. Копирование выделенного текста

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
       <![CDATA[

          private var _focused:TextArea;
          private var _clipboard:Clipboard;

          private function creationCompleteHandler():void {
             _clipboard = Clipboard.generalClipboard;
             var menu:NativeMenu = new NativeMenu();
             var editMenu:NativeMenu = new NativeMenu();
             var copyItem:NativeMenuItem = new NativeMenuItem("copy");
             copyItem.addEventListener(Event.SELECT, copyText);
             var pasteItem:NativeMenuItem = new NativeMenuItem("paste");
             pasteItem.addEventListener(Event.SELECT, pasteText);
             editMenu.addItem(copyItem);
             editMenu.addItem(pasteItem);
             menu.addSubmenu(editMenu, "edit");
             if(NativeApplication.supportsMenu) {
                nativeApplication.menu = menu;
             }
             else if(NativeWindow.supportsMenu) {
                nativeWindow.menu = menu;
             }
          }

          private function copyText(event:Event):void {
            var text:String = _focused.text.substring(
            ➥_focused.selectionBeginIndex,   _focused.selectionEndIndex);
            _clipboard.clear();
            _clipboard.setData(ClipboardFormats.TEXT_FORMAT, text);
          }

          private function pasteText(event:Event):void {
             var text:String =
             _clipboard.getData(ClipboardFormats.TEXT_FORMAT) as String;
```

```
            var currentText1:String =
            ➥_focused.text.substr(0, _focused.selectionBeginIndex);
            var currentText2:String =
            ➥_focused.text.substr(_focused.selectionEndIndex);
            _focused.text = currentText1 + text + currentText2;
         }

      ]]>
   </mx:Script>
   <mx:HBox width="100%" height="100%">
      <mx:TextArea id="textAreaA" width="50%" height="100%"
         focusIn="_focused = textAreaA;">
         <mx:text>
         Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
         </mx:text>
      </mx:TextArea>
      <mx:TextArea id="textAreaB" width="50%" height="100%"
         focusIn="_focused = textAreaB;" />
   </mx:HBox>
</mx:WindowedApplication>
```

Мы завершим этот пример в следующем разделе, но уже сейчас вы можете опробовать его и убедиться, что пункт меню Edit→Copy обеспечивает копирование выделенного текста в текстовой области приложения AIR и вставку его в другое приложение, например в текстовый редактор. Это доказывает, что текст успешно копируется в системный буфер обмена.

В первом примере мы воспользовались для копирования текста меню. Вы не ограничены выбором меню в качестве способа запуска и не ограничены выбором для копирования именно текста. Сейчас мы рассмотрим простой пример, в котором с помощью кнопки выполняется мгновенный снимок приложения AIR в виде растрового изображения, которое записывается в системный буфер. Рисунок 4.2 показывает, как выглядит работающее приложение.

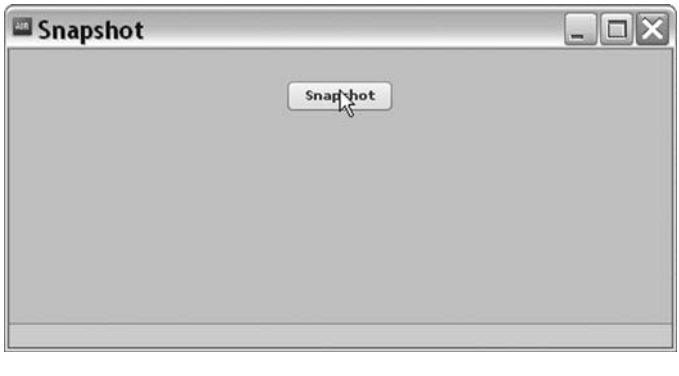


Рис. 4.2. *Выполните мгновенный снимок приложения путем нажатия на кнопку, после чего можно вставить снимок из буфера в другое приложение*

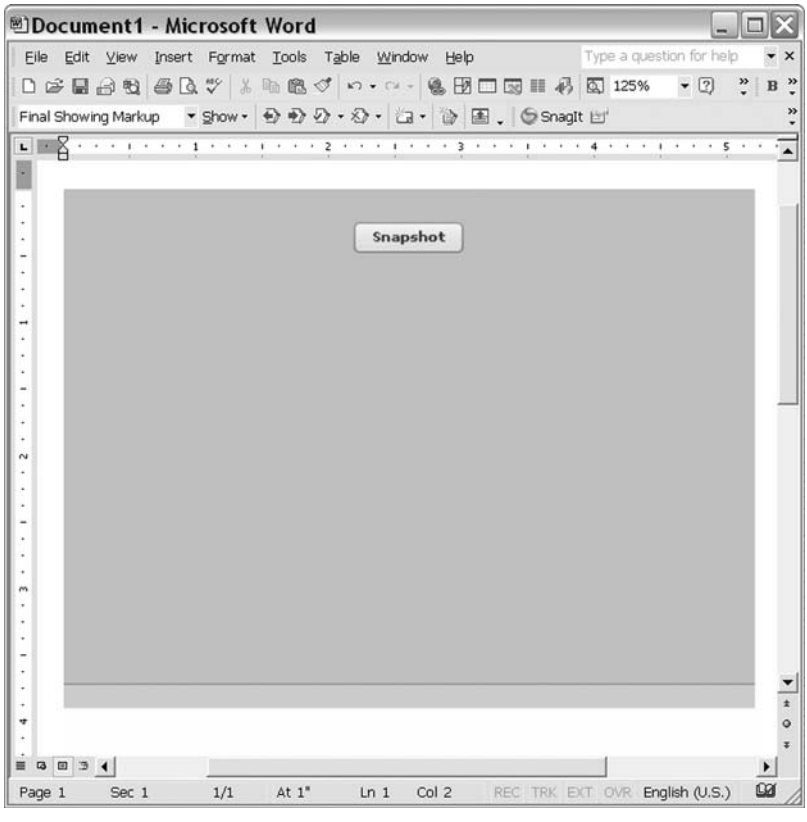


Рис. 4.3. *Снимок можно вставить в другое приложение, например документ Word*

После того как пользователь сделал снимок, нажав на кнопку, он может вставить растровое изображение в другое приложение. На рис. 4.3 приведен пример вставки растрового изображения в документ Word.

В листинге 4.4 показан код этого примера, который, как можно видеть, удивительно прост.

Листинг 4.4. Выполнение мгновенного снимка и помещение его в системный буфер

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[

            private function takeSnapshot():void {
                var bitmapData:BitmapData = new BitmapData(stage.width,
                                                           stage.height);
                bitmapData.draw(stage);
```

❶ Создать объект BitmapData

❷ Сделать снимок

```
            var clipboard:Clipboard =
            ➥Clipboard.generalClipboard;
            clipboard.clear();
            clipboard.setData(ClipboardFormats.BITMAP_FORMAT,
                              bitmapData);
        }

        ]]>
    </mx:Script>
    <mx:Button label="Snapshot" click="takeSnapshot();" />
</mx:WindowedApplication>
```

Получить ссылку на системный буфер

Записать растр в буфер ❸

Кнопка для вызова метода

Как видно из этого примера, нам всего лишь потребовалось создать объект `BitmapData` ❶, скопировать содержимое сцены в объект ❷ и записать его в буфер ❸. В данном случае мы указали формат `BITMAP_FORMAT`.

Тщательно изучив копирование контента, обратимся теперь к операции вставки.

4.2.3. Вставка контента

Вставка контента составляет ту часть операции копирования/вставки, которая относится к чтению. Копирование пишет данные в буфер, а вставка читает их оттуда. При вставке контента в приложение AIR вы осуществляете чтение из системного буфера, считая, что этот контент был создан самим приложением AIR или другим приложением, выполняющимся на компьютере. Например, пользователь мог скопировать файл в системе, а потом попытаться вставить его в приложение AIR.

Вставка контента в приложение AIR состоит всего лишь в применении той технологии извлечения данных, с которой вы ознакомились в разделе 4.1.3. Там вы узнали о методе `getData()`. С помощью метода `getData()` можно извлекать данные из системного буфера в любом формате, который доступен в данный момент. Чтобы продемонстрировать это, мы расширим пример, приведенный в листинге 4.3. В нем пользователь мог копировать текст из текстовой области с помощью меню приложения или окна. Теперь мы добавим средства для вставки контента в текстовую область – также с помощью меню. Листинг 4.5 содержит соответствующий код.

Листинг 4.5. Вставка текста в текущую текстовую область

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
       <![CDATA[

          private var _focused:TextArea;
          private var _clipboard:Clipboard;

          private function creationCompleteHandler():void {
             _clipboard = Clipboard.generalClipboard;
```

```
            var menu:NativeMenu = new NativeMenu();
            var editMenu:NativeMenu = new NativeMenu();
            var copyItem:NativeMenuItem = new NativeMenuItem("copy");
            copyItem.addEventListener(Event.SELECT, copyText);
            var pasteItem:NativeMenuItem = new NativeMenuItem("paste");
            pasteItem.addEventListener(Event.SELECT, pasteText);
            editMenu.addItem(copyItem);
            editMenu.addItem(pasteItem);
            menu.addSubmenu(editMenu, "edit");
            if(NativeApplication.supportsMenu) {
               nativeApplication.menu = menu;
            }
            else if(NativeWindow.supportsMenu) {
               nativeWindow.menu = menu;
            }
         }

         private function copyText(event:Event):void {
            var text:String = _focused.text.substring(
            ➥_focused.selectionBeginIndex, _focused.selectionEndIndex);
            _clipboard.clear();
            _clipboard.setData(ClipboardFormats.TEXT_FORMAT, text);
         }

         private function pasteText(event:Event):void {
            if(_focused is TextArea) {
               var text:String = _clipboard.getData(
               ➥ClipboardFormats.TEXT_FORMAT) as String;
               var currentText1:String = _focused.text.substr(0,
               ➥_focused.selectionBeginIndex);\
               var currentText2:String =
               ➥_focused.text.substr(
               ➥ _focused.selectionEndIndex);
               _focused.text = currentText1 + text + currentText2;
            }
         }

      ]]>
   </mx:Script>
   <mx:HBox width="100%" height="100%">
      <mx:TextArea id="textAreaA" width="50%" height="100%"
         focusIn="_focused = textAreaA;">
         <mx:text>
         Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
         </mx:text>
      </mx:TextArea>
      <mx:TextArea id="textAreaB" width="50%" height="100%"
         focusIn="_focused = textAreaB;" />
   </mx:HBox>
</mx:WindowedApplication>
```

❶ Текстовая область выбрана?

❷ Получить данные о формате текста

Текст до выделения

Текст после выделения

Присвоить новый текст

В этом коде мы сначала проверяем, выделил ли пользователь текст в выбранной области ❶, чтобы избежать обращения к свойствам нулевой

ссылки. Затем мы извлекаем из буфера данные о формате текста ❷. Остальная часть нового кода просто вставляет текст из буфера в место выделения.

Не исключено, что пользователь не скопировал в буфер никакого текста. В таком случае метод `getData()` возвратит `null`, что приведет к вставке в текстовое поле строки со значением `null`. Чтобы избежать этого, можно добавить в код еще одну проверку. Листинг 4.6 показывает, как будет выглядеть метод `pasteText()`, если в операторе `if` добавить обращение к `hasFormat()`.

Листинг 4.6. Проверка существования данных может избавить от лишних вычислений

```
private function pasteText(event:Event):void {
   if(_focused is TextArea &&
   ➥_clipboard.hasFormat(ClipboardFormats.TEXT_FORMAT)) {
      var text:String = _clipboard.getData(
      ➥ClipboardFormats.TEXT_FORMAT) as String;
      var currentText1:String = _focused.text.substr(0,
      ➥_focused.selectionBeginIndex);
      var currentText2:String =
      ➥_focused.text.substr(_focused.selectionEndIndex);
      _focused.text = currentText1 + text + currentText2;
   }
}
```

Проверив, есть ли в буфере обмена данные в нужном формате, мы можем избежать лишнего выполнения кода, если данных в таком формате нет.

4.2.4. Вырезание контента

Многие приложения расширяют операцию копирования/вставки, добавляя возможность *вырезать* (*cut*) контент. Вырезание контента похоже на копирование: в обоих случаях контент копируется в буфер. Однако при вырезании, кроме того, происходит удаление контента из его первоначального места. Можно добиться такого же поведения от приложения AIR, написав немного дополнительного кода. Сейчас мы рассмотрим, как можно добавить функцию вырезания в простое приложение, над которым мы работали в последних двух разделах.

Листинг 4.7 показывает, как будет выглядеть приложение при добавлении функции вырезания в код из листингов 4.5 и 4.6.

Листинг 4.7. Добавление в меню функции cut

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
```

```
private var _focused:TextArea;
private var _clipboard:Clipboard;

private function creationCompleteHandler():void {
   var menu:NativeMenu = new NativeMenu();
   var editMenu:NativeMenu = new NativeMenu();
   var copyItem:NativeMenuItem = new NativeMenuItem("copy");
   copyItem.addEventListener(Event.SELECT, copyText);
   var cutItem:NativeMenuItem = new NativeMenuItem("cut");
   cutItem.addEventListener(Event.SELECT, cutText);
   var pasteItem:NativeMenuItem = new NativeMenuItem("paste");
   pasteItem.addEventListener(Event.SELECT, pasteText);
   editMenu.addItem(copyItem);
   editMenu.addItem(cutItem);
   editMenu.addItem(pasteItem);
   menu.addSubmenu(editMenu, "edit");
   if(NativeApplication.supportsMenu) {
      nativeApplication.menu = menu;
   }
   else if(NativeWindow.supportsMenu) {
      nativeWindow.menu = menu;
   }
   _clipboard = Clipboard.generalClipboard;
}

private function copyText(event:Event):void {
   var text:String = _focused.text.substring(
   ➥_focused.selectionBeginIndex, _focused.selectionEndIndex);
   _clipboard.setData(ClipboardFormats.TEXT_FORMAT, text);
}

private function cutText(event:Event):void {
   copyText(event);
   var currentText1:String = _focused.text.substr(0,
   ➥_focused.selectionBeginIndex);#6
   var currentText2:String = _focused.text.substr(
   ➥_focused.selectionEndIndex);
   _focused.text = currentText1 + currentText2;
   _focused.setSelection(_focused.selectionBeginIndex,
   ➥_focused.selectionBeginIndex);
}

private function pasteText(event:Event):void {
   if(_focused is TextArea) {
      var text:String = _clipboard.getData(
      ➥ClipboardFormats.TEXT_FORMAT) as String;
      var currentText1:String = _focused.text.substr(0,
      ➥_focused.selectionBeginIndex);
      var currentText2:String =
      ➥_focused.text.substr(_focused.selectionEndIndex);
      _focused.text = currentText1 + text + currentText2;
   }
}
```

Создать пункт меню cut

Обрабатывать событие select

Добавить пункт в меню

Обрабатывать событие cut select ❶

Копировать текст

❷ Получить текст до выделения

❸ Получить текст после выделения

Задать выделение

Удалить выделение ❹

```
        ]]>
    </mx:Script>
    <mx:HBox width="100%" height="100%">
        <mx:TextArea id="textAreaA" width="50%" height="100%"
            focusIn="_focused = textAreaA;">
            <mx:text>
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
            </mx:text>
        </mx:TextArea>
        <mx:TextArea id="textAreaB" width="50%" height="100%"
            focusIn="_focused = textAreaB;" />
    </mx:HBox>
</mx:WindowedApplication>
```

Новый код похож на прежний. В методе `cutText()` ❶ мы вызываем `copyText()`, чтобы скопировать текст из поля. Следующие строки ❷ ❸ идентичны коду метода `pasteText()`. Они извлекают текст области, находящийся до и после выделения. Затем мы изменяем текст в поле, удаляя выделенный текст ❹ и устанавливая выделение (подсвеченный текст) по тому индексу, где раньше начинался выделенный текст.

Независимо от того, вырезаете вы текст или копируете, важно определить, как эти данные будут записаны в буфер. Обычно буфер не понимает данные пользовательских типов, и сохраняется базовый объект (экземпляр класса Object), которому присваиваются открытые свойства пользовательского типа данных. В следующем разделе мы узнаем, как заставить приложение сохранять данные пользовательского типа и осуществлять их запись и чтение через буфер обмена.

4.2.5. Пользовательские форматы данных

Ранее в этой главе мы обсуждали работу с пользовательскими форматами. Одна из главных проблем здесь заключается в правильной сериализации данных. Если вы хотите воспользоваться копированием/вставкой для переноса данных между двумя приложениями AIR или в другое приложение, которое понимает AMF (см. главу 3), то обычно сериализация AMF оказывается лучшим выбором, поскольку она имеет встроенную поддержку в AIR и выполняется почти автоматически. В этом разделе мы создадим пользовательский формат и организуем его копирование/вставку между двумя приложениями AIR.

Ранее нами был предложен сценарий, в котором группа родственных переменных была объединена в класс `Address`. Затем мы обсуждали проблему предоставления доступа другим приложениям к экземплярам таких пользовательских классов, как `Address`, через буфер обмена. Сейчас мы построим пример, в котором будет решена именно эта задача. Мы сделаем два простых приложения AIR, в одном из которых пользователь сможет создать новый адрес, а в другом – просматривать созданные адреса. Начнем с того приложения, которое создает адреса. Его внешний вид представлен на рис. 4.4.

Рис. 4.4. *Приложение, создающее адрес, дает пользователю возможность ввести новый адрес и скопировать его в системный буфер обмена*

Приложению-создателю нужен пользовательский тип данных, который мы назовем `Address`. Необходимо определить класс `Address`. Он представлен в листинге 4.8. Обратите внимание, что приложению сообщается о том, как сопоставить класс и псевдоним, с помощью `[RemoteClass]`. Если бы вы строили вариант этого приложения средствами Flash, вместо этого понадобилось бы зарегистрировать псевдоним класса посредством метода `registerClassAlias()`, о котором говорилось в главе 3.

Листинг 4.8. Класс Address, используемый для хранения данных

```
package com.manning.airinaction {

   [RemoteClass(alias="com.manning.airinaction.Address")]
   public class Address {

      private var _address1:String;
      private var _address2:String;
      private var _city:String;
      private var _province:String;
      private var _postalCode:String;

      public function set address1(value:String):void {
         _address1 = value;
      }

      public function get address1():String {
         return _address1;
      }

      public function set address2(value:String):void {
         _address2 = value;
      }
      public function get address2():String {
         return _address2;
```

```
        }

        public function set city(value:String):void {
            _city = value;
        }

        public function get city():String {
            return _city;
        }

        public function set province(value:String):void {
            _province = value;
        }

        public function get province():String {
            return _province;
        }

        public function set postalCode(value:String):void {
            _postalCode = value;
        }

        public function get postalCode():String {
            return _postalCode;
        }

        public function Address() {
        }

    }
}
```

Ничего особенного в классе `Address` нет. Он просто хранит значения адресов 1 и 2, города, области и почтового индекса.

Теперь мы напишем само приложение, создающее адрес. Оно представлено в листинге 4.9.

Листинг 4.9. Приложение, создающее адрес

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute">
    <mx:Script>
        <![CDATA[
            import com.manning.airinaction.Address;

            private function copy():void {
                var address:Address = new Address();
                address.address1 = address1.text;
                address.address2 = address2.text;
                address.city = city.text;
                address.province = province.text;
                address.postalCode = postalCode.text;
                var clipboard:Clipboard = Clipboard.generalClipboard;
                clipboard.clear();
```

Вызывается при щелчке по кнопке

❶ Создать и заполнить адрес

Получить системный буфер

Очистить буфер

```
            clipboard.setData("address", address);
        }

    ]]>
</mx:Script>
<mx:Form>
    <mx:FormItem label="Address 1">
        <mx:TextInput id="address1" />
    </mx:FormItem>
    <mx:FormItem label="Address 2">
        <mx:TextInput id="address2" />
    </mx:FormItem>
    <mx:FormItem label="City">
        <mx:TextInput id="city" />
    </mx:FormItem>
    <mx:FormItem label="State/Province">
        <mx:TextInput id="province" />
    </mx:FormItem>
    <mx:FormItem label="Postal Code">
        <mx:TextInput id="postalCode" />
    </mx:FormItem>
    <mx:FormItem>
        <mx:Button label="Copy" click="copy();" />
    </mx:FormItem>
</mx:Form>
</mx:WindowedApplication>
```

❷ Поместить адрес в буфер

Этот код с помощью компонент MXML создает простую форму, с помощью которой пользователь может заполнить поля адреса. После щелчка по кнопке создается объект `Address`, свойствам которого присваиваются значения соответствующих полей ❶. Затем объект помещается в системный буфер обмена с использованием пользовательского типа данных `address` ❷.

Теперь пользователь может копировать свой специальный формат данных, но нужно еще создать приложение для просмотра адреса, в которое эти данные можно вставить. Такое приложение показано на рис. 4.5.

В листинге 4.10 приведен его код.

Листинг 4.10. Приложение для просмотра адреса

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute">
    <mx:Script>
        <![CDATA[
            import com.manning.airinaction.Address;

            [Bindable]
            private var _address:Address;

            private function paste():void {
                var clipboard:Clipboard = Clipboard.generalClipboard;
```

❶ Объявить свойство типа Address, связанное с источником данных

Получить системный буфер

Рис. 4.5. *Приложение для просмотра адреса дает пользователю возможность вставить объект Address из буфера обмена*

```
            if(clipboard.hasFormat("address")) {
               _address = clipboard.getData("address") as Address;
            }
         }

      ]]>
   </mx:Script>
   <mx:Form>
      <mx:FormItem label="Address 1">
         <mx:Label text="{_address.address1}" />
      </mx:FormItem>
      <mx:FormItem label="Address 2">
         <mx:Label text="{_address.address2}" />
      </mx:FormItem>
      <mx:FormItem label="City">
         <mx:Label text="{_address.city}" />
      </mx:FormItem>
      <mx:FormItem label="State/Province">
         <mx:Label text="{_address.province}" />
      </mx:FormItem>
      <mx:FormItem label="Postal Code">
         <mx:Label text="{_address.postalCode}" />
      </mx:FormItem>
      <mx:FormItem>
         <mx:Button label="Paste" click="paste();" />
      </mx:FormItem>
   </mx:Form>
</mx:WindowedApplication>
```

Получить объект типа Address

Проверить формат адреса данных ❷

В приложении для просмотра адреса используется форма MXML, аналогичная той, которая используется для создания адреса. Основное отличие в том, что вместо текстовых полей при просмотре используются метки, связанные со свойствами объекта `Address`, объявленного как свойство документа ❶. Когда пользователь щелкает по кнопке Paste,

мы выполняем проверку того, что в системном буфере есть данные в формате `address` ❷. Если результат положительный, читаем эти данные и присваиваем свойству `_address`.

На этом мы заканчиваем рассказ о функциях копирования и вставки в приложениях AIR. Как вы могли убедиться, копирование и вставка являются полезной и мощной, хотя и тонкой функцией приложения. Мы завершаем наше обсуждение рассмотрением родственной операции: перетаскивания (drag-and-drop).

4.3. Перетаскивание

Мы изучили более простую из двух операций, связанных с буфером обмена: копирование со вставкой. Перетаскивание вызывает больше проблем, поэтому его обсуждение мы отложили до настоящего момента. Но то, что эта операция сложнее, не означает, что она неимоверно трудна. Как будет показано, ее логика в значительной мере напоминает копирование со вставкой, хотя и несколько более сложна. В следующих разделах мы разберем ее по частям и рассмотрим все действия, необходимые для активизации разного рода операций перетаскивания в приложениях AIR.

4.3.1. Логика перетаскивания

В основных ключевых моментах операция перетаскивания является тем же копированием со вставкой. По существу, перетаскивание состоит в следующем:

- Пользователь инициирует операцию с помощью некоторого движения (например, щелкнув по объекту).
- Пересылаемые данные заносятся в буфер.
- Пользователь начинает перетаскивать объект.
- Пользователь отпускает объект, и если это происходит над разрешенной целью, происходит извлечение данных из буфера.

Как видите, в основе перетаскивания лежат запись в буфер и чтение из него. Разница между двумя операциями состоит, в основном, в способе взаимодействия пользователя с системой.

Основные этапы действий пользователя во время перетаскивания следующие:

1. Пользователь щелкает по объекту, который называется *инициатором*.
2. Пользователь перемещает мышь, удерживая нажатой кнопку, тем самым начиная перемещение инициатора.
3. Пользователь помещает инициатор над другим элементом интерфейса, который настроен на прием сбрасываемых объектов. Этот элемент интерфейса называется *целью перетаскивания*.

4. Система выясняет, может ли цель перетаскивания принять инициатор того типа, который находится над ней.
5. Пользователь сбрасывает инициатор, отпуская кнопку мыши.
6. Система запускает код для приема сбрасываемого объекта, если это допустимо. Если цель не пригодна для приема объекта, операция прекращается.

Как будет видно из последующего изложения, приложение AIR может отвечать за первую часть этих стадий (перетаскивание объекта из приложения AIR), вторую их часть (перетаскивание элемента на приложение AIR) или за все стадии сразу. Сначала мы рассмотрим события, которые сопровождают каждый из этих этапов.

4.3.2. События, возникающие при перетаскивании

Для каждого из этапов операции перетаскивания есть сопутствующее ей событие. Эти события перечислены в табл. 4.2 и являются основой успешной реализации операции перетаскивания.

Таблица 4.2. События при перетаскивании

Событие	Константа NativeDragEvent	Отправитель	Условия отправки
nativeDragStart	NATIVE_DRAG_START	Инициатор	Пользователь начинает перетаскивание
nativeDragUpdate	NATIVE_DRAG_UPDATE	Инициатор	Процесс перетаскивания
nativeDragComplete	NATIVE_DRAG_COMPLETE	Инициатор	Пользователь освобождает объект
nativeDragEnter	NATIVE_DRAG_ENTER	Цель перетаскивания	Перетаскивание вошло в границы целевого объекта
nativeDragOver	NATIVE_DRAG_OVER	Цель перетаскивания	Перетаскивание происходит в границах целевого объекта
nativeDragExit	NATIVE_DRAG_EXIT	Цель перетаскивания	Перетаскивание вышло за границы целевого объекта
nativeDragDrop	NATIVE_DRAG_DROP	Цель перетаскивания	Пользователь освобождает перетаскиваемый объект над целью, которая ранее согласилась его принять

Из перечисленных в табл. 4.2 событий наиболее важны два: nativeDragEnter и nativeDragDrop. Событие nativeDragEnter важно, потому что с его помощью можно определить, что инициатор появился над целью перетаскивания. Это важно, потому что по умолчанию ни один элемент

интерфейса не настроен на то, чтобы принять сброшенный объект. На принятие сброшенного объекта он должен настраиваться во время события nativeDragEnter. (См. в следующем разделе подробности того, как это сделать.) Событие nativeDragDrop важно потому, что во время него система может выполнить код, который определит, что нужно сделать при сбрасывании объекта. Событие nativeDragDrop происходит только тогда, когда цель перетаскивания настроена на принятие сбрасываемого объекта во время последнего из событий nativeDragEnter. В следующем разделе мы увидим примеры того, как все это действует.

Все события из табл. 4.2 имеют тип flash.events.NativeDragEvent. Класс NativeDragEvent определяет статические константы для каждого из событий, имена которых приведены в табл. 4.2

Примечание

У всех событий типа NativeDragEvent есть свойство clipboard. Это свойство указывает на объект Clipboard, содержащий данные для операции перетаскивания. Когда происходит это событие, всегда нужно обращаться к буферу обмена через свойство clipboard объекта NativeDragEvent. Не пытайтесь обращаться к объекту Clipboard другими способами (за исключением процесса его создания).

Есть еще одно событие – mouseDown (типа MouseEvent), которое не попало в табл. 4.2, поскольку не является специфическим для перетаскивания. Событие mouseDown происходит, когда пользователь щелкает по отображаемому интерактивному объекту. Это вполне стандартное событие Flash и Flex, которое также участвует в операциях перетаскивания, поскольку чаще всего перетаскивание начинается в ответ на событие mouseDown. На самом деле, в ответ на события mouseDown или mouseMove (когда нажата кнопка мыши) AIR лишь позволяет вызвать метод doDrag() класса NativeDragManager (см. следующий раздел).

4.3.3. Использование менеджера перетаскивания

В AIR существует специальный управляющий класс для осуществления системных операций перетаскивания: flash.desktop.NativeDragManager. Даже если вы работаете с Flex (где есть класс DragManager), класс NativeDragManager всегда заменяет любые менеджеры перетаскивания из других библиотек, поскольку только NativeDragManager позволяет осуществлять операции перетаскивания на системном уровне.

Именно классом NativeDragManager нужно воспользоваться для начала операций перетаскивания, а также сообщения системе о возможности сбросить объект на цель. Это две главные функции класса NativeDragManager, которые доступны через два статических метода – doDrag() и acceptDragDrop(), которые будут рассмотрены в этом разделе.

Когда пользователь инициирует операцию перетаскивания (часто через событие mouseDown), вы должны вызвать метод NativeDragManager.doDrag(). Метод doDrag() требует по крайней мере два параметра: ссылку

на инициатор перетаскивания и объект `Clipboard`, содержащий передаваемые данные. Листинг 4.11 содержит простой пример, иллюстрирующий эти действия.

Листинг 4.11. Инициация системной операции перетаскивания

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute">
   <mx:Script>
      <![CDATA[

         private function startInitiatorDrag():void {
            initiatorLabel.text = String(Math.random());
            var clipboard:Clipboard = new Clipboard();
            clipboard.setData(ClipboardFormats.TEXT_FORMAT,
                        initiatorLabel.text);
            NativeDragManager.doDrag(initiatorLabel, clipboard);
         }

      ]]>
   </mx:Script>
   <mx:HBox>
      <mx:Canvas width="200" height="50" backgroundColor="#00FF00">
         <mx:Label id="initiatorLabel" width="100%" height="100%"
            mouseDown="startInitiatorDrag();" />
      </mx:Canvas>
      <mx:VBox>
         <mx:Canvas width="200" height="100" backgroundColor="0xFFFF00">
            <mx:Text width="100%" height="100%" />
         </mx:Canvas>
         <mx:Canvas width="200" height="100" backgroundColor="0xFFFF00">
            <mx:Text width="100%" height="100%" />
         </mx:Canvas>
      </mx:VBox>
   </mx:HBox>
</mx:WindowedApplication>
```


В этом примере есть метка ❹, являющаяся инициатором, и две текстовые компоненты ❺ ❻, выступающие как цели перетаскивания. Пока мы еще не подключили текстовые компоненты в качестве целей перетаскивания. Мы сделаем это позже. Сейчас наш код только делает метку инициатором путем вызова метода `startInitiatorDrag()`, когда пользователь щелкнет по ней. При этом метке присваивается произвольный текст ❶, чтобы можно было наблюдать, как что-то действительно происходит. Затем мы копируем это значение в буфер ❷. После этого нужно вызвать метод `doDrag()` и передать ему ссылку на инициатор и буфер ❸.

Примечание

Никогда не пользуйтесь для перетаскивания системным буфером обмена. Хотя со всеми объектами `Clipboard` после получения на них ссылок можно работать

одинаково, системный буфер следует применять только для копирования и вставки. Если вам нужен буфер для перетаскивания, создайте новый объект `Clipboard` с помощью конструктора.

Чтобы завершить пример листинга 4.11, нужно дать текстовым компонентам возможность быть целями перетаскивания. Запустив пример в нынешнем виде, вы можете убедиться, что должным образом сконфигурированных целей нет, поскольку куда бы вы ни перетаскивали мышь, вы видите только значок с перечеркнутым кругом, как на рис. 4.6. Он означает, что под мышью нет объекта, который примет инициатор, если он будет отпущен.

Рис. 4.6. *Когда нет цели, сконфигурированной для приема сбрасывания, виден только значок недопустимости сброса (перечеркнутый кружок)*

Любой интерактивный объект можно настроить в качестве цели перетаскивания, если перехватить и обработать в нем событие `nativeDragEnter`. Когда возникает событие `nativeDragEnter`, нужно вызвать `NativeDragManager.acceptDragDrop()` и передать ему ссылку на объект цели перетаскивания. Тем самым система узнает, что она должна разрешать сброс текущего инициатора на указанную цель перетаскивания. При этом изменяется курсор мыши, как видно на рис. 4.7.

Листинг 4.12 показывает, как изменить пример, чтобы можно было выполнять перетаскивание на текстовые компоненты.

Листинг 4.12. Разрешение сброса на цели

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute">
   <mx:Script>
      <![CDATA[

         private function startInitiatorDrag():void {
            initiatorLabel.text = String(Math.random());
```

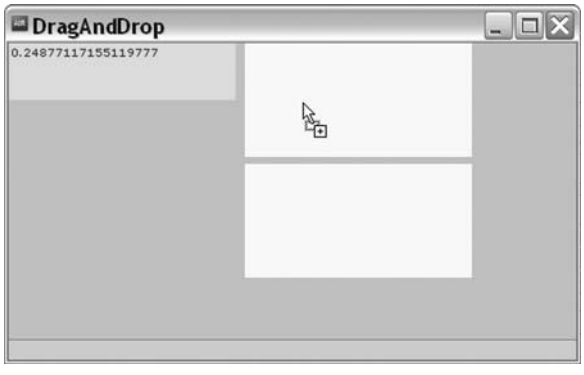


Рис. 4.7. *Когда цель перетаскивания настроена на прием сброса, значок меняется*

```
            var clipboard:Clipboard = new Clipboard();
            clipboard.setData(ClipboardFormats.TEXT_FORMAT,
                        initiatorLabel.text);
            NativeDragManager.doDrag(initiatorObject, clipboard);
         }

         private function nativeDragEnterHandler(event:NativeDragEvent):
         ➥void   {
            NativeDragManager.acceptDragDrop(event.currentTarget
            ➥as Text);
         }
      ]]>
   </mx:Script>
   <mx:HBox>
      <mx:Canvas id="initiatorObject" width="200" height="50"
         backgroundColor="#00FF00" mouseDown="startInitiatorDrag();">
         <mx:Label id="initiatorLabel" />
      </mx:Canvas>
      <mx:VBox>
         <mx:Canvas width="200" height="100" backgroundColor="0xFFFF00">
            <mx:Text width="100%" height="100%"
               nativeDragEnter="nativeDragEnterHandler(event);" />
         </mx:Canvas>
         <mx:Canvas width="200" height="100" backgroundColor="0xFFFF00">
            <mx:Text width="100%" height="100%"
               nativeDragEnter="nativeDragEnterHandler(event);" />
         </mx:Canvas>
      </mx:VBox>
   </mx:HBox>
</mx:WindowedApplication>
```

❶ Разрешить сброс

❷ Зарегистрировать обработчик nativeDragEnter

❸ Зарегистрировать обработчик nativeDragEnter

Мы сделали минимальные изменения, но теперь обе текстовые компоненты могут быть целями сброса для инициатора. Мы всего лишь зарегистрировали обработчик события `nativeDragEnter` для обеих текстовых

компонент ❷ ❸, а затем определили обработчик, вызывающий `acceptDragDrop()` ❶.

Есть еще один шаг для правильного завершения операции перетаскивания: нужно перехватить и обработать событие `nativeDragDrop` для целей перетаскивания. Это событие происходит, когда пользователь сбрасывает инициатор над целью. Листинг 4.13 показывает, какие изменения нужно внести.

Листинг 4.13. Обработка события nativeDragDrop

```
 ?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute">
   <mx:Script>
      <![CDATA[

         private function startInitiatorDrag():void {
         initiatorLabel.text = String(Math.random());
         var clipboard:Clipboard = new Clipboard();
         clipboard.setData(ClipboardFormats.TEXT_FORMAT,
                     initiatorLabel.text);
         NativeDragManager.doDrag(initiatorObject, clipboard);
      }

      private function nativeDragEnterHandler(event:NativeDragEvent):
      ➥void {
         NativeDragManager.acceptDragDrop(event.currentTarget
         ➥as Text);
      }

      private function nativeDragDropHandler(event:NativeDragEvent):
      ➥void {
         var text:Text = event.currentTarget as Text;
         var string:String = event.clipboard.getData(
         ➥ClipboardFormats.TEXT_FORMAT) as String;
         text.text = string;
      }

   ]]>
</mx:Script>
<mx:HBox>
   <mx:Canvas id="initiatorObject" width="200" height="50"
      backgroundColor="#00FF00" mouseDown="startInitiatorDrag();">
      <mx:Label id="initiatorLabel" />
   </mx:Canvas>
   <mx:VBox>
      <mx:Canvas width="200" height="100" backgroundColor="0xFFFF00">
         <mx:Text width="100%" height="100%"
            nativeDragEnter="nativeDragEnterHandler(event);"
            nativeDragDrop=
            ➥"nativeDragDropHandler(event);" />
         </mx:Canvas>
         <mx:Canvas width="200" height="100" backgroundColor="0xFFFF00">
```

❶ Получить данные из буфера

❷ Применить значение к тексту

Перехватить событие nativeDragDrop

```
            <mx:Text width="100%" height="100%"
               nativeDragEnter="nativeDragEnterHandler(event);"
               nativeDragDrop=
               ➥"nativeDragDropHandler(event);" />
         </mx:Canvas>
      </mx:VBox>
   </mx:HBox>
</mx:WindowedApplication>
```

Перехватить событие nativeDragDrop

В этом примере текст извлекается из буфера ❶ и записывается в текстовую компоненту ❷, указывая на успешное завершение перетаскивания.

Мы рассмотрели основы операций перетаскивания. Но этим не ограничиваются наши возможности оживить и усовершенствовать работу программы.

4.3.4. Индикаторы перетаскивания

Как вы могли заметить, по умолчанию во время перетаскивания инициатора за мышью следует простой значок: перечеркнутый кружок, когда мышь находится над целью, которая не принимает сбрасывание, и «плюсик», когда сбрасывание на цель возможно. Такое поведение дает некоторое представление о происходящем, но пользователь мог привыкнуть к другим способам индикации при перетаскивании. Большинство привыкло видеть полупрозрачное изображение инициатора, которое передвигается вслед за мышью. С помощью AIR можно без особого труда организовать такое поведение.

При вызове метода `doDrag()` вы обязаны передать ему первые два параметра, задающие инициатор и буфер обмена. Но можно задать и дополнительные параметры, часть которых относится к полупрозрачному изображению инициатора, следующему за мышью. Чтобы добиться такого эффекта, нужно создать объект `BitmapData` и передать его методу `doDrag()` в качестве третьего параметра. Листинг 4.14 развивает предыдущие примеры, добавляя индикатор перетаскивания. Поскольку изменения касаются только метода `startInitiatorDrag()`, только их мы и приводим в листинге 4.14, а весь остальной код может совпадать с текстом листинга 4.13.

Листинг 4.14. Добавление индикатора перетаскивания

```
private function startInitiatorDrag():void {
    initiatorLabel.text = String(Math.random());
    var clipboard:Clipboard = new Clipboard();
    clipboard.setData(ClipboardFormats.TEXT_FORMAT,
                initiatorLabel.text);
    var initiatorImage:BitmapData =
    ➥new BitmapData(initiatorLabel.width, initiatorLabel.height);
    initiatorImage.draw(initiatorLabel.parent);
    NativeDragManager.doDrag(initiator, clipboard,
    ➥initiatorImage);
}
```

Создать объект BitmapData ❶

❷ Копировать контейнер

❸ Задать индикатор перетаскивания

Все, что потребовалось, – это создать объект BitmapData таких же размеров, как инициатор перетаскивания ❶, нарисовать копию инициатора перетаскивания (в данном случае – контейнер, в котором он расположен) в объекте BitmapData ❷ и затем передать этот объект в качестве третьего параметра метода doDrag() ❸. После этого объект BitmapData автоматически участвует в качестве индикатора перетаскивания и следует за мышью повсюду, даже за пределами приложения AIR.

Как вы могли заметить, индикатор перетаскивания всегда прикрепляется левым верхним углом к курсору мыши. Иногда предпочтительнее, чтобы индикатор был смещен относительно той точки, в которой пользователь щелкнул по инициатору. Например, если пользователь щелкнул в середине инициатора, хорошо было бы, чтобы середина индикатора перетаскивания всегда была связана с курсором мыши. Воздействовать на положение индикатора перетаскивания относительно курсора мыши можно с помощью другого параметра метода doDrag(). Четвертый параметр задается в виде объекта Point и сообщает системе, где должен быть помещен индикатор перетаскивания относительно курсора мыши. Листинг 4.15 показывает дополнительные изменения, которые можно внести в предыдущий пример, чтобы сместить индикатор относительно указателя мыши в зависимости от того, в каком месте инициатора пользователь сделал щелчок.

Листинг 4.15. Смещение индикатора перетаскивания относительно курсора мыши

```
private function startInitiatorDrag():void {
   initiator.text = String(Math.random());
   var clipboard:Clipboard = new Clipboard();
   clipboard.setData(ClipboardFormats.TEXT_FORMAT, initiator.text);
   var initiatorImage:BitmapData =
   ➥new BitmapData(initiator.width, initiator.height);
   initiatorImage.draw(initiator.parent);
   var point:Point = new Point(-initiatorLabel.mouseX,
                        -initiatorLabel.mouseY);
   NativeDragManager.doDrag(initiator, clipboard,
                     initiatorImage, point);
}
```

Теперь, когда нам ясна полная картина перетаскивания внутри приложения AIR, можно рассмотреть ситуации, в которых участвует лишь одна из частей всей процедуры перетаскивания – когда перетаскивание происходит снаружи вовнутрь приложения AIR или наоборот.

4.3.5. Перетаскивание из приложения AIR

Иногда требуется дать пользователю возможность перетащить нечто из приложения AIR в другое приложение. Например, вы хотите, чтобы пользователь мог перетащить изображение из приложения AIR и сохранить его в виде файла на рабочем столе, или перетащить данные из таблицы в приложении в свою электронную таблицу. AIR позволяет делать

такие вещи. На самом деле, для этого достаточно того, что вы уже знаете. В этом разделе мы рассмотрим пример того, как это можно сделать.

Если вам нужно дать пользователю возможность перетащить данные из приложения AIR в другое приложение, достаточно реализовать только первую часть полной последовательности операции перетаскивания. По сути, вам нужно перехватить событие мыши, запускающее операцию, потом записать нужные данные в буфер и вызвать метод `doDrag()`. Часть операции, относящаяся к сбрасыванию объекта, автоматически выполняется другим приложением, и вам нет надобности заниматься целями перетаскивания или чем-либо подобным. Следующий пример листинга 4.16 разрешает пользователю перетащить изображение из приложения AIR в другое приложение (например, в документ Word), которое принимает растровые изображения из буфера обмена.

Листинг 4.16. Перетаскивание изображения из приложения AIR

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute">
   <mx:Script>
      <![CDATA[

         private function startDragImage():void {
            var clipboard:Clipboard = new Clipboard();
            var bitmapData:BitmapData =
            ➥(image.content as Bitmap).bitmapData;
            clipboard.setData(ClipboardFormats.BITMAP_FORMAT,
                        bitmapData, false);
            var point:Point = new Point(-image.mouseX, -image.mouseY);
            NativeDragManager.doDrag(image, clipboard,
                              bitmapData, point);
         }

      ]]>
   </mx:Script>
   <mx:Image id="image" source="image.jpg"
      mouseDown="startDragImage();" />
</mx:WindowedApplication>
```

❶ Получить растровые данные

Занести данные в буфер

Вызвать doDrag()

❷ Показать изображение

В этом примере для показа изображения используется компонента Image ❷. (Учтите, что если вы захотите запустить этот пример, вам понадобится картинка с именем image.jpg в том же каталоге, где находится приложение AIR.) Когда пользователь щелкает по изображению, мы получаем объект `BitmapData` из компоненты ❶, записываем его в буфер и вызываем `doDrag()`. Этот простой код дает пользователю возможность перетаскивать изображение из приложения AIR в другое приложение.

4.3.6. Перетаскивание в приложение AIR

Приложения AIR поддерживают не только перетаскивание контента в прочие приложения, но также позволяют перетаскивать контент из

других приложений в приложения AIR. Для организации перетаскивания в приложения AIR также достаточно тех знаний, которые у вас уже есть. Перетаскивание в приложение AIR – это лишь часть общей операции перетаскивания. По существу, нужно только настроить цели перетаскивания и обработать событие сбрасывания. Листинг 4.17 содержит пример перетаскивания в приложение AIR. Этот простой пример позволяет создать примитивную картинную галерею путем перетаскивания картинок из других приложений (например, веб-броузера) в приложение AIR.

Листинг 4.17. Создание галереи изображений, которая поддерживает перетаскивание картинок

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import mx.controls.Image;

         private function creationCompleteHandler():void {
            addEventListener(
            ➥NativeDragEvent.NATIVE_DRAG_ENTER,
            ➥nativeDragEnterHandler);
            addEventListener(
            ➥NativeDragEvent.NATIVE_DRAG_DROP,
            ➥nativeDragDropHandler);
            var directory:File =
            ➥File.applicationStorageDirectory.resolvePath("images");
            if(!directory.exists) {
               directory.createDirectory();
            }
            var images:Array = directory.getDirectoryListing();
            for(var i:Number = 0; i < images.length; i++) {
               displayImage(images[i]);
            }
         }

         private function displayImage(file:File):void {
            var image:Image = new Image();
            image.source = file.nativePath;
            image.scaleContent = true;
            image.maintainAspectRatio = true;
            image.width = 100;
            tile.addChild(image);
         }

         private function nativeDragEnterHandler(event:NativeDragEvent):
         ➥void  {
            if(event.clipboard.hasFormat(
            ➥ClipboardFormats.FILE_LIST_FORMAT)) {
               NativeDragManager.acceptDragDrop(this);
            }
```

Перехватить событие creationComplete

❶ Перехватить событие nativeDragEnter

❷ Перехватить событие nativeDragDrop

Получить локальный каталог изображений

Создать каталог, если нужно

Показать все изображения

Получить содержимое каталога

❸ Принять сбрасываемые файлы

```
        }

        private function nativeDragDropHandler(event:NativeDragEvent):
        ➥void {
            var files:Array = event.clipboard.getData(
              ➥ClipboardFormats.FILE_LIST_FORMAT) as Array;
            var file:File;
            var newLocation:File;
            for(var i:Number = 0; i < files.length; i++) {
                file = files[i] as File;
                newLocation = File.applicationStorageDirectory.
                ➥resolvePath("images/image" +
                ➥(new Date()).getTime() + i + ".jpg");
                file.moveTo(newLocation);
                displayImage(newLocation);
            }
        }

    ]]>
  </mx:Script>
  <mx:Tile id="tile" width="100%" height="100%" />
</mx:WindowedApplication>
```

Получить файлы

❹ Сохранить файлы локально

❺ Показать изображение

В этом примере мы сначала регистрируем обработчики событий `nativeDragEnter` и `nativeDragDrop` ❶ ❷ для всего приложения. Благодаря этому все приложение становится целью перетаскивания. Когда пользователь перетаскивает что-то на приложение, мы будем принимать только данные, содержащиеся в буфере в файловом формате ❸. После того как пользователь сбросит контент на приложение, мы просмотрим все файлы, сохраним их под уникальными именами ❹ и отобразим ❺.

Мы рассмотрели самые разнообразные применения операций перетаскивания и даже опробовали их практически в таких простых приложениях, как картинная галерея, созданная в этом разделе. Теперь добавим функции перетаскивания в наше приложение AirTube.

4.4. Добавление функций перетаскивания в AirTube

В этом разделе мы добавим некоторые новые функции в приложение AirTube, основываясь на знаниях, полученных в этой главе. Мы предоставим пользователю возможность сохранить файл на своей машине путем перетаскивания элементов из результатов поиска. Мы воспользуемся специальным расширением имен файлов .atv (от «AirTube video»). В последующих главах мы дадим пользователям возможность путем двойного щелчка по файлам .atv запускать соответствующее видео в AirTube.

Чтобы добавить в AirTube функции перетаскивания, достаточно открыть файл com.manning.airtube.ui.VideoTileRenderer.mxml и модифицировать код так, как показано в листинге 4.18.

Листинг 4.18. Добавление функций перетаскивания в AirTube

```
 ?xml version="1.0" encoding="utf-8"?>
<mx:HBox xmlns:mx="http://www.adobe.com/2006/mxml" width="200" height="100"
   verticalScrollPolicy="off" horizontalScrollPolicy="off"
   mouseDown="mouseDownHandler();">
   <mx:Script>
      <![CDATA[

         private function mouseDownHandler():void {
            var file:File = File.applicationStorageDirectory.
            ➥resolvePath("temporary/" + data.video.title + ".atv");
            var writer:FileStream = new FileStream();
            writer.open(file, FileMode.WRITE);
            writer.writeUTF(data.video.id);
            var clipboard:Clipboard = new Clipboard();
            clipboard.setData(ClipboardFormats.FILE_LIST_FORMAT,
            ➥[file]);
            var indicator:BitmapData = new BitmapData(width, height);
            indicator.draw(this);
            var point:Point = new Point(-mouseX, -mouseY);
            NativeDragManager.doDrag(this, clipboard, indicator,
                                     point);
         }

      ]]>
   </mx:Script>
   <mx:Image source="{data.video.thumbnailUrl}" />
   <mx:VBox>
      <mx:Label text="{data.video.title}" />
   </mx:VBox>
</mx:HBox>
```


Когда пользователь щелкает по объекту показа фильма, мы сначала создаем новый файл, исходя из названия фильма ❶. Затем записываем ID фильма в файл ❷. Мы должны создать этот временный файл, потому что именно его мы поместим в буфер ❸, используя формат списка файлов. После этого только остается вызвать `doDrag()`. Обо всем остальном позаботится система. Например, если пользователь перетащит объект на рабочий стол, система создаст копию файла .atv на рабочем столе. Пока мы ничего не собираемся делать с фильмами .atv, но потом мы ими займемся.

4.5. Резюме

В этой главе вы узнали о двух родственных операциях – копировании со вставкой и перетаскивании. Обе они дают пользователю возможность интуитивным образом взаимодействовать с вашим приложением AIR. Поскольку эти операции происходят на системном уровне, с их помощью можно обмениваться данными между приложениями AIR или приложениями AIR и другими приложениями. Например, вы

видели, как пользователь может перетащить изображение с веб-страницы и сбросить его в приложение AIR.

Вы узнали, что копирование/вставка и перетаскивание пользуются буфером обмена как общим передатчиком данных. Буферы обмена представлены в приложениях AIR в виде объектов `Clipboard`. Существуют разнообразные форматы, которые можно применять для данных, помещаемых в буфер или считываемых из него, благодаря чему можно обмениваться данными самых разных типов. Буфер обмена лежит в центре операций обоих типов, но фактический код, применяемый в операциях копирования/вставки и перетаскивания, несколько различен. По этой причине мы перешли от общих рассуждений о буферах обмена к подробному рассмотрению передачи данных в ходе операций того и другого типа.

Завершив наше обсуждение копирования/вставки и перетаскивания, мы можем перейти к следующей главе. В главе 5 мы рассмотрим работу с локальными базами данных.

5

В этой главе:
- Основы SQL
- Создание баз данных
- Выполнение операторов SQL
- Применение параметров
- Получение наборов записей

Работа с локальными базами данных

В главе 3 вы научились работать с файловой системой. Файловая система позволяет хранить данные на постоянной основе и структурировать их разными способами. Запись данных в файлы оказывается отличным решением во многих случаях. Например, файловый формат идеально подходит для записи таких двоичных данных, как графика или видео, которые вашему приложению может потребоваться загружать во время работы. Кроме того, файлы хорошо обеспечивают портируемость. Если нужно вывести текст в таком формате, чтобы пользователь мог легко переслать его своему другу, текст удобно передавать в файле. Но, несмотря на всю свою пользу, файлы не являются универсальным средством для работы с данными любого вида. В данной главе мы рассмотрим более эффективный способ работы с некоторыми видами данных: локальные базы данных. База данных позволяет организовать постоянное хранение данных. Большинство баз данных также предоставляет язык, с помощью которого можно эффективно хранить и извлекать данные. В AIR применяется база данных под названием *SQLite*, а базы SQLite пользуются языком, который называется языком структурированных запросов (Structured Query Language), или просто SQL.

Примечание

Люди по-разному произносят SQL. Одни делают это по буквам – *Эс-Ку-Эль*, другие произносят, как слово «сиквел». Мы предпочитаем второй вариант.

С помощью SQL можно обрабатывать наборы данных гораздо эффективнее, чем путем написания собственного кода для их анализа. В этой главе мы расскажем обо всем, что вам потребуется для работы с локальными базами данных из приложений AIR. Мы расскажем о том, как

создавать новые базы данных, как записывать в них данные, как их читать, обновлять и о многом другом. Мы также приведем несколько примеров приложений и модифицируем приложение AirTube, чтобы оно могло работать с локальной базой данных в автономном режиме.

Примечание

AIR позволяет создавать временные базы данных, хранящиеся в оперативной памяти во время работы приложения. При выходе из приложения такие базы данных уничтожаются. Несмотря на наличие такой возможности, наше внимание будет посвящено в этой главе исключительно постоянным базам данных, хранящимся на диске. Обычно постоянные базы данных оказываются гораздо полезнее, просто потому, что данные сохраняются в них гораздо дольше (пока не будут удалены явным образом).

Чтобы понять, как работать с базами данных в AIR, очень важно знать основы SQL. Мы допускаем, что у читателя может не быть достаточных знаний в этой области. Поэтому там, где это уместно, мы объясняем базовые операторы SQL, которыми пользуемся. Читатель, знакомый с SQL, может пропустить эти объяснения.

Примечание

Хотя мы и рассказываем о принципах SQL на протяжении этой главы, она ни в коей мере не предназначена служить исчерпывающим руководством по SQL. Мы рекомендуем прочесть книгу, специально посвященную этому вопросу. Ряд полезных руководств можно найти в Сети. Отметим такой отличный ресурс, как *www.w3schools.com/sql*.

Прежде чем заняться деталями реализации, нужно обсудить, что такое база данных и для чего она может понадобиться. Об этом говорится в следующем разделе.

5.1. Что такое база данных?

Если вы специалист по базам данных и SQL, то можете пропустить этот раздел, как и ряд других в этой главе, и сразу заняться деталями реализации. Всем остальным, кто хочет получить краткое введение или освежить свои знания, предлагаем потратить несколько минут на разговор о том, что такое базы данных и чем они могут быть полезны.

Попросту говоря, база данных – это определенным образом организованный набор информации, обычно в виде групп, называемых *записями* (*records*). Каждая запись представляет собой заданным образом организованный набор данных. Например, если в записях базы данных хранятся адреса, то они могут состоять из таких элементов, как адрес 1, адрес 2, город, область, почтовый код и страна. Часто удобно представлять себе эти записи в виде строк, как в электронной таблице (которая представляет собой базу данных простого вида). Рисунок 5.1 наглядно представляет запись, содержащую адрес.

StreetAddress1	StreetAddress2	City	Province	PostalCode	Country
1212 Road Street	#555	Plainsville	BC	ABCDEF	Canada

Рис. 5.1. *Базы данных состоят из записей вроде приведенной на этом рисунке*

Есть много способов моделирования баз данных, включая иерархические, сетевые и объектные модели. SQLite – движок, используемый с AIR, – является реляционной базой данных, что означает использование реляционной модели *таблиц* – весьма распространенная и простая для понимания модель. Таблица состоит из колонок и строк. Колонки таблицы иногда называют *атрибутами*, но чаще – просто *колонками*. Например, на рис. 5.1 имеются следующие атрибуты, или колонки: StreetAddress1, StreetAddress2, City, Province, PostalCode и Country. Строки таблицы – это *записи*, а запись – самый мелкий объект, который можно вставить или удалить из таблицы. (Нельзя вставить данные в одну ячейку в колонке, не вставив при этом целую строку.)

Примечание

Подробнее об SQLite можно узнать, посетив официальный веб-сайт *www.sqlite.org*.

В любой базе данных может быть несколько таблиц, а данные в таблицах могут быть связаны между собой, откуда и термин «реляционная» база данных. Связи обычно основаны на ключах (keys). Ключ позволяет уникально идентифицировать запись. Например, можно добавить в таблицу рис. 5.1 колонку ID. Рисунок 5.2 показывает, как теперь будет выглядеть таблица.

ID	StreetAddress1	StreetAddress2	City	Province	PostalCode	Country
0	1212 Road Street	#555	Plainsville	BC	ABCDEF	Canada

Рис. 5.2. *Колонка ID используется в качестве первичного ключа*

Ключ, используемый таким образом в качестве уникального идентификатора строки, называется *первичным* (*primary*). Первичные ключи позволяют легко устанавливать связи между данными в таблицах. Разовьем наш пример, чтобы показать, как это делается. На рис. 5.3 показана таблица с адресами, в которую добавлена вторая запись.

ID	StreetAddress1	StreetAddress2	City	Province	PostalCode	Country
0	1212 Road Street	#555	Plainsville	BC	ABCDEF	Canada
1	4 Route Lane	#8	Lakeview	AK	99999	US

Рис. 5.3. *Можно помещать в таблицу новые записи*

Теперь мы заведем таблицу служащих, в которой будут содержаться ID служащего, его имя и должность. Эта таблица показана на рис. 5.4.

ID	Name	Title
0	Sarah	Manager
1	Francois	Auditor
2	Pan	Human Resource Manager
3	Wendy	New Business
4	Sidney	Trainer

Рис. 5.4. *Таблица служащих показывает ID, имя и должность*

Теперь мы можем установить связь между данными в таблицах, добавив еще одну таблицу. Эта новая таблица будет показывать, по какому адресу работает каждый служащий. Она приведена на рис. 5.5. Как можно видеть, некоторые служащие работают более чем по одному адресу. Например, служащий с ID 2 работает по двум адресам.

EmployeeID	AddressID
0	0
1	0
1	1
2	0
2	1
3	1
4	0

Рис. 5.5. *Создание связей между таблицами с помощью первичных ключей*

Движок базы данных SQLite, используемый в AIR, записывает все эти таблицы в один или несколько файлов на вашей машине. Любая запись или чтение из файлов осуществляются посредством SQL, в том числе создание новых таблиц, удаление существующих, вставка данных, обновление и удаление их. О том, как эти операции осуществляются с помощью SQL, подробнее рассказывается в разделе 5.2.

Теперь, когда вы познакомились со структурой баз данных, может возникнуть вопрос о том, когда их следует применять. Хотя не существует четких правил для принятия такого решения, следующие рекомендации могут оказаться полезными:

- Если приложение AIR использует данные, которые по своей природе являются реляционными, применение баз данных оказывается оправданным.
- Если вам потребуется поиск данных по различным критериям, база данных может быть удобным средством хранения данных, поскольку SQL поддерживает множество способов фильтрации, объединения и сортировки наборов данных.
- Локальная база данных хорошо подходит для буферизации данных, полученных из сети, позволяя приложению AIR продолжить работу с локальными данными, когда оно окажется в автономном

режиме. При восстановлении соединения с Интернетом локальные данные могут быть обновлены.

- Если приложение AIR подключается к интернет-ресурсам для записи данных, то база данных позволяет приложению хранить введенные или обновленные пользователем данные даже в отсутствие подключения к Интернету. Когда пользователь снова подключится к Интернету, приложение AIR сможет прочесть данные из локальной базы данных и записать их в удаленный ресурс.

Теперь рассмотрим язык SQL, с помощью которого можно работать с базами данных AIR.

5.2. Понятие об SQL

Как уже отмечалось, SQL – слишком большая тема, чтобы исчерпывающим образом изложить ее в этой главе. Но мы хотим быть уверены, что даже если вы еще не знакомы с SQL, то все равно эта глава окажется полезной для вас. Поэтому в данном разделе вы узнаете о ряде основных команд SQL, которыми вы сможете воспользоваться. Если эти команды вам знакомы, можете перейти к следующему разделу. Либо быстро просмотрите последующую информацию, чтобы познакомиться с деталями, специфичными для SQLite.

Если вы не знакомы с SQL или просто хотите пробежаться по этому разделу, вам может стать полезным приложение AIR SQLTutorial, которое доступно на официальном сайте этой книги на *www.manning.com/lott*. Приложение SQLTutorial показано на рис. 5.6.

Приложение SQLTutorial автоматически создает базу данных и дает возможность выполнять операторы SQL в этой базе данных. В левой верхней части приложения показаны все имеющиеся таблицы. При первом запуске приложения никаких таблиц нет, как на рис. 5.6. В следующем разделе вы добавите в базу данных таблицу. После того

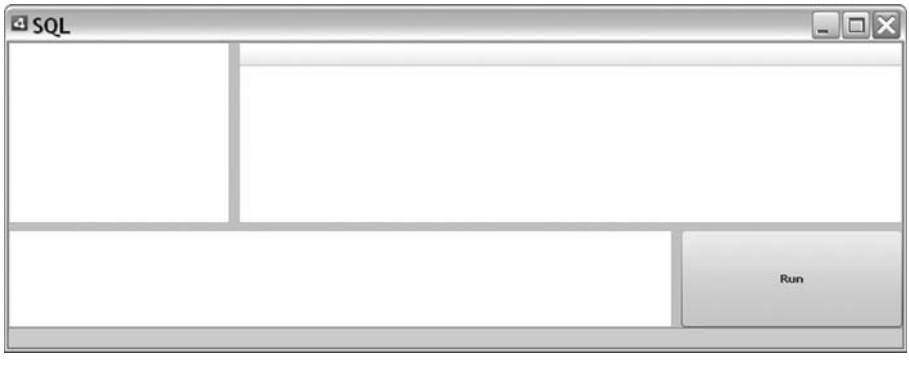


Рис. 5.6. *Приложение SQLTutorial дает возможность выполнять примеры использования SQL, приводимые в следующих разделах*

как появятся таблицы, можно выбрать одну из них, и ее содержимое будет показано в сетке данных справа. В нижней части приложения SQLTutorial есть текстовое поле для ввода команд SQL, которые выполняются при нажатии кнопки Run.

5.2.1. Создание и удаление таблиц

Как уже отмечалось, одним из основных элементов базы данных является таблица. Фактически, нельзя хранить данные, не имея хотя бы одной таблицы. Поэтому создание таблиц является одной из главных команд SQL. Таблицу можно создать с помощью команды `CREATE TABLE`. Она выглядит следующим образом:

```
CREATE TABLE tableName (column[, column, ...])
```

Далее следует конкретный пример команды `CREATE TABLE`, которая создает новую таблицу с именем musicTracks, содержащую колонки id, title, artist, album, length и originalReleaseYear:

```
CREATE TABLE musicTracks (id, title, artist, album, length,
➥originalReleaseYear
```

В SQLite используется понятие *классов памяти* (*storage classes*). Класс памяти дает возможность эффективно хранить данные благодаря тому, что памяти отводится не больше, чем требует тип хранимых данных. SQLite поддерживает следующие классы памяти:

- `NULL` – только для нулевых значений
- `INTEGER` – целые числа со знаком
- `REAL` – числа с плавающей точкой
- `TEXT` – текст, хранимый в принятой для базы данных кодировке, например UTF-8
- `BLOB` – данные, хранимые без преобразований, как есть (то есть допускается хранение двоичных данных)

Чтобы помочь базе данных распознать, какие классы памяти использовать в каждой колонке, можно указать для колонки предпочтительный тип (*affinity*). В результате база данных отдает предпочтение некоторым типам данных, пытаясь преобразовать в них данные, прежде чем ввести их. Например, если указать для колонки тип родства `INTEGER` и попытаться ввести в нее строковое значение, база данных попробует преобразовать его в целое число. Допустимо использование следующих типов:

- `TEXT` – движок пытается сначала преобразовать данные в текст. Это означает, что числа вставляются в виде строк.
- `NUMERIC` – по возможности движок преобразует строки в числа – целые или с плавающей точкой – и записывает их в соответствующем классе хранения. Значения null или blob не преобразуются.
- `INTEGER` – основное отличие от `NUMERIC` в том, что строка, которая может быть преобразована в число с незначащей десятичной частью (например, 5.0), будет преобразована в целое число.

- REAL – все числовые значения преобразуются в величины с плавающей точкой, даже если дробная часть незначима.
- NONE – попытки преобразования не делается. Это значение родственности по умолчанию, если не задано никакое другое.

Предпочтительный тип можно указать при создании таблицы после имени колонки. В следующем примере создается таблица с колонками id, length и originalReleaseYear типа INTEGER, а для остальных предпочтительный тип TEXT:

```
CREATE TABLE musicTracks (id INTEGER, title TEXT, artist TEXT, album TEXT,
➥length INTEGER, originalReleaseYear INTEGER)
```

Очевидно, что создать таблицу можно лишь тогда, когда ее не существует. Если попытаться создать таблицу с уже существующим именем, возникнет ошибка.

Однако ее можно избежать, если воспользоваться в операторе CREATE TABLE выражением IF NOT EXISTS (если не существует). Следующая команда создает таблицу musicTracks, если ее не существует. Если же такая таблица уже есть, ничего не происходит:

```
CREATE TABLE IF NOT EXISTS musicTracks (id INTEGER, title TEXT,
➥artist TEXT, album TEXT, length INTEGER, originalReleaseYear INTEGER)
```

Можно устанавливать ограничения на колонки, указывая их в определении сразу после типа. SQLite поддерживает несколько типов ограничений, но для целей нашей книги представляет интерес только ограничение PRIMARY KEY (первичный ключ). Это ограничение требует, чтобы в данной колонке для каждой записи находилось уникальное значение. (Для каждой таблицы можно сделать первичным ключом только одну колонку.) Если типом колонки является INTEGER и она определена как PRIMARY KEY, то можно добавить в конце ключевое слово AUTOINCREMENT. В этом случае при каждом добавлении новой строки в данную колонку будет автоматически вставляться новое значение, на 1 большее, чем предыдущее вставленное значение. Следующая команда создает таблицу с колонкой id, действующей как автоинкрементируемый первичный ключ:

```
CREATE TABLE IF NOT EXISTS musicTracks (id INTEGER PRIMARY KEY
➥AUTOINCREMENT, title TEXT, artist TEXT, album TEXT,
➥length INTEGER, originalReleaseYear INTEGER)
```

Если хотите сами поэкспериментировать, введите приведенную выше команду в текстовом поле приложения SQLTutorial и нажмите кнопку Run. В списке таблиц должна появиться musicTracks. Если выделить эту таблицу в списке, появится список колонок, как на рис. 5.7.

Создание таблиц – это одна сторона вопроса. Другая сторона – их удаление. Удалять гораздо проще, чем создавать. Чтобы удалить таблицу, выполните команду DROP TABLE, задав имя таблицы, которую нужно удалить. Например, следующий код удаляет таблицу musicTracks:

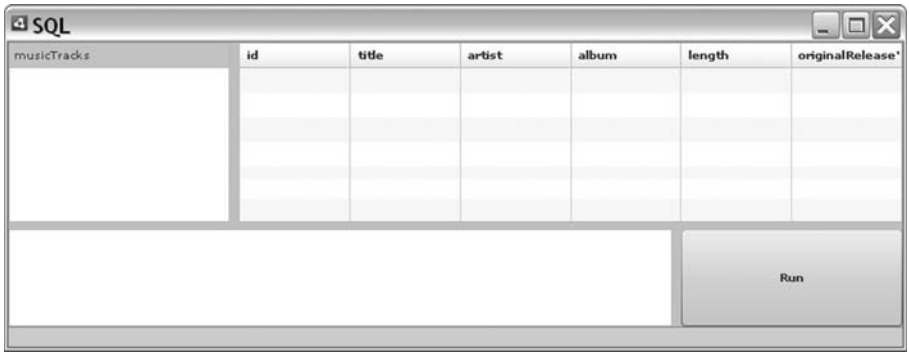


Рис. 5.7. *Создав таблицу и выделив ее в списке, вы можете увидеть ее колонки*

```
DROP TABLE musicTracks
```

Здесь также можно добавить выражение `IF EXISTS` (аналог `IF NOT EXISTS` для `CREATE TABLE`), чтобы избежать ошибки, когда таблица не существует:

```
DROP TABLE IF EXISTS musicTracks
```

Примечание

Если вы проверили код `DROP TABLE` в SQLTutorial, вам нужно снова создать эту таблицу, прежде чем переходить к следующему разделу.

Это все, что касается основ создания и удаления таблиц. Далее мы рассмотрим, как добавлять данные в таблицы.

5.2.2. Добавление данных в таблицы

Таблицы – это всего лишь контейнеры для данных. После того как таблицы созданы, нужно поместить в них данные. В SQL добавление новых данных называется *вставкой* и осуществляется командой `INSERT`.

Команда `INSERT` имеет примерно такой вид:

```
INSERT INTO tableName (column[, column, …]) VALUES (value[, value, …])
```

Количество и порядок колонок и значений должны совпадать. Например, следующая команда вставляет новую запись в таблицу musicTracks. Обратите внимание, что в списке колонок и в списке значений содержится по пять элементов и их порядок совпадает:

```
INSERT INTO musicTracks(title, artist, album, length, originalReleaseYear)
➥VALUES("Just Another Day", "Oingo Boingo", "Dead Man's Party", 243, 1985)
```

Вы могли обратить внимание, что в этом примере мы опустили колонку id в команде `INSERT`. Это связано с тем, что мы наложили на нее ограничение `PRIMARY KEY AUTOINCREMENT`. В результате движок базы данных автоматически присваивает значение колонке id при введении новой записи.

Еще можно было обратить внимание на то, что текстовые величины заключены в кавычки. Если не заключать текст в кавычки, то база данных пытается интерпретировать его как название колонки, а не как текстовое значение. Поэтому текстовые величины нужно заключать в кавычки. (Кавычки могут быть как двойными, так и одинарными.) Числовые величины заключать в кавычки не нужно.

Если хотите проверить этот код, выполните приведенную команду в SQLTutorial. После выполнения кода вы увидите новую запись, как на рис. 5.8.

После того как вы вставили данные, вы не сможете вставить их заново, чтобы сделать какие-то изменения. Вместо этого вам придется воспользоваться командой обновления, описываемой в следующем разделе.

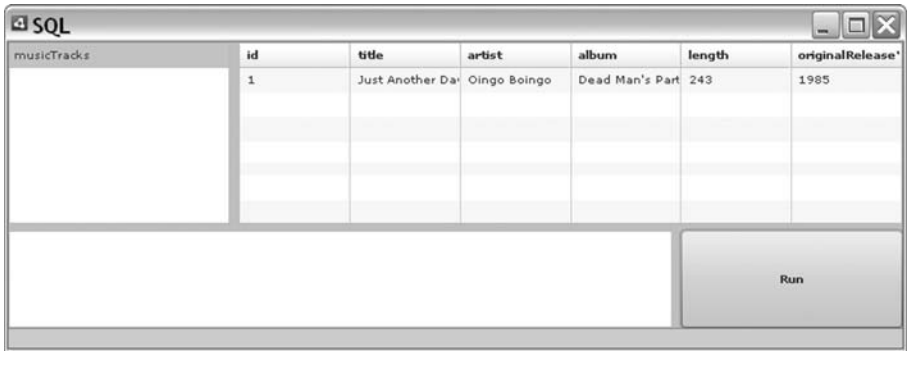


***Рис. 5.8.** Вставьте запись в таблицу, и она появится в сетке данных*

5.2.3. Редактирование данных в таблицах

Редактирование существующих записей называется обновлением и осуществляется с помощью команды `UPDATE`. Синтаксис команды `UPDATE` следующий:

```
UPDATE tableName SET column = value[, column = value, …] WHERE expression
```

Выражение `SET` состоит из списка колонок и их новых значений. Обычно затем следует выражение `WHERE`, задающее строки, которые нужно обновить. Если опустить выражение `WHERE`, то будут обновлены все строки, что обычно не является планируемым результатом команды. В следующем примере обновляется таблица musicTracks путем задания значения 242 для колонки length в записи с id, равным 1 (поскольку id – первичный ключ, изменена будет только одна запись):

```
UPDATE musicTracks SET length = 242 WHERE id = 1
```

Для создания сложных выражений `WHERE` можно применять операторы `AND` и `OR`. Например, чтобы обновить имя исполнителя для всех треков

альбома Oingo Boingo *Dead Man's Party*, можно выполнить такую команду:

```
UPDATE musicTracks SET artist = "OINGO BOINGO"
➥WHERE artist = "Oingo Boingo" AND album = "Dead Man's Party"
```

Как вы могли заметить, в этих примерах мы заключили текстовые значения в кавычки так же, как делали это в командах `INSERT`.

Если вы работаете параллельно с SQLTutorial, выполните приведенную команду SQL. В результате вы увидите, что имя исполнителя Oingo Boingo будет заменено на OINGO BOINGO. Если в таблице были другие записи с именем артиста Oingo Boingo, но с иными названиями альбомов, эти записи не будут изменены.

Ознакомившись с тем, как вставляются и изменяются данные, нужно выяснить, как их удалять, что и будет обсуждаться в следующем разделе.

5.2.4. Удаление данных из таблиц

Удалять данные из таблиц можно с помощью команды `DELETE`. Синтаксис команды `DELETE` следующий:

```
DELETE FROM tableName WHERE expression
```

Выражение `WHERE` в команде `DELETE` не является обязательным (так же, как оно не требуется в команде `UPDATE`), но используется почти всегда. Если опустить `WHERE`, то команда `DELETE` удалит все записи из указанной таблицы. Обычно нужно удалить только одну или несколько записей. Выражение `WHERE` позволяет ограничить удаление только теми записями, которые отвечают критерию в выражении. Например, следующая команда удаляет только записи с id, равным 1:

```
DELETE FROM musicTracks WHERE id = 1
```

Можно воспользоваться составными условиями – так же, как в команде `UPDATE`. Например, следующая команда удаляет все записи, в которых артистом является Devo или Oingo Boingo:

```
DELETE FROM musicTracks WHERE artist = "Devo" OR artist = "Oingo Boingo"
```

Мы уже рассказали о том, как вставлять, обновлять и удалять данные, но еще не показали, как извлекать их из базы данных. Об этом мы поговорим в следующем разделе.

5.2.5. Извлечение данных из таблиц

Извлечение данных из базы может быть простой или сложной задачей в зависимости от требований. Например, извлечь из таблицы все записи – просто, а извлечь только сумму значений в одной колонке для некоторой группы записей оказывается сложнее. Мы начнем рассмотрение извлечения данных с простейших примеров, постепенно переходя

к более сложным. Если вы хотите проверить примеры, приведенные в этом разделе, можете воспользоваться приложением SQLTutorial. Чтобы получить в точности такие же результаты, как приведенные в книге, можно выполнить в SQLTutorial специальную команду, которая запишет в таблицу musicTracks те же значения, что и в нашем контрольном примере. Достаточно выполнить команду `INITIALIZE musicTracks`, и вы увидите, как будет создана (если нужно) таблица musicTracks и в нее будут добавлены данные. Даже если вы уже создали musicTracks, можете начать все сначала, выполнив команду `DROP musicTracks`, а потом уже — `INITIALIZE musicTracks`.

Примечание

`INITIALIZE musicTracks` является специальной командой приложения SQLTutorial и не входит в стандартный SQL.

Набор данных, с которым мы будем работать в этом разделе, показан в табл. 5.1.

Этот набор будет использоваться во всех последующих примерах раздела. Теперь мы готовы начать изучение основ извлечения данных.

Таблица 5.1. Набор данных для всех примеров раздела

Id	title	artist	album	length	originalReleaseYear
1	Just Another Day	Oingo Boingo	Dead Man's Party	243	1985
2	I Scare Myself	Thomas Dolby	The Flat Earth	299	1984
3	Senses Working Overtime	XTC	English Settlement	270	1984
4	She Blinded Me With Science	Thomas Dolby	The Golden Age of Wireless	325	1982
5	Heaven	The Psychedelic Furs	Mirror Moves	250	1984
6	Dance Hall Days	Wang Chung	Points on the Curve	258	1984
7	Love My Way	The Psychedelic Furs	Forever Now	208	1982
8	Everybody Have Fun Tonight	Wang Chung	Mosaic	287	1986

Команда SELECT

Независимо от сложности решаемой вами задачи любое извлечение данных происходит с помощью команды `SELECT`. В простейшем виде синтаксис команды `SELECT` следующий:

```
SELECT column[, column, …] FROM tableName
```

Можно еще более упростить команду, воспользовавшись маской * вместо названий колонок. Маска * позволяет выбрать все колонки из указанной таблицы. Например, следующая команда извлекает все колонки всех записей таблицы musicTracks:

```
SELECT * FROM musicTracks
```

С точки зрения движка базы данных всегда гораздо эффективнее перечислить все колонки, а не применять маску. Кроме того, перечисляя колонки, вы контролируете не только выбор колонок, но и их порядок. Например, если вы хотите извлечь только колонки с названиями и исполнителями для всех записей таблицы musicTracks, можно выполнить команду:

```
SELECT title, artist FROM musicTracks
```

Результат показан в табл. 5.2.

Таблица 5.2. Колонки title и artist, возвращенные для всех записей

title	artist
Just Another Day	Oingo Boingo
I Scare Myself	Thomas Dolby
Senses Working	Overtime XTC
She Blinded Me With Science	Thomas Dolby
Heaven	The Psychedelic Furs
Dance Hall Days	Wang Chung
Love My Way	The Psychedelic Furs
Everybody Have Fun Tonight	Wang Chung

В команду `SELECT` можно также добавить выражение `WHERE`. С помощью `WHERE` можно фильтровать результаты в соответствии с выражением. Например, чтобы извлечь названия всех треков, исполнителем которых является Thomas Dolby, можно выполнить команду:

```
SELECT title FROM musicTracks WHERE artist = "Thomas Dolby"
```

Результат выполнения показан в табл. 5.3.

Таблица 5.3. Названия в musicTracks, исполнитель которых Thomas Dolby

title
I Scare Myself
She Blinded Me With Science

Допускается использование составных выражений. Например, следующая команда извлекает все записи, выпущенные в 1984 году, продолжительность которых больше 260:

```
SELECT title FROM musicTracks
➥WHERE originalReleaseYear = 1984 AND length > 260
```

Результат выполнения показан в табл. 5.4.

Таблица 5.4. Можно выбирать данные на основе сложных выражений

title
I Scare Myself
Senses Working Overtime

Это все, что касается основ работы с командами SELECT. В следующих разделах мы рассмотрим другие способы модификации команд SELECT, позволяющие создавать более сложные запросы.

Удаление дубликатов

Как вы видели, по умолчанию возвращаются все результаты, удовлетворяющие критерию. Но иногда нужно исключить повторяющиеся значения. Например, если вы запросите всех артистов из таблицы musicTracks с помощью следующей команды, то получите повторяющиеся значения:

```
SELECT artist FROM musicTracks
```

Результат выполнения этой команды показан в табл. 5.5.

Таблица 5.5. Иногда результирующий набор содержит повторяющиеся значения

artist
Oingo Boingo
Thomas Dolby
XTC
Thomas Dolby
The Psychedelic Furs
Wang Chung
The Psychedelic Furs
Wang Chung

Если вы хотите исключить повторяющиеся значения, то воспользуйтесь ключевым словом DISTINCT вместе с командой SELECT, как в следующем примере:

```
SELECT DISTINCT artist FROM musicTracks
```

Результаты выполнения этой команды не содержат дубликатов, как видно из табл. 5.6.

Таблица 5.6. Ключевое слово DISTINCT устраняет повторения

artist
Oingo Boingo
Thomas Dolby
XTC
The Psychedelic Furs
Wang Chung

Теперь, когда вы умеете устранять повторяющиеся значения, вас может заинтересовать возможность сортировки результатов. Хорошая идея, как раз этим мы сейчас и займемся.

Упорядочение результатов

Когда вы часто извлекаете из базы данных некий набор данных, может оказаться желательным упорядочить эти данные определенным образом. Например, при извлечении данных из таблицы musicTracks вам захочется расположить результаты в алфавитном порядке или хронологически по году выпуска. В любом случае, такого рода сортировку можно осуществить с помощью выражения `ORDER BY` в команде `SELECT`.

Выражение `ORDER BY` позволяет задать одну или несколько колонок, по которым нужно провести сортировку, данных, а также порядок сортировки: возрастающий (по умолчанию) или убывающий. Вот пример простого выражения `ORDER BY`:

```
SELECT album, originalReleaseYear FROM musicTracks
➥ORDER BY originalReleaseYear
```

Это выражение упорядочивает результаты по возрастанию года выпуска. Результаты показаны в таблице 5.7.

Таблица 5.7. Использование выражения ORDER BY для сортировки данных

album	originalReleaseYear
Forever Now	1982
The Golden Age of Wireless	1982
Points on the Curve	1984
Mirror Moves	1984
English Settlement	1984
The Flat Earth	1984
Dead Man's Party	1985
Mosaic	1986

Можно явным образом указать базе данных, что данные должны быть отсортированы в порядке возрастания, воспользовавшись ключевым

словом ASC, или задать DESC, чтобы потребовать сортировки в порядке убывания. Следующая команда отсортирует результаты, начав с ближайшего года:

```
SELECT album, originalReleaseYear FROM musicTracks
➥ORDER BY originalReleaseYear DESC
```

Можно добавить в выражение сортировки дополнительные колонки в виде списка, разделенного запятыми. Первая колонка в списке оказывает решающее влияние, поскольку определяет первоначальный порядок сортировки. После того как данные отсортированы по первой колонке, вступает в игру вторая, более точно определяя результаты сортировки. Например, вы можете заметить, что в табл. 5.7 данные отсортированы по году выпуска, но не вполне очевидно, как отсортированы данные внутри каждого года выпуска. В результирующем наборе есть четыре записи с годом выпуска 1984, но порядок их дальнейшей сортировки не очевиден. Потребовать от движка базы данных, чтобы сортировка происходила сначала по году выпуска, а потом по названию альбома, можно с помощью следующей команды:

```
SELECT album, originalReleaseYear FROM musicTracks
➥ORDER BY originalReleaseYear, album
```

Результаты представлены в табл. 5.8.

Таблица 5.8. Теперь результаты отсортированы по году выпуска

album	originalReleaseYear
Forever Now	1982
The Golden Age of Wireless	1982
English Settlement	1984
Mirror Moves	1984
Points on the Curve	1984
The Flat Earth	1984
Dead Man's Party	1985
Mosaic	1986

Можно не только упорядочивать данные, но и выполнять функции для извлечения из базы данных специальных значений, что будет продемонстрировано в следующем разделе.

Выполнение функций

SQL позволяет выполнять в запросах функции. Полный список функций, допускаемых SQLite, есть на странице *www.sqlite.org/lang_expr.html*. В этом разделе мы приведем лишь несколько примеров.

Допустим, вы хотите узнать среднюю продолжительность песен в таблице musicTracks. Можно извлечь все записи и воспользоваться Ac-

tionScript, чтобы рассчитать среднее значение. Но с помощью функции SQL `avg()` это можно сделать гораздо проще. Нужно лишь применить эту функцию в команде SELECT следующим образом:

```
SELECT avg(length) FROM musicTracks
```

Эта команда возвратит результирующий набор из одной колонки под именем avg(length), имеющей значение 267.5. Имя колонки avg(length) выглядит не очень изящно. Если мы хотим, чтобы запрос возвратил колонку с более красивым именем, можно задать для нее псевдоним с помощью ключевого слова `AS`. Псевдоним позволяет обращаться к любым элементам списка колонок, независимо от того, являются ли они функциями. Например, можно воспользоваться псевдонимом averageLength:

```
SELECT avg(length) AS averageLength FROM musicTracks
```

А как быть, если вы захотите узнать среднюю продолжительность не всех песен из таблицы, а по каждому из исполнителей отдельно? Это можно сделать с помощью выражения `GROUP BY`. Выражение `GROUP BY` позволяет задать колонку, по которой нужно фильтровать применение обобщающей функции (такой, как `avg()`):

```
SELECT artist, avg(length) as averageLength
➥FROM musicTracks GROUP BY artist
```

Результаты выполнения этой команды представлены в табл. 5.9.

Таблица 5.9. С помощью выражения GROUP BY можно влиять на применение обобщающей функции

Artist	averageLength
Oingo Boingo	243
The Psychedelic Furs	229
Thomas Dolby	312
Wang Chung	272.5
XTC	270

Если вы хотите дополнительно фильтровать возвращаемые результаты, можно добавить в команду выражение HAVING. Выражение HAVING позволяет задать выражение, которое определяет, какие результаты нужно возвращать. Например, ниже приводится команда, которая возвращает только результаты со средней длиной, большей 250:

```
SELECT artist, avg(length) as averageLength FROM musicTracks
➥GROUP BY artist HAVING averageLength > 250
```

Результаты представлены в табл. 5.10.

На этом завершается наше обсуждение работы с функциями в командах SQL.

Таблица 5.10. Результаты со средней продолжительностью, большей 250

artist	averageLength
Thomas Dolby	312
Wang Chung	272.5
XTC	270

Примечание

До сих пор мы обсуждали только работу с данными из одной таблицы. Во многих приложениях средней и высокой сложности требуется работать с данными из нескольких таблиц. Такие вопросы выходят за рамки этой книги, но вы можете найти хорошие источники информации по данной теме, включая уже упомянутый ресурс *www.w3schools.org/sql*.

После рассмотрения основ работы с SQL мы можем обратиться к конкретным особенностям применения SQL в приложениях AIR.

5.3. Создание и открытие баз данных

Представьте, что вы хотите арендовать фильм в местном магазине видеопродукции. Вы хотите взять фильм, в котором играет ваш любимый актер Хамфри Богарт. Требуется выяснить, какие фильмы с его участием есть в наличии. Как это сделать? Можно снять трубку и позвонить в магазин. Когда вам ответят, можно спросить, есть ли у них в наличии фильмы с Хамфри Богартом. Получив ответ, можете поблагодарить служащего магазина и положить трубку. Поверьте, эта процедура вполне аналогична работе с базой данных. При работе с базой данных вы прежде всего должны установить с ней соединение. После того как соединение установлено, можно выполнять команды (например, делать запросы), а по завершении работы отключиться от базы данных. В этом разделе мы рассмотрим первый и последний этапы работы: открытие и закрытие соединений с базой данных.

Базы данных SQLite хранятся в файлах. В главе 3 вы узнали, как работать с файлами, и при работе с базами данных пригодятся некоторые из описанных там приемов. Первое, что нужно для работы с базой данных, – это создать объект `File`, связанный с файлом используемой базы данных. Как будет показано ниже, AIR может автоматически создать файл при попытке открытия соединения с базой данных. Поэтому объект `File` не обязательно должен указывать на уже существующий файл. Необходимо лишь создать объект `File`, который указывает на файл, где хранятся или должны храниться данные. В следующем примере создается объект `File`, указывающий на файл с именем example.db, лежащий в каталоге хранилища приложения:

```
var databaseFile:File = File.applicationStorageDirectory.resolvePath(
➥"example.db");
```

После создания объекта File, который указывает на файл, в котором должны храниться данные, нужно создать объект flash.data.SQLConnection. Объект SQLConnection служит для создания соединения с файлом базы данных, и для выполнения всех команд SQL нужна ссылка на объект SQLConnection. Конструктор SQLConnection не требует параметров. Ниже приводится пример создания нового объекта SQLConnection:

```
var connection:SQLConnection = new SQLConnection();
```

Следующим шагом будет подключение этого объекта к файлу базы данных. Открыть соединение объекта SQLConnection с файлом данных можно в синхронном или асинхронном режиме. Синхронное открытие осуществляется с помощью метода open(). Если соединение открывается синхронно, то все команды SQL в этом соединении будут выполняться синхронным образом. Обычно не рекомендуется этого делать по тем же самым причинам, по которым обычно не рекомендуется выполнять синхронные операции, если их выполнение может оказаться достаточно долгим. (Подробности см. в главе 3.) Гораздо лучше открыть соединение асинхронно с помощью метода openAsync(). Обоим методам требуется один и тот же параметр: ссылка на объект File, указывающий на файл базы данных. Например, следующий код открывает соединение с файлом базы данных, который мы создали ранее в этом разделе:

```
connection.openAsync(databaseFile);
```

Открывая соединение, вы можете указать режим, в котором оно должно действовать. Таких режимов три: чтение, обновление и создание. Режим чтения означает, что данное соединение может быть использовано только для извлечения из базы существующих данных. Режимы обновления и создания допускают как чтение, так и запись данных, но режим создания создаст файл базы данных, если его не существует, тогда как режим обновления в этом случае даст сбой. По умолчанию устанавливается режим создания. Если вы хотите установить режим явным образом, передайте методу open() или openAsync() второй параметр в виде одной из трех констант flash.data.SQLMode: READ, UPDATE или CREATE. Например, следующий код открывает соединение в режиме чтения:

```
connection.openAsync(databaseFile, SQLMode.READ);
```

При открытии соединения в асинхронном режиме нужно перехватить событие open, прежде чем выполнять какие-либо команды SQL. Событие open имеет тип flash.events.SQLEvent.

Завершив работу с базой данных, нужно закрыть соединение с ней. Для этого нужно всего лишь вызвать метод close().

5.4. Выполнение команд SQL

В предыдущем разделе мы обсудили три базовых этапа в работе с базой данных: открытие соединения, выполнение команд и закрытие соеди-

нения. Мы воспользовались аналогией с выяснением того, какие фильмы есть в вашем местном магазине видеопродукции. Когда вы спрашиваете у служащего магазина, какие фильмы есть в магазине, это аналогично выполнению команд SQL в базе данных. Это и будет нашей следующей темой: как выполнять команды SQL после установления соединения с базой данных.

Все команды SQL (`CREATE TABLE`, `INSERT`, `DELETE`, `SELECT` и т. д.) должны выполняться с помощью экземпляра класса `flash.data.SQLStatement`. В нескольких последующих разделах мы расскажем, как работать с объектами `SQLStatement`.

5.4.1. Создание команд SQL

Когда вы хотите выполнить команду SQL, прежде всего нужно создать новый объект `SQLStatement`, воспользовавшись конструктором. Этот конструктор не требует параметров. Таким образом, следующий код демонстрирует создание нового объекта `SQLStatement`:

```
var statement:SQLStatement = new SQLStatement();
```

Когда нужно выполнить команду, вы должны сообщить объекту `SQLStatement`, каким объектом `SQLConnection` воспользоваться. Это делается путем задания свойства `sqlConnection`, как в следующем фрагменте:

```
statement.sqlConnection = connection;
```

Затем нужно задать команду SQL, которую вы хотите выполнить. Это можно сделать путем присваивания строкового значения свойству `text` объекта `SQLStatement`. Например, следующий код записывает команду `SELECT` в свойство `text` объекта `SQLStatement`:

```
statement.text = "SELECT DISTINCT album FROM musicTracks";
```

Вот и все, что нужно, чтобы подготовиться к выполнению простой команды SQL. Ниже мы рассмотрим, как эту команду выполнить.

5.4.2. Выполнение команд SQL

Чтобы выполнить команду SQL, достаточно вызвать метод `execute()`. При синхронном вызове метода `execute()` он может выполниться успешно (и результаты, если они есть, становятся сразу доступны) либо возникает ошибка. Если выполнить `execute()`, когда установлено асинхронное соединение, результат не будет получен немедленно. Результатом будет одно из двух событий: событие `result` или событие `error`. При вызове метода `execute()` без параметров необходимо зарегистрировать обработчики этих событий непосредственно в объекте `SQLStatement`. Например, так регистрируется обработчик события `result`:

```
statement.addEventListener(SQLEvent.RESULT, statementResultHandler);
```

Кроме того, можно обрабатывать результат выполнения команды SQL с помощью объекта `flash.net.Responder`. В этом случае не нужно обра-

батывать события result или error в объекте SQLStatement. Вместо этого вы передаете объект Responder методу execute() в качестве параметра. Метод execute() может принимать два параметра. Первый параметр управляет постраничным выводом результатов, и мы рассмотрим его в разделе 5.4.5. Сейчас же мы воспользуемся его значением по умолчанию, равным -1, что означает вывод всех результатов. Вторым параметром является объект Responder. Ниже показано, как вызывать метод execute() с использованием объекта Responder:

```
statement.execute(-1, new Responder(statementResultHandler,
➥statementErrorHandler));
```

Это все, что относится к выполнению команд SQL. Теперь посмотрим, как получить результаты выполнения команды SELECT.

5.4.3. Обработка результатов SELECT

Практически для любых команд необходимо обрабатывать события error и result. Обычно важно, чтобы приложение могло выяснить, была ли команда выполнена успешно или возникла ошибка. Тогда приложение может адекватным образом реагировать. Например, при возникновении ошибки можно известить пользователя или попытаться выполнить команду снова. Команды SELECT относятся к тому типу, который всегда требует обработки результатов. В конце концов, если не обрабатывать результат SELECT, то как узнать, когда запрошенные вами данные уже доступны?

При обработке результата команды SELECT необходимо извлечь полученный набор данных, для чего нужно вызвать метод getResult() объекта SQLStatement с только что выполненной командой SQL. Метод getResult() возвращает объект flash.data.SQLResult, у которого есть свойство data, представляющее собой массив возвращаемых данных. По умолчанию элементы массива имеют тип Object, и у каждого из них есть свойства с такими же именами, как у колонок в возвращаемом наборе данных. Например, в следующем коде приведен метод, который обрабатывает событие result и с помощью метода trace() выводит названия альбомов на консоль или в окно вывода. Будем предполагать, что этот метод обрабатывает результат выполнения команды SQL SELECT DISTINCT album FROM musicTracks:

```
private function statementResultHandler(event:SQLEvent):void {
   var statement:SQLStatement = event.target as SQLStatement;
   var result:SQLResult = statement.getResult();
   if(result != null && result.data != null) {
      var dataset:Array = result.data;
      for(var i:Number = 0; i < dataset.length; i++) {
         trace(dataset[i].album);
      }
   }
}
```

Как только что было сказано, по умолчанию все результаты представляют собой общие экземпляры Object. Но в следующем разделе мы узнаем, что возможны другие варианты.

5.4.4. Типизация результатов

Иногда при записи данных в базу происходит сериализация пользовательских типов ActionScript. Например, при записи данных в таблицу musicTracks можно записывать экземпляры пользовательского класса ActionScript с именем MusicTrack, созданного в вашем приложении. У класса MusicTrack могут быть такие свойства, как id, title, artist, album, length и originalReleaseYear. Эти объекты записываются в базу данных путем сохранения их свойств в колонках с такими же именами. Это типичная процедура. Поэтому при извлечении данных с помощью команды SELECT было бы удобным, если бы среда AIR смогла автоматически преобразовать результаты в объекты типа MusicTrack. И AIR действительно может это сделать.

Чтобы заставить AIR автоматически преобразовывать результаты в объекты заданного типа, достаточно задать свойство itemClass объекта SQLStatement, прежде чем вызывать метод execute(). Свойству itemClass нужно присвоить ссылку на класс объектов, в которые вы хотите автоматически преобразовывать результаты с помощью AIR. Например, следующий код требует от AIR приводить результаты выполнения объекта SQLStatement к типу MusicTrack:

```
statement.itemClass = MusicTrack;
```

Этот способ действует, когда имена колонок в результирующем наборе и имена свойств класса совпадают друг с другом. При этом не обязательно, чтобы имена колонок в таблице базы данных совпадали с именами свойств класса, потому что с помощью псевдонимов можно отобразить имена колонок в совсем другие имена в результирующем наборе.

5.4.5. Постраничный вывод результатов

Мы пока рассматривали, как получить все результаты команды SELECT сразу. Однако, если наборы данных достаточно велики, более практичным может оказаться получение только тех записей, которые требуются в данный момент. Например, если есть вероятность, что запрос возвратит несколько сот тысяч записей, но ваше приложение может одновременно работать только с 100 из них (скажем, показывая пользователю), нет смысла извлекать все записи, пока пользователь не запросит их явным образом. Сообщить AIR, какое максимальное количество записей должно извлекаться при выполнении метода execute(), можно путем передачи этого количества в качестве первого параметра метода. По умолчанию установлено значение -1, и это означает, что AIR извлечет все записи, однако положительное число укажет AIR, сколько именно записей должно быть извлечено. Например, в следующем примере указывается, что среда AIR должна возвращать не более 20 записей:

```
statement.execute(20);
```

Если вы ограничили количество возвращаемых записей, необходимо иметь возможность «пролистывать» оставшиеся записи результата. Для этого не нужно снова вызывать `execute()`. Вместо этого вызывается метод `next()` того же объекта `SQLStatement`. Метод `next()` принимает те же параметры, что и `execute()`, и вызывает те же события. Разница в том, что `execute()` всегда извлекает только первую группу записей, тогда как `next()` извлекает очередную группу, следующую за последней извлеченной. Например, можно вызвать `execute()` со значением 20, и если затем вызвать `next()` со значением 20, AIR извлечет записи 21–40 (если они есть). Результаты вызова метода `next()` извлекаются так же, как результаты метода `execute()`: когда возникает событие result, нужно вызвать `getResult()` для объекта `SQLStatement`.

При каждом обращении к `next()` в очередь помещается новый объект `SQLResult`, и `getResult()` всегда возвращает первый `SQLResult`, стоящий в очереди, пока очередь не опустеет, и тогда `getResult()` возвратит null.

5.4.6. Параметрические команды SQL

Команды SQL часто содержат переменные величины. Например, вам нужно ввести в таблицу новую запись о музыкальном треке по данным, которые ввел пользователь. Для разработчика Flash или Flex такая задача не вызывает трудностей: достаточно составить строку из переменных, как в следующем примере:

```
statement.text = "INSERT INTO musicTracks(album, artist) VALUES('" +
➥albumInput.text + "', '" + artistInput.text + "')";
```

Однако такой подход открывает возможность для злоупотреблений или случайных ошибок, поскольку дает пользователю возможность ввести свой код в команду SQL. Более правильным подходом является параметризация команд SQL с помощью встроенной технологии AIR.

Чтобы параметризовать команду SQL, воспользуйтесь символами `@` или `:` в качестве начальных для обозначения параметров в команде SQL, которую нужно присвоить свойству `text` объекта `SQLStatement`. Например, предыдущий пример можно переписать так:

```
statement.text = "INSERT INTO musicTracks(album, artist) VALUES(@album,
➥@artist)";
```

Затем воспользуйтесь свойством `parameters` объекта `SQLStatement`, чтобы определить значения параметров. Свойство `parameters` является ассоциативным массивом, в который можно добавлять свойства и их значения. Свойства – это имена параметров, используемых в команде SQL, а значениями должны быть те величины, которые нужно использовать вместо параметров:

```
statement.parameters["@album"] = albumInput.text;
statement.parameters["@artist"] = artistInput.text;
```

AIR автоматически проверит, чтобы значения, присвоенные свойству parameters, не вызвали никаких нежелательных проблем. Рассмотренные нами параметры были именованными. При желании вместо именованных можно использовать упорядоченные параметры. Такие параметры обозначаются в команде SQL символом «?», например:

```
statement.text = "INSERT INTO musicTracks(album, artist) VALUES(?, ?)";
```

После этого можно задать упорядоченный массив значений для свойства parameters:

```
statement.parameters[0] = albumInput.text;
statement.parameters[1] = artistInput.text;
```

Помимо удаления опасных символов и кода SQL такая параметризация (в случае как именованных, так и упорядоченных параметров) может иногда повысить эффективность приложения. При первом выполнении команды SQL приложение AIR должно ее скомпилировать, что занимает некоторое время. Последующие вызовы команды выполняются быстрее. Однако если изменить свойство text, то команду нужно скомпилировать заново. При параметризации команды можно менять значения параметров, и при этом AIR не потребуется заново компилировать команду.

5.4.7. Транзакции

Обычно выполнение команды происходит автономно, что приемлемо во многих ситуациях. Но в некоторых случаях предпочтительнее объединить команды в группу, которая выполняется как единое целое. Стандартным примером является задача, в которой в таблицу добавляется запись, читается ее идентификатор и этот идентификатор помещается в запись другой таблицы. Здесь требуется выполнить по крайней мере две команды SQL, но логически они составляют одну группу команд, которые желательно выполнить вместе. При этом в случае возникновения в какой-то момент ошибки должна быть возможность отменить все сделанные изменения. Посмотрим, что может произойти, если выполнять все команды обычным образом:

- Первая команда успешно выполняется и вставляет запись
- Вторая команда дает сбой

В результате в базе данных сохраняется запись, созданная первой командой, хотя логически ее не должно быть, поскольку группа команд в целом не была успешно выполнена. Можно самому написать код для обработки такой исключительной ситуации, который будет удалять первую запись, если вторая команда окажется невыполненной. Однако есть гораздо более простой и эффективный способ решения таких задач с помощью встроенной в AIR функции транзакций команд SQL.

Транзакция позволяет объединить несколько команд, выполняемых с помощью одного и того же объекта SQLConnection. Чтобы создать транзакцию, нужно действовать следующим образом:

1. Вызвать метод begin() объекта SQLConnection.
2. После этого выполнить команды SQL с помощью SQLConnection. Выполнять методы, как обычно.
3. При возникновении в какой-либо момент ошибки вызвать метод rollback() объекта SQLConnection.
4. Если все команды выполнятся успешно, вызвать метод commit() объекта SQLConnection, который запишет результаты на диск.

Как видно из перечисленного, все объекты SQLStatement, выполняющиеся для конкретного объекта SQLConnection, объединяются вместе, начиная с вызова begin(). Все эти объекты SQLStatement выполняются только в памяти. Транзакция продолжается, пока не случится одно из двух: будет вызван метод rollback() или метод commit(). Тот и другой прекращают транзакцию. Метод rollback() отменяет все команды, уже выполненные в ходе транзакции, и их результаты не записываются на диск. Например, если все четыре первые команды транзакции являются командами INSERT, ни одна из четырех записей не будет записана в базу данных, если вызвать метод rollback(). Напротив, метод commit() останавливает транзакцию и записывает результаты на диск.

Другое достоинство транзакций в том, что можно заблокировать базу данных на время осуществления транзакции. Это гарантирует, что никакой другой процесс не модифицирует базу данных и не сможет повлиять на результаты вашей операции. Это может пригодиться, если, например, есть два участка кода, пытающиеся прочесть и модифицировать одну и ту же запись таблицы. Если бы они могли читать и записывать одновременно, изменения, внесенные одним из процессов, могли бы оказаться потеряны.

Заблокировать базу данных можно с помощью дополнительного параметра метода begin(). Он дает возможность сделать базу данных открытой другим соединениям только для чтения либо полностью недоступной. Кроме того, можно отложить блокировку базы данных до того момента, когда вашей транзакции действительно потребуется выполнить чтение или запись в базе данных. В качестве значений параметра используются следующие константы класса flash.data.SQLTransactionLockType: DEFERRED, EXCLUSIVE и IMMEDIATE. Их смысл описан в табл. 5.11. Блокировка базы данных в методе begin() может осуществляться так:

```
connection.begin(SQLTransactionLockType.IMMEDIATE);
```

Рассказав об основах выполнения команд SQL, построим простое приложение, в котором они используются.

Таблица 5.11. Типы блокировок в транзакциях

Константа	Описание
SQLTransactionLockType.DEFERRED	Заблокировать базу данных при первой операции чтения или записи.
SQLTransactionLockType.EXCLUSIVE	Сразу заблокировать базу данных. Ни одно другое соединение не может выполнять чтение или запись в базе данных.
SQLTransactionLockType.IMMEDIATE	Сразу заблокировать базу данных. Ни одно другое соединение не может выполнять запись в базу данных, но чтение допускается.

5.5. Приложение ToDo

В этом разделе мы построим приложение, в котором применяются все полученные нами знания о базах данных. Приложение ToDo дает пользователям возможность записывать в базу данных элементы списка дел, подлежащих выполнению, редактировать их и при желании удалять. Приложение также показывает список всех текущих дел. Его внешний вид приведен на рис. 5.9.

Чтобы создать это приложение, выполните следующие действия:

1. Постройте класс модели данных для элемента списка.
2. Создайте компоненту для отображения каждого элемента.

Рис. 5.9. *Приложение ToDo перечисляет список текущих дел и позволяет добавлять в него новые элементы, а также редактировать и удалять существующие*

3. Создайте базу данных.
4. Создайте форму для ввода данных.
5. Добавьте команды SQL.

В следующем разделе мы займемся первым шагом.

5.5.1. Создание модели данных элемента списка текущих дел

Мы создадим простой класс ActionScript, моделирующий пункт списка дел. В нашем приложении у каждого элемента списка будут следующие свойства: ID, название, описание, приоритет и дата выполнения. Листинг 5.1 демонстрирует класс ToDoItem.

Листинг 5.1. Класс ToDoItem – модель данных для списка дел

```
package com.manning.todolist.data {
   import flash.events.Event;
   import flash.events.EventDispatcher;

   public class ToDoItem extends EventDispatcher {

      private var _id:int;
      private var _name:String;
      private var _description:String;
      private var _priority:int;
      private var _mustBeDoneBy:Date;

      [Bindable(event="idChanged")]
      public function set id(value:int):void {
         _id = value;
         dispatchEvent(new Event("idChanged"));
      }

      public function get id():int {
         return _id;
      }

      [Bindable(event="nameChanged")]
      public function set name(value:String):void {
         _name = value;
         dispatchEvent(new Event("nameChanged"));
      }

      public function get name():String {
         return _name;
      }

      [Bindable(event="descriptionChanged")]
      public function set description(value:String):void {
         _description = value;
         dispatchEvent(new Event("descriptionChanged"));
      }

      public function get description():String {
```

```
            return _description;
        }

        [Bindable(event="priorityChanged")]
        public function set priority(value:int):void {
            _priority = value;
            dispatchEvent(new Event("priorityChanged"));
        }

        public function get priority():int {
            return _priority;
        }

        [Bindable(event="mustBeDoneByChanged")]
        public function set mustBeDoneBy(value:Date):void {
            _mustBeDoneBy = value;
            dispatchEvent(new Event("mustBeDoneByChanged"));
        }

        public function get mustBeDoneBy():Date {
            return _mustBeDoneBy;
        }

        public function ToDoItem() {
        }

    }
}
```

В классе `ToDoItem` нет ничего особенного. Он просто создает закрытые свойства и функции для их чтения и изменения. Мы также добавили теги метаданных `[Bindable]` для привязки к источникам данных во Flex.

5.5.2. Создание компоненты элемента списка дел

Как видно из рис. 5.9, каждый элемент отображается в списке, расположенном в верхней части окна приложения. Для этого мы создадим компоненту MXML. Она будет называться `ToDoListRenderer`, и мы сохраним ее в каталоге com/manning/todolist/ui/ToDoListRenderer.mxml. Код для этого приведен в листинге 5.2.

Листинг 5.2. Компонента ToDoListRenderer показывает элемент списка задач

```
<?xml version="1.0" encoding="utf-8"?>
<mx:VBox xmlns:mx="http://www.adobe.com/2006/mxml"
    borderStyle="solid" width="100%"
        toolTip="{'description: ' + data.description
            + '\nmust by done by: ' + data.mustBeDoneBy}">
    <mx:HBox>
        <mx:Label text="{data.name}" width="200" />
        <mx:Label text="priority {data.priority}" />
        <mx:Button label="Delete"
        ➥click="dispatchEvent(new Event('delete'));" />
        <mx:Button label="Edit"
```

Добавить подсказку

```
        ➥click="dispatchEvent(new Event('edit'));" />
    </mx:HBox>
</mx:VBox>
```

Как можно видеть, этот код всего лишь показывает значения из свойства `data`. Предполагается, что свойству `data` всегда присваивается объект `ToDoItem`.

5.5.3. Создание базы данных

Теперь нам нужно создать базу данных. В этом примере база данных простая. В ней всего одна таблица с колонками id, priority, name, description и mustBeDoneBy. Обратите внимание, что колонки совпадают со свойствами класса ToDoItem. Это позволит в дальнейшем легко получать данные из таблицы с помощью объектов специального типа.

В этом приложении мы облегчили себе работу, поместив весь код SQL в файл MXML приложения. Поэтому мы поместим код создания базы данных в файл MXML приложения, который назовем ToDo.mxml. Сначала поместим в ToDo.mxml код, показанный на рис. 5.3.

Листинг 5.3. Файл ToDo.mxml создает базу данных и содержащуюся в ней таблицу

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
      <![CDATA[

         private var _connection:SQLConnection;

         private function creationCompleteHandler():void {
            var file:File =
File.applicationStorageDirectory.resolvePath("database.db");
            _connection = new SQLConnection();
            _connection.addEventListener(SQLEvent.OPEN, openHandler);
            _connection.openAsync(file, SQLMode.CREATE);
         }

         private function openHandler(event:SQLEvent):void {
            var sql:SQLStatement = new SQLStatement();
            sql.sqlConnection = _connection;
            sql.text = "CREATE TABLE IF NOT EXISTS todo(" +
                    "id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    "priority INTEGER, " +
                    "name TEXT, " +
                    "description TEXT, " +
                    "mustBeDoneBy DATE)";
            sql.execute();
         }

      ]]>
   </mx:Script>
</mx:WindowedApplication>
```

Объект соединения

Перехват события open

Асинхронное открытие соединения

Создание таблицы

Обратите внимание на использование здесь команды `CREATE TABLE IF NOT EXISTS` с целью создать таблицу, только если ее еще не существует. После создания таблицы (если ее не было до этого) можно разместить код для вставки, обновления, извлечения и удаления элементов.

5.5.4. Создание формы для ввода данных

Теперь создадим форму для ввода данных, с помощью которой пользователь сможет вводить данные для пункта списка дел. Затем мы добавим состояние просмотра, в котором пользователь с помощью той же формы сможет редактировать данные. В состоянии редактирования метки на форме имеют другой вид, а кнопки вызывают другие методы. Листинг 5.4. демонстрирует обновленный код.

Листинг 5.4. Добавление к приложению формы ввода данных

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
      <![CDATA[

         private var _connection:SQLConnection;

         private function creationCompleteHandler():void {
            var file:File =
            ➥File.applicationStorageDirectory.resolvePath("database.db");
            _connection = new SQLConnection();
            _connection.addEventListener(SQLEvent.OPEN, openHandler);
            _connection.openAsync(file, SQLMode.CREATE);
         }

         private function openHandler(event:SQLEvent):void {
            var sql:SQLStatement = new SQLStatement();
            sql.sqlConnection = _connection;
            sql.text = "CREATE TABLE IF NOT EXISTS todo(" +
                    "id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    "priority INTEGER, " +
                    "name TEXT, " +
                    "description TEXT, " +
                    "mustBeDoneBy DATE)";
            sql.addEventListener(SQLEvent.RESULT, selectToDoItems);
            sql.execute();
         }

         private function addItem(event:MouseEvent):void {
         }
         private function updateItem(event:MouseEvent):void {
         }

      ]]>
   </mx:Script>
   <mx:VBox width="100%" height="100%">
      <mx:Form>
```

❶ Добавление пункта

Редактирование пункта ❷

❸ Создание формы

```
            <mx:FormHeading id="formHeading" label="New Item" />
            <mx:FormItem label="Name">
               <mx:TextInput id="itemName" text="{_selectedItem.name}" />
            </mx:FormItem>
            <mx:FormItem label="Description">
               <mx:TextInput id="itemDescription"
                  text="{_selectedItem.description}" />
            </mx:FormItem>
            <mx:FormItem label="Priority">
               <mx:ComboBox id="itemPriority"
                  selectedItem="{_selectedItem.priority}">
                  <mx:dataProvider>
                     <mx:ArrayCollection>
                        <mx:Number>1</mx:Number>
                        <mx:Number>2</mx:Number>
                        <mx:Number>3</mx:Number>
                        <mx:Number>4</mx:Number>
                     </mx:ArrayCollection>
                  </mx:dataProvider>
               </mx:ComboBox>
            </mx:FormItem>
            <mx:FormItem label="Must Be Done By">
               <mx:DateField id="itemMustBeDoneBy"
               ➥selectedDate="{_selectedItem.mustBeDoneBy}" />
            </mx:FormItem>
            <mx:FormItem>
               <mx:Button id="formButton" label="Add"
                  click="addItem(event);" />
            </mx:FormItem>
         </mx:Form>
      </mx:VBox>
      <mx:states>
         <mx:State name="Edit">
            <mx:SetProperty target="{formHeading}"
               name="label" value="Edit Item" />
            <mx:SetProperty target="{formButton}"
               name="label" value="Update" />
            <mx:SetEventHandler target="{formButton}"
               name="click" handlerFunction="updateItem" />
         </mx:State>
      </mx:states>
   </mx:WindowedApplication>
```

❹ Определить состояние редактирования

❺ Изменить заголовок формы

❻ Изменить метку кнопки

❼ Изменить обработчик нажатия на кнопку

В этом коде мы определяем форму ❸ с полями для ввода всех значений пункта списка. По умолчанию состояние формы соответствует вводу нового пункта. Мы также определили состояние для редактирования ❹ пункта. В этом состоянии нужно изменить заголовок формы ❺ и метку кнопки ❻, чтобы они соответствовали новому действию. Затем мы устанавливаем новый обработчик щелчка по кнопке ❼. К этому моменту мы задали в коде методы для обработки щелчка по кнопке в случае

добавления элемента ❶ и редактирования элемента ❷, но не написали код, который выполняет необходимые в каждом случае команды SQL.

Ими мы теперь и займемся.

5.5.5. Добавление команд SQL

Теперь мы напишем код, который добавляет в базу данных новые пункты, а также код, который редактирует и извлекает имеющиеся данные. В листинге 5.5 показан код, который мы добавим в ToDo.mxml.

Листинг 5.5. Код для добавления, редактирования и извлечения записей

```
<mx:Script>
  <![CDATA[
     import com.manning.todolist.data.ToDoItem;
     import com.manning.todolist.ui.ToDoListRenderer;

     private var _connection:SQLConnection;

     [Bindable]
     private var _selectedItem:ToDoItem;

     private function creationCompleteHandler():void {
        var file:File =
        ➥File.applicationStorageDirectory.resolvePath("database.db");
        _connection = new SQLConnection();
        _connection.addEventListener(SQLEvent.OPEN, openHandler);
        _connection.openAsync(file, SQLMode.CREATE);
     }

     private function openHandler(event:SQLEvent):void {
        var sql:SQLStatement = new SQLStatement();
        sql.sqlConnection = _connection;
        sql.text = "CREATE TABLE IF NOT EXISTS todo(" +
                "id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                "priority INTEGER, " +
                "name TEXT, " +
                "description TEXT, " +
                "mustBeDoneBy DATE)";
        sql.addEventListener(SQLEvent.RESULT, selectToDoItems);
        sql.execute();
     }

     private function addItem(event:MouseEvent):void {
        var sql:SQLStatement = new SQLStatement();
        sql.sqlConnection = _connection;
        sql.text = "INSERT INTO todo(priority, name, " +
                "description, mustBeDoneBy)" +
                "VALUES(@priority, @name, " +
                "@description, @mustBeDoneBy)";
        sql.parameters["@priority"] = itemPriority.value;
        sql.parameters["@name"] = itemName.text;
        sql.parameters["@description"] = itemDescription.text;
```

Ссылка на выбранный пункт

Извлечение пунктов после выполнения ❶

❷ Создать параметризованный запрос SQL

```
            sql.parameters["@mustBeDoneBy"] =
            ➥itemMustBeDoneBy.selectedDate;
            sql.addEventListener(SQLEvent.RESULT, selectToDoItems);
            sql.execute();
        }

        private function updateItem(event:MouseEvent):void {
            var sql:SQLStatement = new SQLStatement();
            sql.sqlConnection = _connection;
            sql.text = "UPDATE todo SET priority = @priority, " +
                    "name = @name, description = @description, " +
                    "mustBeDoneBy = @mustBeDoneBy WHERE id = @id";
            sql.parameters["@priority"] = itemPriority.value;
            sql.parameters["@name"] = itemName.text;
            sql.parameters["@description"] = itemDescription.text;
            sql.parameters["@mustBeDoneBy"] =
            ➥itemMustBeDoneBy.selectedDate;
            sql.parameters["@id"] = _selectedItem.id;
            sql.addEventListener(SQLEvent.RESULT, selectToDoItems);
            sql.execute();
            currentState = "";
            _selectedItem = null;
        }

        private function selectToDoItems(event:SQLEvent = null):void {
            var sql:SQLStatement = new SQLStatement();
            sql.sqlConnection = _connection;
            sql.text = "SELECT id, priority, name, " +
                    "description, mustBeDoneBy " +
                    "FROM todo ORDER BY mustBeDoneBy, priority";
            sql.itemClass = ToDoItem;
            sql.addEventListener(SQLEvent.RESULT, selectHandler);
            sql.execute();
        }

        private function selectHandler(event:SQLEvent):void {
        }

    ]]>
</mx:Script>
```

❷ Создать параметризованный запрос SQL

❸ Задать тип результатов

❹ Обработать результаты SELECT

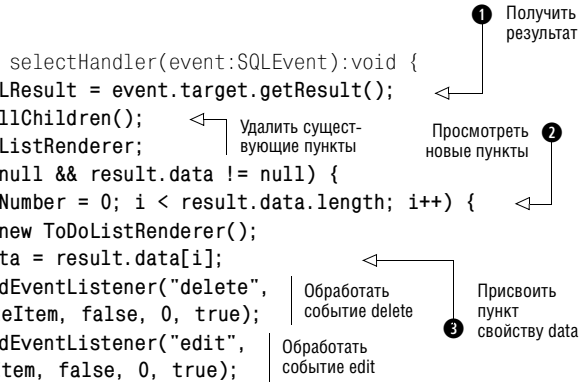
В этом коде мы добавляем в методы `addItem()` и `updateItem()` параметризованные команды SQL ❷, которые используют данные формы ввода. Мы также добавляем метод, в котором есть команда SQL для извлечения из базы данных всех пунктов. Мы также добавляем обработчик события при запуске ❶, который вызывает метод `selectoToDoItems()`, когда AIR создает таблицу базы данных или проверяет ее существование. Кроме того, мы сообщили AIR, что в качестве типа для всех элементов, возвращаемых командой `SELECT`, будет использоваться `ToDoItem` ❸. На данный момент метод `selectHandler()` ❹ пустой. Мы хотим показывать возвращаемые данные с помощью ранее созданной компоненты MXML. Поэтому нужно добавить в код макета контейнер, в который

будут помещаться пункты списка. Листинг 5.6 демонстрирует вид ToDo.mxml после добавления контейнера и обновления `selectHandler()`.

Листинг 5.6. Добавление контейнера для показа и обработки результатов SELECT

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler();">
    <mx:Script>
      <![CDATA[
         import com.manning.todolist.data.ToDoItem;
         import com.manning.todolist.ui.ToDoListRenderer;

         private var _connection:SQLConnection;

         [Bindable]
         private var _selectedItem:ToDoItem;
         private function creationCompleteHandler():void {
            var file:File =
            ➥File.applicationStorageDirectory.resolvePath("database.db");
            _connection = new SQLConnection();
            _connection.addEventListener(SQLEvent.OPEN, openHandler);
            _connection.openAsync(file, SQLMode.CREATE);
         }

         private function openHandler(event:SQLEvent):void {
            var sql:SQLStatement = new SQLStatement();
            sql.sqlConnection = _connection;
            sql.text = "CREATE TABLE IF NOT EXISTS todo(" +
                    "id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    "priority INTEGER, " +
                    "name TEXT, " +
                    "description TEXT, " +
                    "mustBeDoneBy DATE)";
            sql.addEventListener(SQLEvent.RESULT, selectToDoItems);
            sql.execute();
         }

         private function addItem(event:MouseEvent):void {
            var sql:SQLStatement = new SQLStatement();
            sql.sqlConnection = _connection;
            sql.text = "INSERT INTO todo(priority, name, ";
                    "description, mustBeDoneBy)" +
                    "VALUES(@priority, @name, " +
                    "@description, @mustBeDoneBy)";
            sql.parameters["@priority"] = itemPriority.value;
            sql.parameters["@name"] = itemName.text;
            sql.parameters["@description"] = itemDescription.text;
            sql.parameters["@mustBeDoneBy"] =
            ➥itemMustBeDoneBy.selectedDate;
            sql.addEventListener(SQLEvent.RESULT, selectToDoItems);
            sql.execute();
         }
```

```
private function updateItem(event:MouseEvent):void {
   var sql:SQLStatement = new SQLStatement();
   sql.sqlConnection = _connection;
   sql.text = "UPDATE todo SET priority = @priority, " +
           "name = @name, " +
           "description = @description, " +
           "mustBeDoneBy = @mustBeDoneBy " +
           "WHERE id = @id";
   sql.parameters["@priority"] = itemPriority.value;
   sql.parameters["@name"] = itemName.text;
   sql.parameters["@description"] = itemDescription.text;
   sql.parameters["@mustBeDoneBy"] =
   ➥itemMustBeDoneBy.selectedDate;
   sql.parameters["@id"] = _selectedItem.id;
   sql.addEventListener(SQLEvent.RESULT, selectToDoItems);
   sql.execute();
   currentState = "";
   _selectedItem = null;
}

private function selectToDoItems(event:SQLEvent = null):void {
   var sql:SQLStatement = new SQLStatement();
   sql.sqlConnection = _connection;
   sql.text = "SELECT id, priority, name, description, " +
            "mustBeDoneBy FROM todo " +
            "ORDER BY mustBeDoneBy, priority";
   sql.itemClass = ToDoItem;
   sql.addEventListener(SQLEvent.RESULT, selectHandler);
   sql.execute();
}

private function selectHandler(event:SQLEvent):void {
   var result:SQLResult = event.target.getResult();
   items.removeAllChildren();
   var item:ToDoListRenderer;
   if(result != null && result.data != null) {
      for(var i:Number = 0; i < result.data.length; i++) {
         item = new ToDoListRenderer();
         item.data = result.data[i];
         item.addEventListener("delete",
         ➥deleteItem, false, 0, true);
         item.addEventListener("edit",
         ➥editItem, false, 0, true);
         items.addChild(item);
      }
   }
}

private function deleteItem(event:Event):void {
}

private function editItem(event:Event):void {
}
```



```
    ]]>
</mx:Script>
<mx:VBox width="100%" height="100%">
    <mx:VBox id="items" width="100%" height="50%"
        backgroundColor="#FFFFFF" />
    <mx:Form>
        <mx:FormHeading id="formHeading" label="New Item" />
        <mx:FormItem label="Name">
            <mx:TextInput id="itemName" text="{_selectedItem.name}" />
        </mx:FormItem>
        <mx:FormItem label="Description">
            <mx:TextInput id="itemDescription"
                text="{_selectedItem.description}" />
        </mx:FormItem>
        <mx:FormItem label="Priority">
            <mx:ComboBox id="itemPriority"
                selectedItem="{_selectedItem.priority}">
                <mx:dataProvider>
                    <mx:ArrayCollection>
                        <mx:Number>1</mx:Number>
                        <mx:Number>2</mx:Number>
                        <mx:Number>3</mx:Number>
                        <mx:Number>4</mx:Number>
                        </mx:ArrayCollection>
                    </mx:dataProvider>
                </mx:ComboBox>
            </mx:FormItem>
            <mx:FormItem label="Must Be Done By">
                <mx:DateField id="itemMustBeDoneBy"
                ➥selectedDate="{_selectedItem.mustBeDoneBy}" />
            </mx:FormItem>
            <mx:FormItem>
                <mx:Button id="formButton" label="Add"
                    click="addItem(event);" />
            </mx:FormItem>
        </mx:Form>
    </mx:VBox>
    <mx:states>
        <mx:State name="Edit">
            <mx:SetProperty target="{formHeading}" name="label"
                value="Edit Item" />
            <mx:SetProperty target="{formButton}" name="label"
                value="Update" />
            <mx:SetEventHandler target="{formButton}" name="click"
                handlerFunction="updateItem" />
        </mx:State>
    </mx:states>
</mx:WindowedApplication>
```

В этом коде набор данных получается с помощью метода `getResult()` ❶. Затем просматриваются все записи ❷, создаются новые экземпляры

компоненты ToDoListRenderer, и каждая запись присваивается свойству data компоненты ❸. При щелчке пользователя по кнопкам Edit или Delete на компоненте обрабатываются соответствующие события. Код этих методов пока не определен. Он будет вставлен ниже. Листинг 5.7 показывает, как выглядят эти методы.

Листинг 5.7. Удаление и редактирование пунктов

```
private function deleteItem(event:Event):void {
   var item:Object = event.currentTarget.data;
   var sql:SQLStatement = new SQLStatement();
   sql.sqlConnection = _connection;
   sql.text = "DELETE FROM todo WHERE id = @id";
   sql.parameters["@id"] = item.id;
   sql.addEventListener(SQLEvent.RESULT, selectToDoItems);
   sql.execute();
}

private function editItem(event:Event):void {
   var item:ToDoItem = event.currentTarget.data;
   _selectedItem = item;    currentState = "Edit";
}
```

Как можно видеть, метод deleteItem() просто создает параметризованную команду для удаления выбранного пункта. Метод editItem() всего лишь устанавливает значение свойства _selectedItem и изменяет текущее состояние.

На этом разработка приложения ToDo завершается. Теперь вы можете добавлять новые пункты и видеть их в списке. Затем можно редактировать их и удалять.

5.6. Работа с несколькими базами данных

В наши дни большинство операторов телефонных услуг предлагает клиентам схемы и варианты, допускающие трехсторонние разговоры или телеконференции, в которых в одном телефонном разговоре может быть много участников. Не для каждого разговора уместна такая функция. Иногда вам нужно поговорить с человеком один на один. Но иногда удобно и полезно подключить к разговору несколько лиц. Допустим, например, что есть некий проект с рядом участников. Хотя с каждым из них можно поговорить отдельно, иногда более продуктивно провести переговоры всем вместе одновременно. В этом случае обмен информацией и принятие решений могут происходить быстрее. То же самое справедливо в отношении баз данных. Иногда достаточно подключения только к одной базе данных. Но в других случаях приложению может потребоваться подключиться к нескольким базам данных. В этом разделе мы рассмотрим стратегии подключения более чем к одной базе данных.

Ваши нынешние знания позволяют открыть соединение со второй базой данных с помощью второго объекта SQLConnection. Однако этот способ имеет по крайней мере два существенных недостатка:

- Если объектов SQLConnection два, вы не сможете использовать две базы данных в одной команде SQL. (Например, нельзя одновременно выбрать данные, находящиеся в таблицах, расположенных в разных базах данных.)
- Соединения SQL являются дорогостоящими с точки зрения обработки. Поэтому обычно гораздо лучше открыть несколько баз данных с помощью единственного объекта SQLConnection. Первое соединение открывается так, как вы уже делали это: с помощью метода open() или openAsync(). Для новых соединений можно воспользоваться методом attach().

При подключении новой базы данных всегда нужно указывать ее имя. Это имя не совпадает с именем файла базы данных. Это то имя, которое вы будете указывать как базу данных в командах SQL. Мы еще не говорили об этом, но у главной базы данных тоже есть имя: main. В командах SQL можно ссылаться на главную базу данных по ее имени. Например, следующие команды SQL эквивалентны, когда таблица musicTracks находится в главной базе данных:

```
SELECT album FROM main.musicTracks
SELECT album FROM musicTracks
```

Примечание

В действительности всегда более эффективно указывать в командах SQL как имя таблицы, так и имя базы данных. Если вы хотите выжать из своего приложения AIR максимальную скорость, указывайте имя базы данных, даже если оно подразумевается по умолчанию.

Прикрепляя базу данных, вы ссылаетесь впоследствии на ее таблицы по псевдониму, который передается в качестве первого параметра метода attach().

Чаще всего методом attach() открывают соединение с новой, а не главной базой данных, и тогда нужно задать в методе attach() второй параметр: объект File, указывающий на файл базы данных. Задать режим, как в open() или openAsync(), нельзя, поскольку подключаемые с помощью attach() базы данных автоматически открываются в том же режиме, что и главная база данных. Это означает, что если для главной базы данных выбран режим создания (create), то и все подключаемые базы данных будут находиться в режиме создания. В следующем примере подключается база данных с псевдонимом userCustomData:

```
connection.attach("userCustomData",
➥File.applicationStorageDirectory.resolvePath("userdata.db"));
```

Функция подключения баз данных используется не часто, но она весьма удобна.

5.7. Добавление в AirTube поддержки баз данных

Овладев основными навыками работы с базами данных и даже применив их в приложении ToDo, мы теперь можем ввести в наше приложение AirTube функции, связанные с базами данных. В главе 3 мы предоставили пользователю возможность локальной загрузки видеофайлов. Но тогда мы еще не умели работать с базами данных в AIR. Поэтому мы не снабдили приложение функциями, которые позволяют сохранять данные о фильмах и осуществлять поиск и воспроизведение фильмов, сохраненных на локальном диске. Этим мы займемся в последующих разделах. Для этого мы должны сделать следующее:

- Добавить в `ApplicationData` свойство `online`.
- Добавить кнопку для переключения режимов онлайн/офлайн.
- Добавить служебные методы для работы в автономном режиме.

Начнем с модификации `ApplicationData`.

5.7.1. Модификация ApplicationData для поддержки режимов онлайн, офлайн

До сих пор у приложения AirTube был только один режим: онлайн. Мы хотим, чтобы пользователь мог выбирать между режимами онлайн и офлайн. Для этого нужно добавить в класс `ApplicationData` новое свойство. Это свойство с именем `online` будет булевым значением, указывающим режим, в котором должно работать приложение: онлайн или офлайн. Листинг 5.8 показывает, как выглядит `ApplicationData` после добавления этого свойства.

Листинг 5.8. Добавление свойства online в класс ApplicationData

```
package com.manning.airtube.data {

   import flash.events.Event;
   import flash.events.EventDispatcher;

   public class ApplicationData extends EventDispatcher {

      static private var _instance:ApplicationData;

      private var _videos:Array;
      private var _currentVideo:AirTubeVideo;
      private var _downloadProgress:Number;
      private var _online:Boolean;

      [Bindable(event="videosChanged")]
      public function set videos(value:Array):void {
         _videos = value;
         dispatchEvent(new Event("videosChanged"));
      }

      public function get videos():Array {
         return _videos;
```

```
        }

        [Bindable(event="currentVideoChanged")]
        public function set currentVideo(value:AirTubeVideo):void {
           _currentVideo = value;
           dispatchEvent(new Event("currentVideoChanged"));
        }

        public function get currentVideo():AirTubeVideo {
           return _currentVideo;
        }

        [Bindable(event="downloadProgressChanged")]
        public function set downloadProgress(value:Number):void {
           _downloadProgress = value;
           dispatchEvent(new Event("downloadProgressChanged"));
        }

        public function get downloadProgress():Number {
           return _downloadProgress;
        }

        [Bindable(event="onlineChanged")]
        public function set online(value:Boolean):void {
           _online = value;
           dispatchEvent(new Event("onlineChanged"));
        }

        public function get online():Boolean {
           return _online;
        }

        public function ApplicationData() {

        }

        static public function getInstance():ApplicationData {
           if(_instance == null) {
              _instance = new ApplicationData();
           }
           return _instance;
        }

     }
  }
```

Свойство `online` простое. Мы создаем закрытое булево свойство и стандартные методы для его чтения и изменения, а также добавляем обычные метаданные Flex для привязки источников данных. После этого нам нужно создать средство, с помощью которого пользователь сможет переключаться между двумя режимами, чем мы займемся в следующем разделе.

5.7.2. Добавление кнопки для переключения режимов

Теперь можно отредактировать AirTube.mxml, добавив туда кнопку, которая даст возможность переключаться между сетевым и автономным режимами. Соответствующий код приведен в листинге 5.9.

Листинг 5.9. Добавление в AirTube.mxml кнопки переключения режимов

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" width="800" height="600"
   creationComplete="creationCompleteHandler();"
closing="closingHandler();">
   <mx:Script>
      <![CDATA[
         import com.manning.airtube.data.AirTubeVideo;
         import com.manning.airtube.windows.HTMLWindow;
         import com.manning.airtube.windows.VideoWindow;
         import com.manning.airtube.services.AirTubeService;
         import com.manning.airtube.data.ApplicationData;

         static private var _instance:AirTube;

         private var _service:AirTubeService;
         private var _videoWindow:VideoWindow;
         private var _htmlWindow:HTMLWindow;

         static public function getInstance():AirTube {
            return _instance;
         }

         private function creationCompleteHandler():void {
            _service = AirTubeService.getInstance();
            _service.key = "YourAPIKey";
            _instance = this;
            _videoWindow = new VideoWindow();
               _htmlWindow = new HTMLWindow();
         }

         private function getVideosByTags():void {
            _service.getVideosByTags(tags.text);
         }

         private function playVideo():void {
            var video:AirTubeVideo =
             ➥videoList.selectedItem as AirTubeVideo;
            _service.configureVideoForPlayback(video);
            if(_videoWindow.nativeWindow == null) {
               _videoWindow.open();
            }
            else {
               _videoWindow.activate();
            }
         }
```

```
            public function launchHTMLWindow(url:String):void {
              if(_htmlWindow.nativeWindow == null) {
                  _htmlWindow.open();
              }
              else {
                  _htmlWindow.activate();
              }
            }

            private function closingHandler():void {
               for(var i:Number = 0; i <
                ➥nativeApplication.openedWindows.length; i++) {
                 nativeApplication.openedWindows[i].close();
               }
            }

            private function changeOnlineStatus():void {
               ApplicationData.getInstance().online =
               ➥!ApplicationData.getInstance().online;
            }

         ]]>
      </mx:Script>
      <mx:VBox width="100%">
         <mx:Label text="AirTube: Adobe AIR and YouTube" />
         <mx:HBox>
            <mx:Label text="tags:" />
            <mx:TextInput id="tags" text="Adobe AIR" />
            <mx:Button label="Search For Videos"
               click="getVideosByTags();" />
            <mx:Button label="Online" toggle="true"
               selected="{ApplicationData.getInstance().online}"
               click="changeOnlineStatus();" />
         </mx:HBox>
         <mx:TileList id="videoList"
            dataProvider="{ApplicationData.getInstance().videos}"
            width="100%" height="400"
            columnCount="2" horizontalScrollPolicy="off" />
         <mx:Button label="Play Selected Video" click="playVideo();"
            enabled="{videoList.selectedItem != null}" />
      </mx:VBox>
   </mx:WindowedApplication>
```

В этом коде добавлены всего одна кнопка и один метод. Кнопка – это переключатель, в котором выбранное состояние привязано к свойству `online` объекта `ApplicationData`. При щелчке по кнопке метод – обработчик события – просто переключает значение свойства `online` экземпляра `ApplicationData` на противоположное.

Это все, что нужно изменить в интерфейсе пользователя. Теперь мы должны обновить служебный код для поддержки как сетевого, так и автономного режимов.

5.7.3. Поддержка сохранения и поиска для режима офлайн

Большая часть кода, который нужно написать для поддержки сетевого и автономного режимов, располагается в классе `AirTubeService`. Этот код приведен в листинге 5.10. Нового кода довольно много, но это не должно вызывать у вас беспокойства. Объяснения последуют незамедлительно. Все, что мы добавляем, – это простой код для работы с базой данных, который открывает соединение, создает таблицу, добавляет и извлекает данные.

Листинг 5.10. Добавление в AirTubeService поддержки автономного режима

```
package com.manning.airtube.services {

   import  com.adobe.webapis.youtube.YouTubeService;
   import  com.adobe.webapis.youtube.events.YouTubeServiceEvent;
   import  com.manning.airtube.data.AirTubeVideo;
   import  com.manning.airtube.data.ApplicationData;
   import  com.manning.airtube.utilities.YouTubeFlvUrlRetriever;
   import  com.adobe.webapis.youtube.Video;

   import  flash.events.Event;
   import  flash.events.ProgressEvent;
   import  flash.filesystem.File;
   import  flash.filesystem.FileMode;
   import  flash.filesystem.FileStream;
   import  flash.net.URLRequest;
   import  flash.net.URLStream;
   import  flash.utils.ByteArray;
   import  flash.events.SQLEvent;
   import  flash.data.SQLConnection;
   import  flash.data.SQLMode;
   import  flash.data.SQLResult;
   import  flash.data.SQLStatement;

   public class AirTubeService {

      static private var _instance:AirTubeService;

      private  var _proxied:YouTubeService;
      private  var _flvFile:File;
      private  var _imageFile:File;
      private  var _downloadingVideo:AirTubeVideo;
      private  var _connection:SQLConnection;

      public function set key(value:String):void {
         _proxied.apiKey = value;
      }

      public function AirTubeService() {
         _proxied = new YouTubeService();
         _proxied.addEventListener(
➥YouTubeServiceEvent.VIDEOS_LIST_BY_TAG, getVideosByTagsResultHandler);
```

```
    var databaseFile:File =
    ➥File.applicationStorageDirectory.resolvePath("AirTube.db");
    _connection = new SQLConnection();
    _connection.addEventListener(SQLEvent.OPEN,
                            databaseOpenHandler);
    _connection.openAsync(databaseFile, SQLMode.CREATE);
}

static public function getInstance():AirTubeService {
    if(_instance == null) {
        _instance = new AirTubeService();
    }
    return _instance;
}

private function databaseOpenHandler(event:Event):void {
    var sql:SQLStatement = new SQLStatement();
    sql.sqlConnection = _connection;
    sql.text = "CREATE TABLE IF NOT EXISTS videos(" +
        "id TEXT PRIMARY KEY, title TEXT, " +
        "url TEXT, tags TEXT)";
    sql.execute();
}

public function getVideosByTags(tags:String):void {
    if(_proxied.apiKey.length == 0) {
        throw Error("YouTube API key not set");
    }
    if(ApplicationData.getInstance().online) {
        _proxied.videos.listByTag(tags);
    }
    else {
        var sql:SQLStatement = new SQLStatement();
        sql.addEventListener(SQLEvent.RESULT,
                             getOfflineVideosResultHandler);
        sql.sqlConnection = _connection;
        var text:String = "SELECT * FROM videos WHERE 1 = 0";
        var tagsItems:Array = tags.split(" ");
        for(var i:Number = 0; i < tagsItems.length; i++) {
            text += " OR tags LIKE ?";
            sql.parameters[i] = "%" + tagsItems[i] + "%";
        }
        sql.text = text;
        sql.itemClass = Video;
        sql.execute();
    }
}

private function getVideosByTagsResultHandler(
➥event:YouTubeServiceEvent):void     {
    var videos:Array = event.data.videoList as Array;
    for(var i:Number = 0; i < videos.length; i++) {
        videos[i] = new AirTubeVideo(videos[i]);
```

❶ Создать соединение с базой данных

❷ Создать таблицу

❸ Проверить режим

❹ Составить SELECT для автономного режима

```
        }
        ApplicationData.getInstance().videos = videos;
    }

    private function getOfflineVideosResultHandler(event:SQLEvent):
    ➥void   {
        var statement:SQLStatement = event.target as SQLStatement;
        var result:SQLResult = statement.getResult();
        var videos:Array = new Array();
        var video:AirTubeVideo;
        if(result != null && result.data != null) {
            for(var i:Number = 0; i < result.data.length; i++) {
                video = new AirTubeVideo(result.data[i]);
                video.offline = true;
                video.flvUrl =
                ➥File.applicationStorageDirectory.resolvePath("videos/" +
                ➥video.video.id + ".flv").nativePath;
                video.video.thumbnailUrl =
            ➥File.applicationStorageDirectory.resolvePath("thumbnails/" +
                ➥video.video.id + ".jpg").nativePath;
                videos.push(video);
            }
        }
        ApplicationData.getInstance().videos = videos;
    }

    public function configureVideoForPlayback(video:AirTubeVideo):
    ➥void   {
        ApplicationData.getInstance().currentVideo = video;
        if(video.flvUrl == null) {
            new YouTubeFlvUrlRetriever().getUrl(video);
        }
    }

    public function saveToOffline(video:AirTubeVideo):void {
        _downloadingVideo = video;

        _flvFile =
        ➥File.applicationStorageDirectory.resolvePath("videos/" +
        ➥video.video.id + ".flv");
        var videoLoader:URLStream = new URLStream();
        videoLoader.load(new URLRequest(video.flvUrl));
        videoLoader.addEventListener(Event.COMPLETE,
                              videoDownloadCompleteHandler);
        videoLoader.addEventListener(ProgressEvent.PROGRESS,
        ➥videoDownloadProgressHandler);

        _imageFile =
        ➥File.applicationStorageDirectory.resolvePath("thumbnails/" +
        ➥video.video.id + ".jpg");
        var imageLoader:URLStream = new URLStream();
        imageLoader.load(new URLRequest(video.video.thumbnailUrl));
        imageLoader.addEventListener(ProgressEvent.PROGRESS,
```

❺ Обернуть результат в AirTubeVideo

Получить ссылки на файлы

```
        imageDownloadProgressHandler);
            }

            private function videoDownloadProgressHandler(event:ProgressEvent):
            ➥void{
                var loader:URLStream = event.target as URLStream;
                var bytes:ByteArray = new ByteArray();
                loader.readBytes(bytes);
                var writer:FileStream = new FileStream();
                writer.open(_flvFile, FileMode.APPEND);
                writer.writeBytes(bytes);
                writer.close();
                var ratio:Number = event.bytesLoaded / event.bytesTotal;
                ApplicationData.getInstance().downloadProgress = ratio;
            }

            private function videoDownloadCompleteHandler(event:Event):void {
                _downloadingVideo.offline = true;
                ApplicationData.getInstance().downloadProgress = 0;
                var sql:SQLStatement = new SQLStatement();
                sql.sqlConnection = _connection;
                 sql.text = "INSERT INTO videos(" +
                    "title, id, url, tags) VALUES(" +
                    "@title, @id, @url, @tags)";
                 sql.parameters["@title"] = _downloadingVideo.video.title;
                 sql.parameters["@id"] = _downloadingVideo.video.id;
                 sql.parameters["@url"] = _downloadingVideo.video.url;
                 sql.parameters["@tags"] = _downloadingVideo.video.tags;
                 sql.execute();      }

            private function imageDownloadProgressHandler(event:ProgressEvent):
            ➥void  {
                var loader:URLStream = event.target as URLStream;
                var bytes:ByteArray = new ByteArray();
                loader.readBytes(bytes);
                var writer:FileStream = new FileStream();
                writer.open(_imageFile, FileMode.APPEND);
                writer.writeBytes(bytes);
                writer.close();
            }
        }
    }
```

Создать ❻ параметрическую команду

Несмотря на объем добавленного кода, он не должен представлять трудностей для понимания при том навыке работы с базами данных в AIR, который вы получили в этой главе. Сначала нужно создать соединение с базой данных ❶. Подключившись к базе данных, нужно создать таблицу, если ее еще не существует ❷. В данном случае создается одна таблица с колонками id, title, url и tags. Затем в методе поиска фильмов проверяется текущий режим ❸. Если режим сетевой, можно искать фильмы в сети, как обычно. В противном случае нужно выполнять поиск среди автономно хранящихся фильмов. Составляем

команду SELECT ❹, учитывая заданные пользователем ключевые слова. Получив результаты, просматриваем все записи (имеющие вид объектов com.adobe.webapis.youtube.Video) и заключаем их в объекты AirTubeVideo ❺. С другой стороны, когда пользователь сохраняет фильм для автономного режима, теперь недостаточно записать файл с фильмом. Мы еще должны сохранить в базе данных сведения о фильме ❻.

Пока это все на данном этапе разработки приложения AirTube. Тестируя это приложение, вы должны будете суметь сохранить фильмы локально, а потом переключиться в режим офлайн и найти эти фильмы (и воспроизвести их).

5.8. Резюме

В этой главе вы научились пользоваться в AIR базами данных. С помощью локальных баз данных можно организовать постоянное хранение данных и осуществлять их чтение и запись, применяя SQL – стандартный язык для работы с базами данных. Вы узнали, что в AIR используется движок баз данных под названием SQLite, и научились применять его для создания баз данных и таблиц, записи данных в базу, чтения, редактирования и удаления данных. Вы не только познакомились с базами данных в теории, но и применили знания на практике, создав приложение ToDo и модифицировав приложение AirTube, в котором теперь есть функция, использующая локальные базы данных.

Как говорилось в начале главы, одно из применений локальных баз данных – хранение данных приложений, не имеющих постоянного подключения к сети, для обеспечения их работы в автономном режиме. Теперь вам известно о локальных базах данных все, что необходимо для записи и извлечения из них данных в такой ситуации. Но вы еще пока не умеете определять состояние подключения к сети. Следующая глава 6 рассматривает эту тему.

6

В этой главе:
- Контроль соединений по HTTP
- Контроль соединений через сокеты

Сетевое взаимодействие

Представьте себе, что вы создаете приложение, которое должно дать пользователям возможность редактировать текстовые документы, находящиеся на сервере, подключенном к Интернету. Когда пользователь запускает приложение, оно соединяется с сервером и получает список имеющихся там документов. Пользователь выбирает файл, после чего текст загружается с сервера и становится доступен для редактирования. Когда нужно сохранить файл, приложение посылает новый текст на сервер, где он и сохраняется. Такое приложение может прекрасно работать, пока пользователь подключен к Интернету. Но если соединения нет, возникает проблема: нельзя ни редактировать, ни сохранять файлы, потому что для этого требуется соединение с Интернетом. С помощью AIR можно построить более удачное приложение, которое будет знать, когда пользователь подключен к Интернету. Определив, подключен ли пользователь к Интернету или доступен ли некий ресурс, приложение AIR может вести себя соответствующим образом. Оно может действовать по-разному в зависимости от доступности сети. Например, если пользователь приложения, редактирующего текст, запускает его, будучи подключенным к сети, он может начать редактировать файл. Но если до того, как файл был сохранен, произошло отключение (например, пользователь сел в самолет), текст будет сохранен в локальной базе данных до того момента, как сеть не станет снова доступна.

В этой главе вы сможете научиться применению различных сетевых функций AIR. В их число входят мониторы доступности сети.

6.1. Контроль подключения к сети

Улучшенная версия приложения, которое мы описали во введении в эту главу, служит примером приложения с непостоянным подключением

к сети. Это означает, что приложение предназначено для работы как в сетевом, так и в автономном режиме, при этом пользователям обычно не приходится существенно менять характер своей работы при переходе из одного режима в другой. В некоторых случаях пользователи могут оставаться в полном неведении относительно того, в каком режиме они работают, потому что приложение совершенно незаметно настраивается на работу в текущем состоянии. Приложения с непостоянным подключением к сети могут сильно различаться в отношении сложности, контента и функциональности, но у них есть общее качество: они должны контролировать состояние подключения к сети. Этот вопрос мы и изучим в данном разделе.

AIR позволяет создавать приложения, которые контролируют подключение к сети с помощью мониторов (service monitors). В AIR есть два типа таких мониторов: `URLMonitor` и `SocketMonitor`. Оба эти класса делают примерно одно и то же: следят за сетевым соединением и проверяют его доступность. Когда состояние доступности контролируемого соединения меняется, объект монитора рассылает событие. Различие между `URLMonitor` и `SocketMonitor` состоит в типе соединения, которое контролирует каждый из них. `URLMonitor` может следить за соединениями HTTP, а `SocketMonitor` может следить за соединениями более низкого уровня.

Классы `URLMonitor` и `SocketMonitor` не входят в стандартные библиотеки AIR. Это означает, что если вы захотите скомпилировать приложение AIR, в котором используются эти классы, вы должны подключить соответствующую библиотеку. Эти классы находятся в файле .swc под названием servicemonitor.swc:

- Если вы работаете с Flex Builder, подключать вручную библиотеку не требуется, поскольку Flex Builder сделает это автоматически.
- Если вы работаете с Flex SDK, то обнаружите файл .swc в каталоге frameworks/libs/air.
- Если вы работаете с Flash CS3, то обнаружите файл .swc в каталоге AIK/frameworks/libs/air установочного каталога Flash CS3.

Далее мы рассмотрим, как пользоваться URLMonitor.

6.1.1. Контроль соединения HTTP

Для контроля соединения по HTTP можно воспользоваться классом `air.net.URLMonitor`. Сначала нужно создать экземпляр класса. Конструктору нужно передать объект `flash.net.URLRequest`, который определяет адрес HTTP, подлежащий мониторингу. Следующий код создает объект `URLMonitor` для слежения за сайтом Manning:

```
var monitor:URLMonitor =
➥new URLMonitor(new URLRequest("http://www.manning.com"));
```

Как уже говорилось, все объекты-мониторы посылают события, когда меняется состояние соединения. Это событие имеет тип `status`, и при

регистрации обработчика в качестве имени можно использовать константу `StatusEvent.STATUS`. В следующем примере регистрируется обработчик события `status`:

```
monitor.addEventListener(StatusEvent.STATUS, statusHandler);
```

После того как вы создадите объект URLMonitor и зарегистрируете обработчик события `status`, нужно запустить монитор, иначе ничего не произойдет. Запуск осуществляется методом `start()`:

```
monitor.start();
```

Если монитор запущен, можно обратиться к нему, чтобы узнать, доступно ли в данный момент контролируемое им соединение. Эта информация находится в булевом свойстве `available`, которое имеет значение `true`, если соединение доступно, и `false` – в противном случае. Проверить значение свойства `available` можно в любое время, хотя обычно разумно это сделать, когда возникает событие `status`.

Код листинга 6.1 представляет собой простую программу контроля доступности сети. Контролируется доступность *www.manning.com*. Если состояние доступности меняется, пользователь оповещается текстовым сообщением.

Листинг 6.1. Применение объекта URLMonitor для контроля доступности ресурса HTTP

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import air.net.URLMonitor;

         private var _monitor:URLMonitor;

         private function creationCompleteHandler():void {
            _monitor =
            ➥new URLMonitor(new URLRequest("http://www.manning.com"));
            _monitor.addEventListener(StatusEvent.STATUS,
                                      statusHandler);
            _monitor.start();
         }

         private function statusHandler(event:StatusEvent):void {
            textArea.text += "www.manning.com available? " +
            ➥_monitor.available + "\n";
         }

      ]]>
   </mx:Script>
   <mx:TextArea id="textArea" width="100%" height="100%" />
</mx:WindowedApplication>
```

Можете сами протестировать этот пример. Запустите приложение, а потом измените доступность сети, подключив или отсоединив сетевой кабель или отключив и заново включив сетевое устройство.

Обычно объекты-мониторы не занимаются периодическим опросом сетевых ресурсов. Они опрашивают ресурс только при запуске монитора и при каждом изменении состояния сети. Это означает, что по умолчанию монитор в действительности чаще проверяет, есть ли у компьютера пользователя доступ к сети, чем доступность самого сетевого ресурса. Возьмем приложение из листинга 6.1. Когда состояние подключения компьютера пользователя к сети изменяется, пользователь извещается об этом. Но пока компьютер подключен к Интернету, он не получит сообщения о том, что сервер *www.manning.com* стал недоступен, если такое произойдет. Если помимо состояния соединения с сетью вы еще хотите проверять доступность некоторого ресурса, нужно сообщить монитору, чтобы он чаще осуществлял опрос. Для этого нужно изменить значение свойства `pollInterval`. По умолчанию оно равно 0, что означает отсутствие периодического опроса ресурса. Любое положительное целое число в качестве значения этого свойства определяет частоту опроса в миллисекундах. Например, следующий код требует, чтобы объект опрашивал ресурс каждые 10 секунд:

```
monitor.pollInterval = 10000;
```

По умолчанию объект `URLMonitor` интерпретирует следующие коды состояния как положительные ответы, свидетельствующие о доступности ресурса: 200, 202, 204, 205 и 206. Изменить список приемлемых кодов состояния можно двумя способами: передать массив кодов в качестве второго параметра конструктору `URLMonitor` или присвоить этот массив кодов свойству `acceptableStatusCodes` объекта монитора.

После запуска монитора его свойство `running` принимает значение `true`. Остановить монитор можно, вызвав метод `stop()`.

6.1.2. Контроль за доступностью сокетов

С помощью класса `air.net.SocketMonitor` можно контролировать соединение с другими сокетами – не только теми, которые поддерживают соединения HTTP. Например, если ваше приложение подключается к серверу, распространяющему потоковое видео, можете контролировать доступность сервера с помощью объекта `SocketMonitor`.

Для работы с объектом `SocketMonitor` нужно прежде всего вызвать его конструктор. Конструктор `SocketMonitor` принимает два параметра: имя сервера и номер порта. Следующий код создает объект `SocketMonitor` для контроля доступности сервера FTP:

```
var monitor:SocketMonitor = new SocketMonitor("ftp.exampleserver.com", 21);
```

`SocketMonitor` и `URLMonitor` расширяют класс `air.net.ServiceMonitor` и потому их поведение схоже. Объект `SocketMonitor` работает аналогично

объекту `URLMonitor`. Вы можете перехватывать события `status`, запускать и останавливать мониторинг с помощью методов `start()` и `stop()`, проверять текущее состояние с помощью свойства `available` и задавать интервал опроса с помощью свойства `pollInterval`.

Листинг 6.2 демонстрирует пример использования объекта `SocketMonitor` для определения того, что показывать пользователю в приложении, которое получает текущее время с сервера National Institute of Standards and Times (NIST) на порту 13.

Листинг 6.2. Получение времени с помощью объекта Socket

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import air.net.SocketMonitor;

         private var _monitor:SocketMonitor;
         private var _socket:Socket;
         private var _timer:Timer;

         private const SERVER:String = "time-A.timefreq.bldrdoc.gov";
         private const PORT:int = 13;

         private function creationCompleteHandler():void {
            _monitor = new SocketMonitor(SERVER, PORT);
            _monitor.addEventListener(StatusEvent.STATUS,
                                      statusHandler);
            _monitor.start();

            _socket = new Socket();
            _socket.addEventListener(ProgressEvent.SOCKET_DATA,
                                     socketDataHandler);

            _timer = new Timer(10000);
            _timer.addEventListener(TimerEvent.TIMER, timerHandler);
         }

         private function statusHandler(event:StatusEvent):void {
            if(!_monitor.available) {
               timeText.text = "Server unavailable.
               ➥Will reconnect when the server is next available.";
               _timer.stop();
            }
            else {
               timerHandler();
               _timer.start();
            }
         }

         private function socketDataHandler(event:ProgressEvent):void {
            var fullTime:String =
            ➥_socket.readUTFBytes(_socket.bytesAvailable);
            var time:String = fullTime.split(" ")[2];
```

❶ Создать монитор

Запустить монитор

❷ Создать таймер

Проверить статус сети

❸ Остановить таймер

❹ Перезапустить таймер

Прочесть время

```
            timeText.text = "The current time (UTC) is: " + time;
            _socket.close();
        }

        private function timerHandler(event:TimerEvent = null):void {
            if(_monitor.available && !_socket.connected) {
                _socket.connect(SERVER, PORT);
            }
        }

    ]]>
    </mx:Script>
    <mx:Text id="timeText" width="100%" height="100%" />
</mx:WindowedApplication>
```

Закрыть сокет

Проверить сетевое соединение

Подключиться к сокету

В этом примере мы используем монитор ❶, чтобы контролировать соединение с сервером NIST на порту 13. Мы также пользуемся таймером ❷, чтобы запрашивать на сервере текущее время каждые 10 секунд. Если состояние сети изменится, мы хотим выполнить правильные действия. Если сетевое соединение недоступно, мы показываем пользователю сообщение и останавливаем таймер ❸. Если соединение восстановлено, мы снова запускаем таймер ❹. Если сервер отвечает на запросы, то формат его данных похож на 54540 08-03-15 18:06:11 50 0 0 348.3 UTC(NIST) *. Можно видеть, что время суток – это третья из групп символов, разделенных пробелами. Поэтому мы читаем данные из сокета с помощью `readUTFBytes()` и выбираем третью группу для показа пользователю.

6.2. Добавление контроля сети в AirTube

В этом разделе мы добавим контроль сетевого соединения в приложение AirTube. С помощью объекта `URLMonitor` мы сможем следить за тем, подключена ли система в данный момент к сети, и автоматически настраивать приложение для работы в сетевом или автономном режиме. Сначала мы добавим в класс `ApplicationData` новое свойство. Свойство `networkAvailable` является булевым, и с его помощью мы будем определять, есть ли у приложения доступ к сети. Листинг 6.3 демонстрирует класс `ApplicationData` после добавления этого свойства.

Листинг 6.3. Добавление свойства networkAvailable в класс ApplicationData

```
package com.manning.airtube.data {

    import flash.events.Event;
    import flash.events.EventDispatcher;

    public class ApplicationData extends EventDispatcher {

        static private var _instance:ApplicationData;

        private var _videos:Array;
        private var _currentVideo:AirTubeVideo;
        private var _downloadProgress:Number;
```

```
private var _online:Boolean;
private var _networkAvailable:Boolean;

[Bindable(event="videosChanged")]
public function set videos(value:Array):void {
   _videos = value;
   dispatchEvent(new Event("videosChanged"));
}

public function get videos():Array {
   return _videos;
}

[Bindable(event="currentVideoChanged")]
public function set currentVideo(value:AirTubeVideo):void {
   _currentVideo = value;
   dispatchEvent(new Event("currentVideoChanged"));
}

public function get currentVideo():AirTubeVideo {
   return _currentVideo;
}

[Bindable(event="downloadProgressChanged")]
public function set downloadProgress(value:Number):void {
   _downloadProgress = value;
   dispatchEvent(new Event("downloadProgressChanged"));
}

public function get downloadProgress():Number {
   return _downloadProgress;
}

[Bindable(event="onlineChanged")]
public function set online(value:Boolean):void {
   _online = value;
   dispatchEvent(new Event("onlineChanged"));
}

public function get online():Boolean {
   return _online;
}

[Bindable(event="networkAvailableChanged")]
public function set networkAvailable(value:Boolean):void {
   if(value != _networkAvailable) {
      _networkAvailable = value;
      if(!_networkAvailable) {
         online = false;
         videos = null;
      }
      dispatchEvent(new Event("networkAvailableChanged"));
   }
}

public function get networkAvailable():Boolean {
```

```
            return _networkAvailable;
        }

        public function ApplicationData() {
        }

        static public function getInstance():ApplicationData {
            if(_instance == null) {
                _instance = new ApplicationData();
            }
            return _instance;
        }

    }
}
```

Как видите, сеттер[1] осуществляет еще некоторые функции помимо установки нового значения для закрытого свойства. В данном случае мы хотим установить для приложения автономный режим работы, если сеть недоступна. Кроме того, мы также обнуляем массив `videos`, чтобы пользователь не пытался смотреть фильмы, которые доступны только через сеть, пока она не подключена.

После этого нужно добавить объект `URLMonitor` в класс сервиса. Можете отредактировать `AirTubeService.as`. Сначала добавьте новое закрытое свойство `_monitor`, как в следующей строке кода:

```
private var _monitor:URLMonitor;
```

Затем измените конструктор:

```
public function AirTubeService() {
    _proxied = new YouTubeService();
    _proxied.addEventListener(YouTubeServiceEvent.VIDEOS_LIST_BY_TAG,
                      getVideosByTagsResultHandler);
    var databaseFile:File =
    ➥File.applicationStorageDirectory.resolvePath("AirTube.db");
    _connection = new SQLConnection();
    _connection.addEventListener(SQLEvent.OPEN, databaseOpenHandler);
    _connection.openAsync(databaseFile, SQLMode.CREATE);
    _monitor = new URLMonitor(new URLRequest("http://www.youtube.com"));
    _monitor.addEventListener(StatusEvent.STATUS, networkStatusHandler);
    _monitor.start();
}
```

Как видно, мы контролируем с помощью объекта `URLMonitor` соединение с *www.youtube.com*. Теперь добавьте такой метод `networkStatusHandler()`:

```
private function networkStatusHandler(event:StatusEvent):void {
    ApplicationData.getInstance().networkAvailable = _monitor.available;
}
```

---

[1] Геттеры и сеттеры описывались в главе 3 в подразделе «Чтение объектов» (стр. 157). – *Прим. науч. ред.*

Этот метод просто устанавливает свойство networkAvailable объекта ApplicationData, когда изменяется статус подключения к сети. При этом, как мы видели в листинге 6.3, меняются также значения online и videos.

Осталось лишь модифицировать AirTube.mxml, чтобы пользователь мог видеть состояние подключения к сети. Мы добавим метку, как показано в следующем коде. Новый код выделен жирным шрифтом, а остальной приведен, чтобы вы могли увидеть контекст:

```
mx:VBox width="100%">
  <mx:Label text="AirTube: Adobe AIR and YouTube" />
  <mx:HBox>
     <mx:Label text="tags:" />
      <mx:TextInput id="tags" text="Adobe AIR" />
      <mx:Button label="Search For Videos" click="getVideosByTags();" />
      <mx:Button label="Online" toggle="true"
         selected="{ApplicationData.getInstance().online}"
         click="changeOnlineStatus();" />
      <mx:Label text="{ApplicationData.getInstance().networkAvailable ?
    ➥'Network Available' : 'Network Unavailable'}" />
   </mx:HBox>
   <mx:TileList id="videoList"
      dataProvider="{ApplicationData.getInstance().videos}"
      width="100%" height="400" columnCount="2"
      horizontalScrollPolicy="off"
      itemRenderer="com.manning.airtube.ui.VideoTileRenderer" />
   <mx:Button label="Play Selected Video" click="playVideo();"
      enabled="{videoList.selectedItem != null}" />
</mx:VBox>
```

Эта новая метка использует связанный источник данных для показа пользователю состояния сети.

И это все, что нужно изменить в приложении AirTube для поддержки изменения состояния сети.

6.3. Резюме

В этой главе мы узнали, как определять изменение состояния сети с помощью классов URLMonitor и SocketMonitor. Мы научились определять, когда у компьютера есть доступ к конкретным сетевым ресурсам, таким как веб-страница или некоторый порт сервера. С помощью этих классов мониторов можно создавать приложения, имеющие непостоянное подключение к сети.

В главе 7 мы займемся новой темой: работой в AIR с HTML. Вы узнаете, как показывать HTML и как интегрировать функциональность JavaScript на HTML-страницах, загружаемых в приложения AIR.

В этой главе:

- Загрузка и показ HTML
- Управление загрузкой HTML
- Обращение к объектам JavaScript из ActionScript
- Обращение к ActionScript из JavaScript

7

HTML в AIR

Обычно мы пытаемся умерять свой энтузиазм в отношении отдельных функций AIR, стараясь вести себя с холодной рассудительностью. Но работа с HTML в AIR организована настолько замечательно, что сдерживать свое восхищение становится просто невозможным. В этой главе мы поделимся с вами тем, что нам известно о работе с HTML в AIR, и когда факты предстанут перед вашими глазами, мы надеемся, что вы разделите наш энтузиазм.

Будучи разработчиком Flash или Flex, вы, конечно, часто помещали файлы .swf на страницах HTML. А ведь неплохо было бы иметь возможность делать обратное: выводить HTML внутри приложений Flash или Flex! Такую возможность как раз и предоставляет AIR, поскольку включает в себя движок веб-броузера WebKit (*www.webkit.org*) – тот самый, на котором основан популярный кросс-платформенный броузер Safari. Движок WebKit позволяет приложениям AIR отображать HTML и выполнять JavaScript – почти как в стандартном броузере. Но еще более примечательно то, что когда HTML отображается в приложении AIR, он не только интерактивен, каким и должен быть HTML в броузере, но и выступает в качестве отображаемого объекта в приложении AIR. Это означает, что с ним можно проводить такие операции, как масштабирование, вращение, размывание, маскирование и т. д.

На протяжении этой главы мы рассмотрим важные темы, касающиеся работы с HTML в приложениях AIR. Сначала рассмотрим базовые вопросы, такие как отображение контента HTML в приложении AIR. Мы также обсудим, как проводить с контентом HTML такие операции, как прокрутка, навигация среди посещенных страниц и взаимодействие с JavaScript.

7.1. Показ HTML в AIR

При работе с HTML прежде всего нужно научиться загружать его, отображать и показывать в приложениях AIR. В среде AIR это осуществляется удивительно легко. Нужно лишь создать экземпляр встроенного класса `flash.html.HTMLLoader` и сообщить ему, где взять HTML, который нужно отобразить.

Если вы разрабатываете приложение AIR с помощью Flash CS3, то всегда будете работать только с экземплярами `HTMLLoader` и вам требуется прочесть следующий подраздел (7.1.1), но можно пропустить раздел 7.1.3. Если вы разрабатываете приложение AIR с помощью Flex, то вам будет полезно прочесть оба эти подраздела. Хотя Flex предоставляет разработчику компоненту для работы с HTML, эта компонента все равно внутренне пользуется `HTMLLoader`, поэтому чем глубже вы разберетесь в низкоуровневом классе `HTMLLoader`, тем лучше.

7.1.1. Применение встроенных объектов Flash, отображающих HTML

В классе `flash.html.HTMLLoader` есть практически все необходимое для работы с HTML в приложении AIR. Этот класс имеет тип отображаемого объекта, благодаря чему его экземпляры можно добавлять в список отображения. В классе `HTMLLoader` есть также функции, позволяющие сообщить ему, откуда загружать контент HTML, после чего он знает, как выводить контент.

Есть два основных способа сообщить объекту `HTMLLoader`, какой HTML нужно выводить:

- С помощью метода `load()` сообщить объекту `HTMLLoader`, из какого ресурса, например страницы на сервере, загружать HTML.
- С помощью метода `loadString()` передать объекту `HTMLLoader` строку HTML, которую нужно отобразить.

Как можно предположить, гораздо чаще приходится загружать HTML из ресурса, поэтому сначала рассмотрим этот вариант. Метод `load()` требует лишь одного параметра: объекта `flash.net.URLRequest`, указывающего место, где объект `HTMLLoader` может найти HTML, который нужно загрузить. Следующий пример показывает, как легко можно загрузить и отобразить контент HTML. Код листинга 7.1 загружает страницу HTML с *www.manning.com* и выводит ее в объекте `HTMLLoader`.

Листинг 7.1. Загрузка и отображение страниц HTML с помощью HTMLLoader

```
package com.manning.airinaction.html {

   import flash.display.MovieClip;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
```

```
    public class Main extends MovieClip {
        private var _htmlLoader:HTMLLoader;
        public function Main() {
            _htmlLoader = new HTMLLoader();
            _htmlLoader.width = stage.stageWidth;
            _htmlLoader.height = stage.stageHeight;
            addChild(_htmlLoader);
            _htmlLoader.load(new URLRequest(
            ➥"http://www.manning.com/lott"));
        }
    }
}
```

Результат выполнения приведенного кода показан на рис. 7.1.

***Рис. 7.1.** Загрузка страницы HTML в объект HTMLLoader с помощью метода load( )*

Как показывает этот пример, ввиду того, что класс `HTMLLoader` принадлежит к отображаемому типу, у него есть обычные свойства отображаемого объекта, такие как ширина и высота, и с ними можно обращаться так же, как с этими свойствами любого отображаемого объекта. В данном примере мы задаем ширину и высоту объекта `HTMLLoader`, но с таким же успехом мы могли бы задать угол поворота, значение прозрачности и фильтры (размывание, тени и т. д.).

Как уже отмечалось, можно «загружать» HTML программным образом, присваивая строки объекту `HTMLLoader`. Для этого применяется метод `loadString()`, как демонстрирует листинг 7.2.

Листинг 7.2. Отображение строк HTML

```
package com.manning.airinaction.html {
    import flash.display.MovieClip;
```

```
import flash.html.HTMLLoader;
import flash.net.URLRequest;

public class Main extends MovieClip {

   private var _htmlLoader:HTMLLoader;
   public function Main() {
    _htmlLoader = new HTMLLoader();
    _htmlLoader.width = stage.stageWidth;
    _htmlLoader.height = stage.stageHeight;
    addChild(_htmlLoader);
      htmlLoader.loadString("<html><body><h1>HTML in AIR</h1>
    ➥</body></html>");
  }
 }

}
```

Результат работы этого кода показан на рис. 7.2.

***Рис. 7.2.** Показ строк HTML с помощью метода loadString( )*

Метод `loadString()` удобен, когда вы программным образом хотите указать приложению AIR, какой HTML нужно вывести. Например, можно создать шаблоны HTML и сохранить их в файлах в каталоге приложения AIR, после чего программно загрузить шаблоны HTML, провести их анализ, присвоить значения переменным шаблонов, а затем присвоить полученный HTML объекту `HTMLLoader` с помощью `loadString()`.

7.1.2. Загрузка контента PDF

С помощью подключаемого модуля Adobe Acrobat 8.1 или выше AIR может показывать контент PDF. Если на машине установлен Acrobat Reader 8.1 или выше, то загрузка и показ PDF происходят так же, как для HTML. Нужно только указать PDF URI при запросе загрузки контента в экземпляр HTMLLoader. Однако если в системе нет Acrobat Reader 8.1 или выше, такая операция невозможна. По этой причине в AIR включе-

на проверка способности системы показывать контент PDF, осуществляемая с помощью статического свойства HTMLLoader.pdfCapability.

Свойство HTMLLoader.pdfCapability может иметь четыре значения. Эти значения соответствуют четырем константам класса flash.html.HTMLPDFCapability: STATUS_OK, ERROR_INSTALLED_READER_NOT_FOUND, ERROR_INSTALLED_READER_TOO_OLD, ERROR_PREFERRED_READER_TOO_OLD. Если значение равно STATUS_OK, то AIR сможет показать PDF. В противном случае AIR не сможет показать PDF по одной из следующих причин: не найден Acrobat Reader, найденная версия слишком стара или пользователь настроил систему на использование старой версии, несмотря на наличие 8.1 или выше.

7.1.3. Использование компоненты Flex

При создании приложений AIR на базе Flex обычно следует использовать вместо стандартного отображаемого объекта компоненту Flex. Например, в приложениях Flex для показа текста используется компонента Text вместо прямого обращения к объекту более низкого уровня TextField. То же справедливо при работе в приложении Flex с HTML: вместо прямого обращения к объекту HTMLLoader используется компонента Flex. В данном случае, эта компонента называется HTML.

Подобно тому, как компонента Text служит оболочкой для объекта низшего уровня TextField, компонента HTML служит оболочкой для объекта HTMLLoader. По этой причине компонента HTML допускает те же виды поведения, что и объект HTMLLoader. Имеются следующие отличия:

- Компоненты HTML являются компонентами Flex и могут добавляться в другие компоненты Flex (вкладываться в контейнеры)
- API компонент HTML и HTMLLoader несколько различны

Если вам нужно загрузить контент в компоненту HTML, вы должны воспользоваться свойством location. Присваивание значения свойству location приводит к тому, что компонента автоматически запрашивает, загружает и отображает контент. Например, код листинга 7.3 загружает и выводит ту же веб-страницу, что и код листинга 7.1.

Листинг 7.3. Свойство location указывает на HTML, который нужно показать

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" width="350">
   <mx:HTML width="100%" height="100%"
      location="http://www.manning.com/lott" />
</mx:WindowedApplication>
```

Результат выполнения кода представлен на рис. 7.3.

Если вы были внимательны, то заметили различие между результатами листингов 7.1 и 7.3. Оно заключается в том, что когда мы пользуемся компонентой HTML, Flex автоматически добавляет полосы прокрутки.

***Рис. 7.3.** С помощью компоненты HTML можно выводить контент HTML в приложении Flex*

Напротив, при использовании низкоуровневого класса `HTMLLoader` полосы прокрутки сами не появятся. Но не тревожьтесь: в следующем разделе мы обсудим, как прокручивать контент HTML. Но сначала мы более подробно изучим применение компоненты `HTML`.

Вас могло заинтересовать, позволяет ли компонента `HTML` выводить строки HTML таким же образом, как это делает с помощью метода `loadString()` объект `HTMLLoader`. Оказывается, компонента HTML не предоставляет такой функции непосредственно. Однако компонента HTML обеспечивает доступ к экземпляру `HTMLLoader`, для которого служит оболочкой, через свойство `htmlLoader`. Это означает, что можно получить через это свойство ссылку на объект `HTMLLoader`, а затем вызвать метод `loadString()` этого объекта, как показано в листинге 7.4.

Листинг 7.4. Доступ к объекту HTMLLoader через свойство htmlLoader

```
<?xml version="1.0" encoding="utf-8"?>
  <mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
     layout="absolute" width="350"
     creationComplete="creationCompleteHandler();">
     <mx:Script>
        <![CDATA[

           private function creationCompleteHandler():void {
              html.htmlLoader.loadString("<html><body>
              ➥<h1>HTML in AIR</h1></body></html>");
           }

        ]]>
     </mx:Script>
     <mx:HTML id="html" width="100%" height="100%" />
  </mx:WindowedApplication>
```

С помощью объекта HTMLLoader, вложенного в компоненту HTML, можно делать все то же, что с любым другим объектом HTMLLoader. Это означает, что чуть ли не все, что вы узнали в этой главе об объекте HTMLLoader, также применимо к компоненте HTML, хотя и косвенным образом – через ее базовый объект HTMLLoader.

Итак, вы теперь знаете, как загружать и показывать HTML в приложении AIR. У вас могло возникнуть подозрение, что удивительная легкость загрузки и показа HTML таит в себе какую-то уловку. Например, не ограничивает ли AIR ваши возможности управлять разными аспектами загрузки HTML? Нет, ваши подозрения напрасны. Как будет показано в следующем разделе, AIR позволяет управлять различными аспектами загрузки HTML приложениями AIR.

7.2. Управление загрузкой HTML

Как правило, вам не приходится думать о деталях, связанных с загрузкой контента HTML приложением AIR. Обычно действующих по умолчанию параметров оказывается достаточно практически для любого загружаемого контента HTML. Но иногда бывает желательно несколько изменить действия приложения при запросе и обработке контента HTML.

Например, вам может потребоваться, чтобы приложение всегда получало контент с сервера, а не брало его из локального кэша, либо вы захотите автоматизировать аутентификацию пользователей. В последующих разделах мы покажем, как можно осуществить эти и некоторые другие вещи. Все описываемые ниже свойства относятся к объектам HTMLLoader, и применять их к компонентам HTML можно через их внутренний объект HTMLLoader.

7.2.1. Управление кэшированием контента

По умолчанию AIR кэширует контент, запрашиваемый приложением через объект HTMLLoader (или компоненту HTML). Это означает, что когда пользователь просматривает контент HTML в приложении AIR, копии HTML, графики и пр. сохраняются локально; при повторном запросе приложением AIR каких-либо из этих ресурсов используются их локально сохраненные версии, если только не истек срок их годности. Кэширование контента – стандартный режим работы веб-броузеров, позволяющий повысить скорость загрузки контента при повторном посещении страниц, поскольку необходимые ресурсы размещаются локально, а не в сети.

Несмотря на пользу кэширования контента, такое поведение иногда оказывается нежелательным. В некоторых случаях требуется обеспечить пользователю загрузку контента с сервера, чтобы гарантировать просмотр самой свежей версии. Управлять кэшированием контента можно двумя способами:

- Сообщить AIR, нужно ли кэшировать контент
- Сообщить AIR, нужно ли использовать ранее кэшированный контент

Если вы хотите потребовать от объекта `HTMLLoader`, чтобы он не кэшировал контент, нужно присвоить свойству `cacheResponse` значение `false`, прежде чем вызывать метод `load()`. По умолчанию это свойство имеет значение `true`, что влечет выполнение кэширования контента для объекта.

Если вы хотите потребовать от объекта `HTMLLoader`, чтобы он не использовал ранее кэшированный контент, нужно присвоить свойству `useCache` значение `false`, прежде чем вызывать метод `load()`. По умолчанию это свойство имеет значение `true`, что влечет чтение объектом кэшированного контента, если таковой существует.

7.2.2. Управление аутентификацией

Иногда сервер требует аутентификации для доступа к определенному контенту. Например, на многих серверах используются файлы .htaccess для ограничения прав доступа пользователей к определенным каталогам. В таких ситуациях приложения AIR выводят диалоговое окно, в котором пользователь может ввести идентифицирующие его имя и пароль. Если вы хотите отключить такое поведение AIR и не дать пользователю возможности ввести свои учетные данные, можно воспользоваться свойством `authenticate` объекта `HTMLLoader`. Просто задайте для свойства `authenticate` объекта `HTMLLoader` значение `false`, прежде чем вызывать `load()`, и тогда, если сервер потребует ввода данных аутентификации, приложение AIR не покажет пользователю это диалоговое окно. В этом случае сервер возвратит ошибку.

7.2.3. Задание агента пользователя

Приложение, которое запрашивает у сервера веб-контент, называется агентом пользователя. Когда агент пользователя запрашивает у сервера веб-контент, он посылает свои идентификационные данные. Агент пользователя называет себя с помощью информации, обозначаемой как `userAgent`. Некоторые скрипты в зависимости от значения `userAgent` определяют, какой контент предоставить или как его показать (и даже показывать ли его вообще). По этой причине не столь редко прибегают к подделке данных агента (user agent-spoofing). Смысл ее в том, что агент пользователя предоставляет фальшивые данные, идентифицирующие его, чтобы выдать себя за агента другого типа. Например, один веб-броузер выдает себя за другой, просто указывая иное значение `userAgent`.

При запросе контента HTML с помощью `HTMLLoader` вы также можете подделать данные агента пользователя, задав значение свойству `userAgent` объекта HTMLLoader. Компонента HTML позволяет непосредственно присваивать значение свойству `userAgent`.

7.2.4. Управление постоянными данными

Некоторые веб-страницы хранят постоянные данные в локальных файлах, контролируемых веб-броузером. Эти файлы называются *cookies*. Есть страницы, которые не могут функционировать без таких cookies. По умолчанию AIR сохраняет cookies для страниц, которые вы просматриваете в приложениях AIR. Но есть ряд причин, по которым может понадобиться отключить cookies для контента HTML в AIR. Одна из главных среди них заключается в том, что пользователю вашего приложения нужно предоставить возможность отключить cookies, поскольку он может возражать против того, чтобы веб-страницы хранили данные локально на его компьютере. Каковы бы ни были причины, вы можете явно управлять записью cookies с помощью свойства `manageCookies` объекта `HTMLLoader`. Его значение по умолчанию `true`, а значение `false` отключает cookies.

7.2.5. Задание значений по умолчанию

До сих пор мы рассматривали способы управления отдельными объектами `HTMLLoader`. Как правило, мы указывали, какими значениями по умолчанию пользуется приложение AIR, если вы не задали значение свойства явно. Например, мы отмечали, что значением по умолчанию свойства `useCache` является `true`. Обычно так оно и есть, но не всегда. Дело в том, что можно задать значения по умолчанию, которые будет использовать приложение AIR для свойств `authenticate`, `useCache`, `cacheResponse`, `userAgent` и `manageCookies`. Чтобы установить значения по умолчанию, присвойте нужные значения свойствам с такими же именами класса `flash.net.URLRequestDefaults`. (Все свойства этого класса статические.) Например, можно по умолчанию отключить кэширование:

```
URLRequestDefaults.useCache = false;
```

Все объекты `HTMLLoader` получают значения по умолчанию из `URLRequestDefaults`. Это означает, что если у объекта `HTMLLoader` свойство `useCache` имеет значение `null`, будет использовано его значение из `URLRequestDefaults`. Однако значения по умолчанию всегда можно перекрыть, задав свойства объекта `HTMLLoader` явным образом.

Разобравшись с управлением загрузкой HTML в AIR, посмотрим, что можно делать с контентом после того, как он уже загружен. Конкретно, займемся прокруткой контента.

7.3. Прокрутка контента HTML

Иногда контент HTML больше, чем область, в которой вы хотите его показать. Например, вы установили высоту объекта `HTMLLoader` равной 500, а контент оказался высотой в 1000 пикселов. В таких случаях обычно нужно дать пользователю возможность прокручивать контент.

В этом разделе мы рассмотрим различные проблемы, возникающие при прокрутке контента HTML в приложениях AIR.

7.3.1. Прокрутка HTML во Flex

Как уже отмечалось в этой главе, компонента Flex `HTML` при необходимости автоматически добавляет полосы прокрутки к контенту HTML. Поэтому если вы хотите, чтобы появлялись полосы прокрутки, когда контент больше, чем компонента `HTML`, ничего делать не нужно. Если вы хотите управлять полосами прокрутки более явным образом, воспользуйтесь свойствами `horizontalScrollPolicy` и `verticalScrollPolicy`. Это стандартные свойства многих компонент Flex со встроенными полосами прокрутки, включая `TextArea` и `List`. Значениями этих свойств могут быть `auto` (по умолчанию), `on` и `off`. При значении `on` полосы прокрутки видны всегда, даже если прокрутка невозможна. При значении `off` полосы прокрутки не показываются никогда.

Примечание

`HTMLLoader` (лежащий в основе компоненты `HTML`) автоматически разрешает вертикальную прокрутку с помощью колесика мыши. Даже если отключить полосы прокрутки в компоненте `HTML`, пользователь сохранит возможность прокрутки с помощью колесика.

Поскольку прокрутка является встроенной возможностью компоненты `HTML`, при работе во Flex вам можно больше особенно не беспокоиться. Но если вам требуется разобраться в низкоуровневом механизме прокрутки или вы создаете приложения AIR с помощью Flash, тогда прочтите следующие разделы.

7.3.2. Прокрутка контента HTML с помощью ActionScript

Работая с объектом `HTMLLoader`, вы не получаете встроенных полос прокрутки. Но, добавив немного кода ActionScript, вы сможете выполнять прокрутку в объекте `HTMLLoader`, как будет показано в этом разделе.

Свойства `width` и `height` объекта `HTMLLoader` определяют размер контейнера, но ничего не говорят о содержимом этого контейнера. Хотя размеры контейнера важны для прокрутки, они составляют лишь часть того, что вам нужно знать. Вам нужно еще знать размеры контента. Как определить ширину и высоту контента объекта `HTMLLoader`? Сделать это в AIR почти так же просто, как определить размеры контейнера. Достаточно прочесть значения свойств `contentWidth` и `contentHeight`. Но здесь есть одна хитрость: чтобы получить точные значения `contentWidth` и `contentHeight`, нужно дождаться конца загрузки контента. Отсюда вопрос: как узнать об окончании загрузки контента? Для этого нужно дождаться, когда объект `HTMLLoader` отправит сообщение `complete`. После получения события `complete` можно прочесть значения `contentWidth` и `contentHeight` и определить размеры контента, загружен-

ного в объект HTMLLoader. Если contentWidth больше значения свойства width, то вы должны организовать горизонтальную прокрутку. Если contentHeight больше значения свойства height, то вы должны организовать вертикальную прокрутку.

Определив, что прокрутка нужна, вы должны реализовать ее. Можно программным образом прокручивать контент объекта HTMLLoader с помощью свойств scrollH и scrollV. Свойство scrollH управляет горизонтальной прокруткой, а свойство scrollV – вертикальной. Значение 0 в том и другом случае означает, что контент выровнен с контейнером. Положительные значения вызывают прокрутку вниз или вправо на соответствующее число пикселов. Например, если задать для scrollH значение 50, контент будет прокручен на 50 пикселов вправо. Диапазон значений прокрутки можно определить, вычтя из размеров контента размеры контейнера. Например, разность между contentWidth и width укажет вам максимальное значение scrollH.

Рассмотрим пример. Листинг 7.5 содержит код, который добавляет полосы прокрутки, с помощью которых пользователь может прокручивать содержимое объекта HTMLLoader.

Листинг 7.5. Прокрутка контента с помощью scrollH и scrollV

```
package com.manning.airinaction.html {

   import flash.display.MovieClip;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
   import fl.controls.ScrollBar;
   import flash.events.Event;

   public class Main extends MovieClip {

      private var _htmlLoader:HTMLLoader;
      private var _scrollBarH:ScrollBar;
      private var _scrollBarV:ScrollBar;

      public function Main() {
         _scrollBarV = new ScrollBar();
         _scrollBarV.height = stage.stageHeight - 16;
         _scrollBarV.x = stage.stageWidth - 16;
         _scrollBarV.addEventListener(Event.SCROLL,
                              scrollVerticalHandler);
         addChild(_scrollBarV);

         _scrollBarH = new ScrollBar();
         _scrollBarH.direction = "horizontal";
         _scrollBarH.width = stage.stageWidth - 16;
         _scrollBarH.x = 0;
         _scrollBarH.y = stage.stageHeight - 16;
         _scrollBarH.addEventListener(Event.SCROLL,
                              scrollHorizontalHandler);
         addChild(_scrollBarH);
```

❶ Создать вертикальную полосу прокрутки

❷ Создать горизонтальную полосу прокрутки

```
            _htmlLoader = new HTMLLoader();
            _htmlLoader.width = stage.stageWidth - 16;
            _htmlLoader.height = stage.stageHeight - 16;
            _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
            _htmlLoader.addEventListener(Event.SCROLL, scrollHandler);
            _htmlLoader.load(new URLRequest(
            ➥"http://www.manning.com/lott"));
            addChild(_htmlLoader);
        }

        private function completeHandler(event:Event):void {
            _htmlLoader.scrollH = 0;
            _htmlLoader.scrollV = 0;
            _scrollBarV.setScrollProperties(_htmlLoader.height, 0,
            ➥_htmlLoader.contentHeight - _htmlLoader.height);
            _scrollBarH.setScrollProperties(_htmlLoader.width, 0,
            ➥_htmlLoader.contentWidth - _htmlLoader.width);
        }

        private function scrollVerticalHandler(event:Event):void {
            _htmlLoader.scrollV = _scrollBarV.scrollPosition;
        }

        private function scrollHorizontalHandler(event:Event):void {
            _htmlLoader.scrollH = _scrollBarH.scrollPosition;
        }

        private function scrollHandler(event:Event):void {
            _scrollBarV.scrollPosition = _htmlLoader.scrollV;
        }

    }

}
```

Перехватить событие complete ❸

Перехватить событие scroll ❹

❺ Задать свойства полос прокрутки

Вертикальная прокрутка ❻

Горизонтальная прокрутка ❼

❽ Обновить полосы прокрутки

В этом примере используются две полосы прокрутки ❶ ❷, предоставляющие пользователю возможности горизонтальной и вертикальной прокрутки. Как мы уже знаем, нужно дождаться события `complete` ❸ и только потом делать вычисления для прокрутки. После получения события `complete` можно настроить свойства полос прокрутки ❺, исходя из размеров контента. Когда пользователь двигает полосы прокрутки, обновляем свойства `scrollV` и `scrollH` ❻ ❼. В этом примере мы знакомимся с событием `scroll`. Мы перехватываем событие `scroll` ❹, посылаемое объектом `HTMLLoader`. Это нужно нам для того, чтобы пользователь мог прокручивать контент в вертикальном направлении с помощью колесика мыши. Мы хотим, чтобы в этом случае значение вертикальной полосы прокрутки следовало за реальным положением контента ❽. Результат выполнения кода показан на рис. 7.4.

Мы рассмотрели программный способ управления прокруткой. Теперь рассмотрим особый случай, когда можно попросить AIR автоматически добавлять полосы прокрутки.

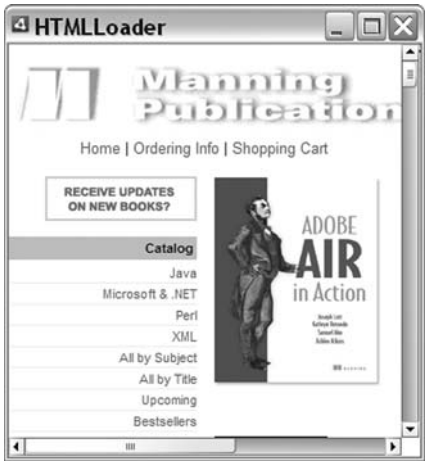


Рис. 7.4. *Добавление полос прокрутки контента HTML*

7.3.3. Создание окон с автопрокруткой

Иногда требуется показать контент HTML в новом окне AIR. В таких случаях можно заставить AIR открыть новое окно и автоматически поместить в него полосы прокрутки – все в одном методе. В классе `HTMLLoader` есть статический метод `createRootWindow()`, который именно это и делает.

Метод `createRootWindow()` создает новый объект `HTMLLoader` и возвращает его. Он также создает новое окно AIR и помещает в него этот объект `HTMLLoader`. Вам остается только вызвать метод `load()` объекта `HTMLLoader` и загрузить контент. Вот пример вызова данного метода:

```
var htmlLoader:HTMLLoader = HTMLLoader.createRootWindow();
htmlLoader.load(new URLRequest("http://www.manning.com/lott"));
```

Этот код открывает *www.manning.com/lott* в новом окне AIR и автоматически добавляет полосы прокрутки, если они нужны.

Метод `createRootWindow()` также принимает несколько необязательных параметров, а именно:

- `visible` – булево значение, указывающее, видимо ли окно изначально.
- `WindowInitOptions` – объект `NativeWindowInitOptions`. Подробнее об объектах `NativeWindowInitOptions` см. главу 2.
- `scrollBarsVisible` – булево значение, указывающее, должны ли показываться полосы прокрутки.
- `bounds` – объект `flash.geom.Rectangle`, с помощью которого можно задать координаты `x` и `y`, а также ширину и высоту нового окна.

Следующий код открывает окно размером 250 на 250 пикселов в левом верхнем углу экрана:

```
var htmlLoader:HTMLLoader = HTMLLoader.createRootWindow(true, null,
➥true, new Rectangle(0, 0, 250, 250));
htmlLoader.load(new URLRequest("http://www.manning.com/lott"));
```

На этом мы заканчиваем обсуждение прокрутки контента HTML и переходим к навигации по журналу посещений HTML (history).

7.4. Навигация по журналу посещений

Работая в обычном веб-броузере, вы, вероятно, привыкли пользоваться кнопками Назад и Вперед для перемещения по уже посещавшимся страницам. Например, если, открыв щелчком страницу, вы обнаружите, что попали не туда, куда вам нужно, вы, вероятно, щелкнете по кнопке броузера Назад и вернетесь на ту страницу, которую просматривали последней. Мы пока не встречались с такой функциональностью в AIR, но не потому, что она недоступна. Добавив небольшой объем кода, вы можете предоставить своим пользователям возможность навигации по журналу посещений. Все описываемые в этом разделе свойства и методы применимы к объектам `HTMLLoader` и компонентам HTML в равной степени.

`HTMLLoader` и HTML ведут учет посещенных страниц с помощью объектов `flash.html.HTMLHistoryItem`. Объекты `HTMLHistoryItem` располагают следующими свойствами:

- `url` – URL страницы.
- `originalUrl` – иногда страницы осуществляют переадресацию, и значение свойства `url` может отличаться от URL, к которому первоначально обращалась среда AIR. Свойство `originalUrl` содержит значение URL, по которому AIR обращалась до переадресации. Если переадресации не было, значения `originalUrl` и `url` совпадают.
- `title` – заголовок страницы.
- `isPost` – булево значение, указывающее, передавались ли на страницу данные POST.

У объектов `HTMLLoader` и `HTML` есть свойство `historyLength`, которое сообщает длину журнала. Оно говорит о том, сколько объектов `HTMLHistoryItem` находится в журнале. С помощью метода `getHistoryAt()` можно запросить `HTMLHistoryItem` с заданным индексом. Если в журнале есть хотя бы один объект, вызов `getHistoryAt()` с индексом 0 вернет самую старую из всех записей о посещениях, тогда как вызов `getHistoryAt()` с индексом, на единицу меньшим `historyLength`, возвратит самую свежую запись в журнале.

Можно также осуществлять относительное перемещение по журналу посещений с помощью методов `historyBack()`, `historyForward()` и `historyGo()`. Метод `historyBack()` перемещает на предыдущую страницу в жур-

***Рис. 7.5.** Навигация в журнале посещений*

нале, а метод `historyForward()` – на следующую. (Это возможно, только если уже происходило перемещение назад.) Метод `historyGo()` позволяет перемещаться шагами, большими единицы. Например, если нужно вернуться на две страницы назад, вызовите `historyGo()` со значением -2. Вызов `historyGo()` с отрицательными значениями вызывает перемещение назад, а с положительными значениями – вперед.

Теперь мы напишем пример приложения, предоставляющего возможность навигации по журналу посещений. На рис. 7.5 показано, как выглядит это приложение. Обратите внимание, что мы добавили не только кнопки Previous и Next, но и панель адреса.

В листинге 7.6 приведен код этого приложения. По большей части он совпадает с листингом 7.5, а добавленный код выделен жирным шрифтом. Следует учитывать, что в этом примере предполагается, что `_next`, `_previous` и `_go` – экземпляры кнопок, а `_htmlUrl` – экземпляр компоненты текстового ввода, и что все эти компоненты помещены на сцену во Flash.

Листинг 7.6. Навигация в журнале с помощью historyBack( ) и historyForward( )

```
package com.manning.airinaction.html {

   import flash.display.MovieClip;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
   import fl.controls.ScrollBar;
   import flash.events.Event;
   import flash.events.MouseEvent;

   public class Main extends MovieClip {

      private var _htmlLoader:HTMLLoader;
      private var _scrollBarH:ScrollBar;
```

```
private var _scrollBarV:ScrollBar;

public function Main() {
   _scrollBarV = new ScrollBar();
   _scrollBarV.height = stage.stageHeight - 16 - _previous.height;
   _scrollBarV.y = _previous.height;
   _scrollBarV.x = stage.stageWidth - 16;
   _scrollBarV.addEventListener(Event.SCROLL,
                        scrollVerticalHandler);
   addChild(_scrollBarV);

   _scrollBarH = new ScrollBar();
   _scrollBarH.direction = "horizontal";
   _scrollBarH.width = stage.stageWidth - 16;
   _scrollBarH.x = 0;
   _scrollBarH.y = stage.stageHeight - 16;
   _scrollBarH.addEventListener(Event.SCROLL,
                        scrollHorizontalHandler);
   addChild(_scrollBarH);

   _htmlLoader = new HTMLLoader();
   _htmlLoader.width = stage.stageWidth - 16;
   _htmlLoader.height = stage.stageHeight - 16 - _previous.height;
   _htmlLoader.y = _previous.height;
   _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
   _htmlLoader.addEventListener(Event.SCROLL, scrollHandler);
   addChild(_htmlLoader);

   _previous.addEventListener(MouseEvent.CLICK, previousHandler);
   _next.addEventListener(MouseEvent.CLICK, nextHandler);
   _go.addEventListener(MouseEvent.CLICK, goHandler);
   _htmlUrl.text = "http://www.manning.com/lott";
   goHandler();
}

private function completeHandler(event:Event):void {
   _htmlUrl.text = _htmlLoader.location;
   _htmlLoader.scrollH = 0;
   _htmlLoader.scrollV = 0;
   _scrollBarV.enabled = _htmlLoader.contentHeight >
   ➥_htmlLoader.height;
   _scrollBarH.enabled = _htmlLoader.contentWidth >
   ➥_htmlLoader.width;
   _scrollBarV.setScrollProperties(_htmlLoader.height, 0,
   ➥_htmlLoader.contentHeight - _htmlLoader.height);
   _scrollBarH.setScrollProperties(_htmlLoader.width, 0,
   ➥_htmlLoader.contentWidth - _htmlLoader.width);
}

private function scrollVerticalHandler(event:Event):void {
   _htmlLoader.scrollV = _scrollBarV.scrollPosition;
}

private function scrollHorizontalHandler(event:Event):void {
   _htmlLoader.scrollH = _scrollBarH.scrollPosition;
```

```
        }

        private function scrollHandler(event:Event):void {
           _scrollBarV.scrollPosition = _htmlLoader.scrollV;
        }

        private function goHandler(event:MouseEvent = null):void {
           _htmlLoader.load(new URLRequest(_htmlUrl.text));
        }

        private function previousHandler(event:MouseEvent):void {
           _htmlLoader.historyBack();
        }

        private function nextHandler(event:MouseEvent):void {
           _htmlLoader.historyForward();
        }

    }

}
```

Этот код всего лишь вызывает методы `historyBack()` и `historyForward()` при щелчке по соответствующей кнопке. В результате объект `HTMLLoader` перемещается по журналу посещений.

Теперь, став экспертом по навигации в журнале посещений, вы, вероятно, гадаете, какие еще задачи нас ждут. Следующей нашей темой будет взаимодействие между ActionScript и JavaScript при загрузке контента HTML в приложение AIR.

7.5. Взаимодействие с JavaScript

AIR не ограничивается загрузкой HTML и его выполнением. После загрузки и показа контента можно с ним еще и взаимодействовать, и контент может взаимодействовать с приложением AIR. Это происходит благодаря тому, что объект `HTMLLoader`, который загружает HTML, полностью предоставляет объектную модель документа HTML (DOM). Ниже мы рассмотрим несколько способов, которыми можно организовать взаимодействие ActionScript и JavaScript.

7.5.1. Управление элементами HTML/JavaScript из ActionScript

При загрузке контента HTML в объект `HTMLLoader` вся модель DOM становится доступной через свойство `window`. Свойство `window` объекта HTMLLoader отображается прямо на свойство JavaScript `window` внутри страницы HTML. Это означает возможность обращаться к любым элементам страницы HTML через свойство `window` объекта `HTMLLoader` так же, как вы бы это делали со страницы HTML через свойство `window` JavaScript. Нужно только предупредить, что при обращении к HTML DOM необходимо дождаться загрузки всей страницы.

В ряде последующих примеров будет использован HTML, приведенный в листинге 7.7. Для использования в примерах его нужно сохранить в файле example.html.

Листинг 7.7. Код HTML, записанный в example.html

```
<html>
   <script>

      var pageTitle = "Example";
      var description = "This is an example HTML page";

      function showAlert() {
         alert(description);
      }

   </script>
   <body>
      <p id="p1">
         HTML in AIR
      </p>
      <button onclick="showAlert()">Click</button>
   </body>
</html>
```

Как можно заметить, этот код делает следующее:

- Определяет две переменные JavaScript с именами `pageTitle` и `description`.
- Определяет функцию `showAlert()`, которая показывает окно, отображающее значение переменной `description`.
- Создает тег `p` с `id`, равным `p1`.
- Создает кнопку, при щелчке по которой вызывается `showAlert()`.

Сначала посмотрим, как можно извлечь значение переменной JavaScript с помощью ActionScript. Поскольку переменные JavaScript создаются как свойства объекта `window`, достаточно дождаться события `complete`, после чего можно прочесть значение нужного свойства объекта `window`. Код листинга 7.8 читает значение свойства `pageTitle` и выводит его в заголовке окна.

Листинг 7.8. Чтение переменной JavaScript

```
package com.manning.airinaction.html {

   import flash.display.MovieClip;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
   import flash.events.Event;
   import flash.desktop.NativeApplication;

   public class Main extends MovieClip {

      private var _htmlLoader:HTMLLoader;

      public function Main() {
```

```
            _htmlLoader = new HTMLLoader();
            _htmlLoader.width = stage.stageWidth;
            _htmlLoader.height = stage.stageHeight;
            _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
            _htmlLoader.load(new URLRequest("example.html"));
            addChild(_htmlLoader);
        }

        private function completeHandler(event:Event):void {
            stage.nativeWindow.title = _htmlLoader.window.pageTitle;
        }
    }
}
```

Как можно видеть, чтобы получить значение переменной `pageTitle` со страницы HTML, достаточно при возникновении события `complete` прочесть значение `_htmlLoader.window.pageTitle`. Результат выполнения кода показан на рис. 7.6.

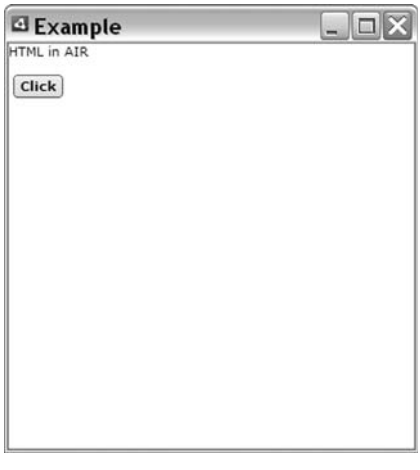


Рис. 7.6. *Показ значения в окне, организованный из ActionScript*

Переменные JavaScript доступны нам не только для чтения значений, но и для записи в них новых значений. Если вы запустите код листинга 7.8 и щелкнете по кнопке, то увидите окно, в котором показано значение переменной `description`, заданное на странице HTML. В листинге 7.9 мы записываем в переменную другое значение. Когда затем пользователь щелкает по кнопке, появляющееся окно показывает новое значение.

Листинг 7.9. Запись значения в переменную JavaScript из ActionScript

```
package com.manning.airinaction.html {

    import flash.display.MovieClip;
    import flash.html.HTMLLoader;
```

```
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.desktop.NativeApplication;

    public class Main extends MovieClip {

        private var _htmlLoader:HTMLLoader;

        public function Main() {
            _htmlLoader = new HTMLLoader();
            _htmlLoader.width = stage.stageWidth;
            _htmlLoader.height = stage.stageHeight;
            _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
            _htmlLoader.load(new URLRequest("example.html"));
            addChild(_htmlLoader);
        }

        private function completeHandler(event:Event):void {
            _htmlLoader.window.description = "This is a new description
            ➥from ActionScript";
        }

    }
}
```

Как видите, достаточно сослаться на переменную из объекта `window` после того, как произойдет событие `complete`, и присвоить ей новое значение. Рисунок 7.7. показывает отображение нового значения в окне.

Присваивать можно не только значения таких простых типов, как строки или булевы величины, но и ссылки на более сложные типы данных, включая функции. В листинге 7.10 функции `showAlert()` на странице HTML присваивается функция ActionScript. В результате, когда пользователь щелкает по кнопке на странице HTML, вызывается функция ActionScript, а не функция, определенная на странице HTML.

Рис. 7.7. *Показ значения переменной, заданного из ActionScript*

Листинг 7.10. Присваивание ссылочных типов переменным JavaScript

```
package com.manning.airinaction.html {

   import flash.display.MovieClip;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
   import flash.events.Event;

   public class Main extends MovieClip {

      private var _htmlLoader:HTMLLoader;

      public function Main() {
         _htmlLoader = new HTMLLoader();
         _htmlLoader.width = stage.stageWidth;
         _htmlLoader.height = stage.stageHeight;
         _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
         _htmlLoader.load(new URLRequest("example.html"));
         addChild(_htmlLoader);
      }

      private function completeHandler(event:Event):void {
         _htmlLoader.window.showAlert = function():void {
            trace("we've usurped control");
         };
      }

   }

}
```

Можно также получать ссылки на объекты, находящиеся внутри документа. Например, если вам нужно сослаться на тег `p` в HTML, ссылку на него можно получить через `_htmlLoader.window.document.getElementById("p1")`. Листинг 7.11 демонстрирует, как изменить текст в теге `p`.

Листинг 7.11. Получение ссылки на объект документа

```
package com.manning.airinaction.html {

   import flash.display.MovieClip;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
   import flash.events.Event;
   import flash.desktop.NativeApplication;

   public class Main extends MovieClip {

      private var _htmlLoader:HTMLLoader;
      public function Main() {
         _htmlLoader = new HTMLLoader();
         _htmlLoader.width = stage.stageWidth;
         _htmlLoader.height = stage.stageHeight;
         _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
         _htmlLoader.load(new URLRequest("example.html"));
         addChild(_htmlLoader);
      }
```

```
            private function completeHandler(event:Event):void {
                _htmlLoader.window.document.getElementById("p1").innerText =
                ➥"Hello from ActionScript";
            }
        }
    }
```

Если вы запустите код листинга 7.11, то увидите, что сначала появится первоначальный текст, но как только загрузится контент, этот текст будет заменен текстом, заданным в ActionScript, как показано на рис. 7.8.

Теперь, когда мы знаем, как ссылаться на элементы страницы HTML из ActionScript, можно сделать еще один шаг. В следующем разделе мы покажем, как обрабатывать в ActionScript события, генерируемые элементами HTML.

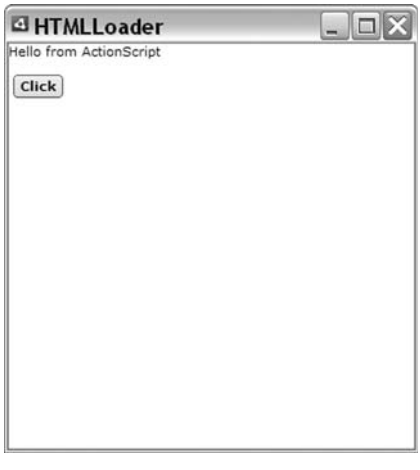


Рис. 7.8. *Замена текста на странице HTML на лету из ActionScript*

7.5.2. Обработка событий JavaScript из ActionScript

Мы видели в предыдущем разделе, что можно присвоить ссылку на функцию ActionScript переменной со страницы HTML, тем самым фактически захватив над ней контроль. Благодаря этому можно вызывать функции ActionScript из JavaScript. Но возможности работы с событиями далеко не ограничиваются заменой уже имеющихся обработчиков. Вы можете регистрировать новые обработчики событий.

Есть два способа регистрации обработчиков событий: присвоить ссылку на функцию (ActionScript или JavaScript) атрибуту обработчика события объекта `HTML` или воспользоваться методом `addEventListener()`.

Сначала рассмотрим присваивание ссылки на функцию атрибуту обработчика события. Если вы хотите, чтобы при щелчке по контенту тега

p вызывалась функция ActionScript, можно воспользоваться кодом вроде представленного в листинге 7.12.

Листинг 7.12. Присваивание ссылки на функцию атрибуту обработчика события

```
package com.manning.airinaction.html {

   import flash.display.MovieClip;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
   import flash.events.Event;
   public class Main extends MovieClip {

      private var _htmlLoader:HTMLLoader;

      public function Main() {
         _htmlLoader = new HTMLLoader();
         _htmlLoader.width = stage.stageWidth;
         _htmlLoader.height = stage.stageHeight;
         _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
         _htmlLoader.load(new URLRequest("example.html"));
         addChild(_htmlLoader);
      }

      private function completeHandler(event:Event):void {
         _htmlLoader.window.document.getElementById("p1").onclick =
         ➥clickHandler;
      }

      private function clickHandler(event:Object):void {
         trace("click");
      }

   }

}
```

Это совершенно корректный код, но вы имеете еще одну возможность: воспользоваться стандартной моделью рассылки событий ActionScript, зарегистрировав обработчик события с помощью addEventListener(), как показано в листинге 7.13.

Листинг 7.13. Регистрация обработчика событий с помощью addEventListener( )

```
package com.manning.airinaction.html {

   import flash.display.MovieClip;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
   import flash.events.Event;
   import flash.desktop.NativeApplication;

   public class Main extends MovieClip {
```

```
        private var _htmlLoader:HTMLLoader;

        public function Main() {
            _htmlLoader = new HTMLLoader();
            _htmlLoader.width = stage.stageWidth;
            _htmlLoader.height = stage.stageHeight;
            _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
            _htmlLoader.load(new URLRequest("example.html"));
            addChild(_htmlLoader);
        }

        private function completeHandler(event:Event):void {
            _htmlLoader.window.document.getElementById("p1").
            ➥addEventListener("click", clickHandler);
        }

        private function clickHandler(event:Object):void {
            trace("click");
        }

    }

}
```

Листинги 7.12 и 7.13 обращают на себя внимание тем, что метод, который обрабатывает событие, принимает параметр типа `Object`, а не типа `Event`. Этот параметр действует во многом так же, как объект `Event`, и у него есть свойства `target` и `currentTarget`, указывающие на элемент `HTML`, который отправил событие, но объект не относится к типу `Event`.

Теперь мы умеем обращаться к элементам JavaScript и HTML из ActionScript. Рассмотрим практический пример, демонстрирующий применение изученных нами методов.

7.5.3. Создание смешанного приложения

Может возникнуть вопрос о том, в каких ситуациях возникает необходимость интегрировать HTML и JavaScript в одном приложении AIR с помощью способов, описанных в предыдущих разделах. Рассмотрим сценарий: создание приложения AIR для заполнения опросного листа. После того как приложение создано, его владелец пожелал обновлять опросные листы с помощью приложения Ruby on Rails, выполняющегося на сервере и генерирующего новые опросники в виде файлов HTML. Приложение AIR должно загружать эти файлы HTML. В такой ситуации весьма разумно интегрировать ActionScript и HTML с помощью изученных нами способов. В этом разделе мы построим простой пример опросника, иллюстрирующий эти идеи.

Наше простое приложение будет загружать страницу HTML на этапе исполнения. Эта страница HTML содержит опросник, в котором может быть любое количество вопросов и ответов. Приложение AIR должно уметь получать ответы на вопросы, когда пользователь щелкает по кнопке HTML. Мы условимся, что в файле HTML опросника есть

кнопка Submit с идентификатором `submitButton`, и на странице присутствует функция `getSurveyResponses()`, которая возвращает массив ответов на вопросник.

В нашем примере файл HTML называется questionnaire.html, и его содержимое представлено в листинге 7.14.

Листинг 7.14. Файл HTML вопросника, предлагаемого пользователю

```
<html>
   <script>

      function getSurveyResponses() {
         var response;
         var options = document.questionnaire.skyColor;
         for(var i = 0; i < options.length; i++) {
            if(options[i].checked) {
               response = options[i].value;
            }
         }
         var response1 = {question: document.getElementById(
         ➥"question1").innerHTML, answer: response};
         options = document.questionnaire.skySize;
         for(i = 0; i < options.length; i++) {
            if(options[i].checked) {
               response = options[i].value;
            }
         }
         var response2 = {question: document.getElementById(
         ➥"question2").innerHTML, answer: response};

         return [response1, response2];
      }
   </script>
   <body>
      <h1>Questionnaire</h1>
      <form name="questionnaire">
         Please complete the following survey.
         <h2 id="question1">1. What is the color of the sky?</h2>
         <input type="radio" name="skyColor" value="grey">grey
         <input type="radio" name="skyColor" value="blue">blue
         <input type="radio" name="skyColor" value="no color">no color
         <input type="radio" name="skyColor" value="brown">brown

      <h2 id="question2">2. What is the size of the sky?</h2>
         <input type="radio" name="skySize" value="big">big
         <input type="radio" name="skySize" value="really big">
         ➥really big
         <input type="radio" name="skySize" value="not very big">
         ➥not very big
         <input type="radio" name="skySize" value="smaller than a cow">
         ➥smaller than a cow

         <h2>Click the button to finish
```

```
        <button id="submitButton">Submit Answers</button>
      </form>
    </body>
  </html>
```

Теперь создадим класс документа для ActionScript-проекта, в котором есть объект `HTMLLoader`, загружающий questionnaire.html. Мы перехватываем событие `click` от кнопки Submit, и когда пользователь щелкает по кнопке, выбираем все полученные ответы с помощью функции `getSurveyResponses()` и отображаем их средствами ActionScript. Код представлен листингом 7.15.

Листинг 7.15. Класс документа для приложения опросника

```
package com.manning.airinaction.questionnaire {

   import flash.display.MovieClip;
   import flash.events.MouseEvent;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
   import flash.events.Event;
   import fl.controls.TextArea;

   public class Main extends MovieClip {

      private var _htmlLoader:HTMLLoader;
      private var _textArea:TextArea;

      public function Main() {
         _htmlLoader = new HTMLLoader();
         _htmlLoader.width = stage.stageWidth;
         _htmlLoader.height = stage.stageHeight;
         _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
         _htmlLoader.load(new URLRequest("questionnaire.html"));
         addChild(_htmlLoader);

         _textArea = new TextArea();
         _textArea.width = stage.stageWidth;
         _textArea.height = stage.stageHeight;
      }

      private function completeHandler(event:Event):void {
         _htmlLoader.window.document.getElementById("submitButton").
        ➥addEventListener(MouseEvent.CLICK, clickHandler);
      }

      private function clickHandler(event:Object):void {
         removeChild(_htmlLoader);
         var responses:Array = _htmlLoader.window.getSurveyResponses();
         _textArea.text = "";
         var response:Object;
         for(var i:Number = 0; i < responses.length; i++) {
            response = responses[i] as Object;
            _textArea.text += response.question;
            _textArea.text += "\n\t" + response.answer + "\n\n";
         }
```

Загрузить questionnaire.html ❶

Зарегистрировать обработчик события click ❷

Получить ответы ❸

```
            addChild(_textArea);
        }
    }
}
```

Это очень простой код. Объект `HTMLLoader` загружает файл HTML ❶. По окончании загрузки код получает ссылку на кнопку в контенте HTML и регистрирует обработчик события `click` на этой кнопке ❷. Когда пользователь щелкает по кнопке, код вызывает функцию JavaScript ❸ из контента HTML, чтобы получить ответы на вопросы и показать их в компоненте `textArea`.

Выяснив, как управлять элементами HTML и JavaScript из ActionScript, посмотрим теперь, как получить доступ к объектам и классам ActionScript из JavaScript.

7.5.4. Обработка стандартных команд JavaScript

Многие стандартные команды JavaScript адресованы внешнему приложению, каковым обычно является веб-броузер. В нашем случае внешним является приложение AIR. Можно организовать обработку этих команд приложением AIR. Например, в веб-броузере при задании свойства `window.status` изменяется значение в строке состояния броузера. В приложении AIR нет действия, выполняемого по умолчанию в ответ на изменение свойства `window.status` из JavaScript.

Можно потребовать от AIR обрабатывать методы/изменения свойств, перечисленных в табл. 7.1, воспользовавшись объектом типа `flash.html.HTMLHost`. Объект `HTMLHost` обладает методами и свойствами, соответствующими методам/свойствам JavaScript, и это соответствие также приведено в табл. 7.1.

Табл. 7.1. Соответствие между JavaScript и HTMLHost

JavaScript	HTMLHost
`Window.status`	`updateStatus()`
`Window.location`	`updateLocation()`
`Window.document`	`title updateTitle()`
`Window.open()`	`createWindow()`
`Window.close()`	`windowClose()`
`Window.blur()`	`windowBlur()`
`Window.focus()`	`windowFocus()`
`Window.moveBy()`	`windowRect()`
`Window.moveTo()`	`windowRect()`
`Window.resizeBy()`	`windowRect()`
`Window.resizeTo()`	`windowRect()`

Вот как нужно действовать:

1. Создать свой класс, который расширяет HTMLHost.
2. Переопределить методы, где это требуется.
3. Присвоить экземпляр пользовательского класса свойству htmlHost объекта HTMLLoader, в который вы загружаете контент HTML.

Теперь посмотрим, как можно создать подкласс HTMLHost. Конструктор HTMLHost принимает необязательный булев параметр, который указывает, должно ли действовать поведение по умолчанию. Обычно конструктор подкласса реализуется точно таким же образом, например:

```
package com.manning.airinaction.html {
   import flash.html.HTMLHost;

   public class CustomHTMLHost extends HTMLHost {
      public function CustomHTMLHost(defaultBehavior:Boolean = true) {
         super(defaultBehavior);
      }

   }
}
```

Теперь нужно переопределить те методы, для которых мы хотим обеспечить особое поведение. Каждый из методов updateStatus(), updateLocation() и updateTitle() принимает один параметр: строку с новым значением. Мы напишем реализацию для updateTitle(), чтобы показать, как это делается. Существует много способов реализации методов подкласса HTMLHost. В данном случае мы просто поменяем заголовок главного окна:

```
package com.manning.airinaction.html {
   import flash.html.HTMLHost;
   import flash.desktop.NativeApplication;

   public class CustomHTMLHost extends HTMLHost {
      public function CustomHTMLHost(defaultBehavior:Boolean = true) {
         super(defaultBehavior);
      }

      override public function updateTitle(title:String):void {
         NativeApplication.nativeApplication.openedWindows[0].title =
         ➥title;
      }

   }
}
```

Затем напишем реализацию createWindow(). Методу createWindow() передается параметр типа flash.html.HTMLWindowCreateOptions. Этот объект соответствует опциям, передаваемым методу window.open() в качестве третьего параметра. В табл. 7.2 приведены атрибуты этого параметра метода window.open() и соответствующие свойства объекта HTMLWindowCreateOptions.

Табл. 7.2. Свойства HTMLWindowCreateOptions

Атрибут Window.open()	Свойство HTMLWindowCreateOptions
Width	Width
Height	Height
screenX, left	X
screenY, top	Y
Location	locationBarVisible
Menu	menuBarVisible
Scrollbars	scrollBarsVisible
Status	statusBarVisible
Toolbar	toolBarVisible
resizable	resizable
fullscreen	fullscreen

Метод createWindow() должен возвратить объект HTMLLoader. Метод createWindow() не знает, какие данные нужно загружать в объект HTMLLoader, да это ему и не нужно. Этот метод должен просто создать объект HTMLLoader и возвратить его. Загрузкой указанного контента в экземпляр HTMLLoader занимается AIR.

Теперь рассмотрим пример реализации createWindow(). В данном случае воспользуемся методом HTMLLoader.createRootWindow(), чтобы создать новое окно с размерами, указанными в параметре. Код показан в листинге 7.16.

Листинг 7.16. Создание нового окна с полосами прокрутки для контента HTML

```
package com.manning.airinaction.html {
   import flash.html.HTMLHost;
   import flash.desktop.NativeApplication;
   import flash.html.HTMLWindowCreateOptions;
   import flash.html.HTMLLoader;
   import flash.geom.Rectangle;

   public class CustomHTMLHost extends HTMLHost {
      public function CustomHTMLHost(defaultBehavior:Boolean = true) {
         super(defaultBehavior);
      }

      override public function updateTitle(title:String):void {
         NativeApplication.nativeApplication.openedWindows[0].title =
         ➥title;
      }

      override public function createWindow(
      ➥options:HTMLWindowCreateOptions):HTMLLoader   {
```

```
            var bounds:Rectangle = new Rectangle(options.x, options.y,
            ➥options.width, options.height);
            var loader:HTMLLoader = HTMLLoader.createRootWindow(true,
            ➥null, true, bounds);
            return loader;
        }

    }
}
```

Методы `windowClose()`, `windowFocus()` и `windowBlur()` не возвращают значений и не принимают параметров. В листинге 7.17 приведена простая реализация `windowClose()`, закрывающая основное окно приложения.

Листинг 7.17. Закрытие окна в реализации метода windowClose( )

```
package com.manning.airinaction.html {
    import flash.html.HTMLHost;
    import flash.desktop.NativeApplication;
    import flash.html.HTMLWindowCreateOptions;
    import flash.html.HTMLLoader;
    import flash.geom.Rectangle;

    public class CustomHTMLHost extends HTMLHost {
        public function CustomHTMLHost(defaultBehavior:Boolean = true) {
            super(defaultBehavior);
        }

        override public function updateTitle(title:String):void {
            NativeApplication.nativeApplication.openedWindows[0].title =
            ➥title;
        }

        override public function
        ➥createWindow(options:HTMLWindowCreateOptions):HTMLLoader {
            var bounds:Rectangle = new Rectangle(options.x, options.y,
            ➥options.width, options.height);
            var loader:HTMLLoader = HTMLLoader.createRootWindow(true,
            ➥null, true, bounds);
            return loader;
        }

        override public function windowClose():void {
            NativeApplication.nativeApplication.openedWindows[0].close();
        }

    }
}
```

Свойство `windowRect` можно переопределить, заменив сеттер с тем же именем. Свойство `windowRect` имеет тип `Rectangle`. Листинг 7.18 иллюстрирует, как можно воспользоваться свойством `windowRect`, чтобы переместить или изменить размер главного окна приложения.

Листинг 7.18. Изменение размеров окна с помощью реализации сеттера windowRect

```
package com.manning.airinaction.html {
   import flash.html.HTMLHost;
   import flash.desktop.NativeApplication;
   import flash.html.HTMLWindowCreateOptions;
   import flash.html.HTMLLoader;
   import flash.geom.Rectangle;

   public class CustomHTMLHost extends HTMLHost {

      override public function set windowRect(value:Rectangle):void {
         NativeApplication.nativeApplication.openedWindows[0].bounds =
         ➥value;
      }

      public function CustomHTMLHost(defaultBehavior:Boolean = true) {
         super(defaultBehavior);
      }

      override public function updateTitle(title:String):void {
         NativeApplication.nativeApplication.openedWindows[0].title =
         ➥title;
      }

      override public function
      ➥createWindow(options:HTMLWindowCreateOptions):HTMLLoader   {
         var bounds:Rectangle = new Rectangle(options.x, options.y,
         ➥options.width, options.height);
         var loader:HTMLLoader = HTMLLoader.createRootWindow(true,
         ➥null, true, bounds);
         return loader;
      }

      override public function windowClose():void {
         NativeApplication.nativeApplication.openedWindows[0].close();
      }

   }
}
```

Весь этот код можно проверить, загрузив страницу HTML в объект `HTMLLoader`, в котором применен экземпляр `CustomHTMLHost`. Листинг 7.19 иллюстрирует пример класса документа, в котором это осуществлено.

Листинг 7.19. Применение CustomHTMLHost в классе документа

```
package com.manning.airinaction.html {

   import flash.display.MovieClip;
   import flash.html.HTMLLoader;
   import flash.net.URLRequest;
   import com.manning.airinaction.html.CustomHTMLHost;

   public class Main extends MovieClip {
```

```
        private var _htmlLoader:HTMLLoader;

        public function Main() {
            _htmlLoader = new HTMLLoader();
            _htmlLoader.width = stage.stageWidth;
            _htmlLoader.height = stage.stageHeight;
            _htmlLoader.htmlHost = new CustomHTMLHost();
            _htmlLoader.load(new URLRequest("example.html"));
            addChild(_htmlLoader);
        }

    }

}
```

В качестве HTML можно взять файл, показанный в листинге 7.20.

Листинг 7.20. Специальная реализация HTMLHost

```
<html>
   <body>
      <button onclick="window.document.title='New Title'">Title</button>
      <button onclick="window.open('http://www.manning.com/lott', null,
      ➥'screenX=0, screenY=10, width=500, height=400')">Window</button>
      <button onclick="window.close()">Close</button>
      <button onclick="window.moveTo(0, 0)" />Reset Location</button>
   </body>
</html>
```

Конечно, не обязательно реализовывать методы `HTMLHost` именно так, как показано в этом примере. Можно реализовывать их разными способами, добиваясь различных эффектов. Но общие принципы верны независимо от конкретной реализации.

Примечание

Объект `HTMLHost` (или экземпляр подкласса) автоматически получает ссылку на объект `HTMLLoader`, которому он присвоен. Эта ссылка хранится в свойстве под именем `htmlLoader`.

С помощью `HTMLHost` можно косвенно вызывать методы ActionScript из JavaScript. В следующем разделе мы увидим способ более прямой ссылки на элементы ActionScript из JavaScript.

7.5.5. Ссылки на элементы ActionScript из JavaScript

Можно не только ссылаться на элементы JavaScript и HTML из ActionScript, но и ссылаться на элементы ActionScript из JavaScript. Когда страница HTML загружена в приложение AIR, все возможности AIR API, включая стандартные возможности Flash Player API, доступны из JavaScript. Доступ к классам и функциям исполнительной среды осуществляется из `window.runtime`. Например, если вы хотите вызвать глобальный метод `trace()`, это можно сделать, как `window.runtime.trace()`.

Если классы находятся в пакетах, можно ссылаться на них с помощью полностью специфицированного имени, следующего за `window.runtime`. Например, листинг 7.21 показывает, как создать новый объект `Shape` и добавить его на сцену главного окна.

Листинг 7.21. Создание объекта Shape и помещение его на сцену

```
<html>
   <script>

      function loadHandler() {
         var shape = new window.runtime.flash.display.Shape();
         shape.graphics.lineStyle(0, 0, 0);
         shape.graphics.beginFill(0, 1);
         shape.graphics.drawRect(0, 0, 100, 50);
         shape.graphics.endFill();
         var mainWindow = window.runtime.flash.desktop.
         ➥NativeApplication.nativeApplication.openedWindows[0];
         mainWindow.stage.addChild(shape);
      }

   </script>
   <body onload="loadHandler()">
   </body>
</html>
```

Обратите внимание, как в первой строке функции мы создаем новый объект `Shape` с помощью ссылки `window.runtime.flash.display.Shape` на класс `Shape`. После создания объекта можно обращаться ко всем его свойствам и методам так же, как в ActionScript. В данном примере мы обращаемся к свойству `graphics` объекта `Shape` и вызываем методы этого свойства. Можно также получить ссылку на главное окно с помощью `window.runtime.flash.desktop.NativeApplication.nativeApplication.openedWindows[0]`.

Можно ссылаться не только на стандартные, но и на пользовательские классы. Чтобы сделать пользовательские классы доступными JavaScript, нужно загрузить страницу HTML в тот же самый домен приложения `ApplicationDomain`, где определены классы ActionScript. Если вы не знакомы с понятием `ApplicationDomain`, не удивляйтесь. У большинства разработчиков ранее не было особых причин прибегать к `ApplicationDomain`. В принципе, `ApplicationDomain` – это раздел, в котором хранится код. Обычно при загрузке .swf или HTML в приложение AIR загрузка осуществляется в новый `ApplicationDomain`, отличный от `ApplicationDomain`, используемого главным приложением AIR. В результате загруженный контент не имеет доступа к коду, содержащемуся в главном `ApplicationDomain`. Если вы хотите, чтобы пользовательские классы ActionScript были доступны в JavaScript, вы можете загрузить контент HTML в главный `ApplicationDomain` путем задания для свойства `runtimeApplicationDomain` объекта `HTMLLoader` значения `flash.system.ApplicationDomain.currentDomain`.

Мы сейчас рассмотрим пример обращения к пользовательским классам из JavaScript. Будем исходить из примера опросника, рассматривавшегося в предыдущем разделе. Теперь мы создадим пользовательский тип `Response` для хранения ответов на вопросы. Этот класс показан в листинге 7.22.

Листинг 7.22. Пользовательский класс Response, моделирующий ответы вопросника

```
package com.manning.airinaction.questionnaire {

   public class Response {

      private var _question:String;
      private var _answer:String;

      public function get question():String {
         return _question;
      }

      public function get answer():String {
         return _answer;
      }

      public function Response(question:String, answer:String) {
         _question = question;
         _answer = answer;
      }

   }

}
```

В классе `Response` есть два метода чтения свойств: `question` и `answer`. Модифицируем файл questionnaire.html, чтобы хранить ответы, получаемые функцией `getSurveyResponses()`, в виде значений этого типа. Листинг 7.23 показывает, как будет теперь выглядеть функция `getSurveyResponses()` (изменения выделены жирным шрифтом).

Листинг 7.23. Использование класса Response в JavaScript

```
function getSurveyResponses() {
   var response;
   var options = document.questionnaire.skyColor;
   for(var i = 0; i < options.length; i++) {
      if(options[i].checked) {
         response = options[i].value;
      }
   }
   var response1 = new window.runtime.com.manning.
   ➥airinaction.questionnaire.Response(document.
   ➥getElementById("question1").innerText, response);

   options = document.questionnaire.skySize;
   for(i = 0; i < options.length; i++) {
      if(options[i].checked) {
```

```
            response = options[i].value;
        }
    }
    var response2 = new window.runtime.com.manning.
    ➥airinaction.questionnaire.Response(document.
    ➥getElementById("question2").innerText, response);

    return [response1, response2];
}
```

Если попытаться сейчас запустить это приложение, то мы получим ошибки этапа исполнения JavaScript, из-за которых приложение молча аварийно завершится, поскольку класс `Response` не определен на этапе исполнения. Чтобы исправить это, нужно модифицировать класс документа, добавив в него класс `Response` и загрузку страницы HTML в главный `ApplicationDomain`. Листинг 7.24 показывает, как теперь будет выглядеть класс, с изменениями, выделенными жирным шрифтом.

Листинг 7.24. Включение нового класса Response

```
package com.manning.airinaction.questionnaire {

    import  flash.display.MovieClip;
    import  flash.events.MouseEvent;
    import  flash.html.HTMLLoader;
    import  flash.net.URLRequest;
    import  flash.events.Event;
    import  fl.controls.TextArea;
    import  flash.system.ApplicationDomain;
    import  com.manning.airinaction.questionnaire.Response;

    public class Main extends MovieClip {

        private var _htmlLoader:HTMLLoader;
        private var _textArea:TextArea;

        public function Main() {
            _htmlLoader = new HTMLLoader();
            _htmlLoader.width = stage.stageWidth;
            _htmlLoader.height = stage.stageHeight;
            _htmlLoader.addEventListener(Event.COMPLETE, completeHandler);
            _htmlLoader.runtimeApplicationDomain =
            ➥ApplicationDomain.currentDomain;
            _htmlLoader.load(new URLRequest("questionnaire.html"));
            addChild(_htmlLoader);

            _textArea = new TextArea();
            _textArea.width = stage.stageWidth;
            _textArea.height = stage.stageHeight;
        }

        private function completeHandler(event:Event):void {
            _htmlLoader.window.document.getElementById("submitButton").
          ➥addEventListener(MouseEvent.CLICK, clickHandler);
        }
```

```
        private function clickHandler(event):void {
            removeChild(_htmlLoader);
            var responses:Array = _htmlLoader.window.getSurveyResponses();
            _textArea.text = "";
            var response:Object;
            for(var i:Number = 0; i < responses.length; i++) {
                response = responses[i] as Response;
                _textArea.text += response.question;
                _textArea.text += "\n\t" + response.answer + "\n\n";
            }
            addChild(_textArea);
        }
    }
}
```

Теперь, когда класс `Response` включен в приложение AIR, он доступен на этапе исполнения. Так как страница HTML загружается в главный `ApplicationDomain`, она может обращаться к классу `Response`, и приложение заработает.

Примечание

По соображениям безопасности, излагаемым в следующем разделе, нельзя разрешать доступ к пользовательским классам из файлов, загруженных не из домена приложения. Например, предыдущее приложение не будет работать, если загружать questionnaire.html с удаленного сервера, т. к. AIR не позволит удаленным файлам обращаться к пользовательским классам ActionScript.

Мы еще не обсудили особенности загрузки HTML с различных адресов; мы займемся этим в следующем разделе.

7.6. Проблемы безопасности

Представьте себе, что вы – опытный водопроводчик. Вас только что пригласила компания MegaCorp на должность главного водопроводчика, что достаточно престижно. Однако в первый же день своего выхода на работу вы обнаруживаете, что отвечаете не только за обычные водопроводные дела, которые входят в вашу сферу компетенции, но и за всю систему безопасности MegaCorp, у которой есть офисы во всех крупных городах мира. Смешно, не правда ли? Нам кажется, что смешно. Однако эта ситуация не лишена сходства с положением, в котором вы оказываетесь сначала как веб-разработчик, а потом как разработчик приложений AIR. Хотя наш основной опыт может быть связан с разработкой приложений, мы обнаруживаем, что на нас ложится другая и очень серьезная обязанность: ответственность за безопасность этих приложений и их пользователей. Несмотря на несправедливость такого положения дел (в чем мы уверены), весьма похоже, что разделить обязанности разработчика приложений и инженера по безопасности приложений полностью не удастся. Следовательно, даже ес-

ли проблемы безопасности не входят в данный момент в круг ваших специальных знаний, мы настоятельно рекомендуем внимательно прочесть несколько следующих разделов, в которых мы рассмотрим, как эти проблемы связаны с разработкой приложений AIR.

AIR открывает многочисленные возможности для контента HTML и JavaScript, загружаемого в приложения AIR. Как вы уже видели, AIR позволяет JavaScript обращаться к исполнительной среде AIR, получая доступ к различным режимам, которые в обычных условиях для JavaScript недоступны. Таким образом, представьте себе, что будет, если вы построите приложение AIR, которое разрешит пользователю загружать любой HTML с любого адреса, и на какой-нибудь из этих страниц окажется зловредный код JavaScript, имеющий целью установку с помощью исполнительной среды AIR вируса на компьютере пользователя. Если все страницы HTML смогут получить такой неограниченный доступ к AIR, вы невольно создадите массу проблем для пользователей. По этой причине AIR предлагает определенную модель безопасности, которая должна уменьшить риск возникновения такого рода проблем.

7.6.1. Песочницы

Весь контент HTML загружается AIR в одну из двух «безопасных песочниц» (sandbox), для каждой из которых есть свои правила относительно того, что разрешается контенту HTML/JavaScript. Эти две песочницы называются «песочницей приложения» (application sandbox) и «песочницей не-приложения» (nonapplication sandbox). Весь контент, загружаемый из каталога приложения (того, в который установлено приложение, или его подкаталога), автоматически помещается в домен приложения (application domain). Весь остальной контент помещается в домен не-приложения (nonapplication domain). Таким образом, любой контент, загружаемый с веб-сервера, помещается в домен не-приложения.

В каждой из песочниц действуют свои правила в отношении разрешенных действий. В песочнице приложения накладываются ограничения на динамические действия, осуществляемые на этапе исполнения. В песочнице приложения действуют ограничения:

- Не разрешается выполнять команду `eval()` с аргументами, отличными от литералов объектов и констант.
- Функции `setTimeout()` и `setInterval()` могут использоваться только для вызова литералов функций и не вычисляют строки.
- Запрещается анализировать элементы скриптов с помощью `innerHTML` или `outerHTML`.
- Нельзя пользоваться схемой URI `javascript`.
- Нельзя импортировать файлы JavaScript из-за пределов области приложения.

Эти ограничения введены как защитный слой против загрузки и выполнения злонамеренного кода JavaScript. Хотя эти ограничения не гарантируют, что зловредный код никогда не будет загружен, они все же создают разумный уровень защиты. Это важно, потому что внутри области приложения у JavaScript есть доступ ко всему AIR API, благодаря чему он может читать и писать файлы в локальной файловой системе.

Напротив, для файлов, загружаемых из-за пределов домена приложения, нет ограничений, налагаемых на файлы, загружаемые из домена приложения. Вы можете решить, что крайне несправедливо ограничивать в правах файлы из домена приложения и в то же время разрешать это же поведение файлам за его пределами. Не стоит расстраиваться по этому поводу, поскольку здесь есть некоторый компромисс. Видите ли, файлы, загруженные из-за пределов домена приложения, не имеют доступа к AIR API. Это означает, что хотя такой файл может беспрепятственно выполнять команды `eval()`, он не имеет доступа к локальной файловой системе.

Мы хотим рассказать вам, как можно обойти ограничения, налагаемые песочницами. Но прежде чем мы раскроем вам эти сведения, подчеркнем важность применения этой техники только в условиях абсолютной необходимости и только с крайней степенью осторожности. При обходе модели защищенных песочниц лучше проявить консервативность. Не забывайте, что песочницы устроены по веским причинам, и ломать их без веских оснований не следует. С учетом всего сказанного, в случае абсолютной необходимости можно обойти песочницу с помощью технологии, называемой «sandbox bridging» (шунтирование).

7.6.2. Шунтирование песочниц

Принцип шунтирования песочниц основан на том, что страница HTML, загружаемая в iframe другой страницы HTML, может обмениваться данными с родителем, а родитель может обмениваться данными с потомком. Если две страницы загружены в разные песочницы (например, когда одна из них удаленная, а другая взята из каталога приложения), то они могут наладить между собой взаимодействие с целью обойти ограничения каждой из них. Обычно для этого создают файл HTML с iframe, находящийся в каталоге приложения. Этот файл будет загружен в песочницу приложения. Этот локальный файл HTML загружает файл HTML не-приложения (например, с удаленного сервера) в свой iframe. Файл HTML не-приложения загружается в песочницу не-приложения. После того как обе страницы загружены, они могут обмениваться данными через интерфейсы, которые предоставляют друг другу.

Интерфейсы, которые каждая из страниц может использовать совместно с другой, можно определить с помощью переменных `parentSandboxBridge` и `childSandboxBridge`. Вы создаете объект со свойствами, содержащими значения или ссылки на функции, и присваиваете его переменной с именем `parentSandboxBridge` или `childSandboxBridge` внутри контента

HTML, загруженного в iframe. Ниже приводится пример, иллюстрирующий эту идею. Листинг 7.25 показывает страницу HTML, в которой есть iframe. Эту страницу HTML можно сохранить в каталоге приложения. После загрузки она будет иметь доступ к AIR API.

Листинг 7.25. Локальный файл HTML, содержащий iframe для загрузки удаленного контента

```
<html>
   <script>

      var bridge = new Object();
      bridge.writeMemo = writeMemo;

      function loadHandler() {
         window.document.getElementById("bridgeFrame").
         ➥contentWindow.parentSandboxBridge = bridge;
      }

      function writeMemo(subject, message) {
         var file = window.runtime.flash.filesystem.File.
         ➥documentsDirectory.resolvePath("memo.txt");
         var writer = new window.runtime.flash.filesystem.FileStream;
         writer.open(file, "write");
         writer.writeUTFBytes(subject + "\n" + message);
         writer.close();
      }

   </script>
   <body onload="loadHandler()">
      <iframe id="bridgeFrame"
            src="http://www.example.com/texteditor.html"
            width="100%" height="100%"></iframe>
   </body>
</html>
```

❶ Создать объект «шунт»

❷ Присвоить шунт дочернему объекту

Как видно из этого примера, создается объект, работающий в качестве шунта ❶. После загрузки страницы присваиваем шунт переменной с именем `parentSandboxBridge` в окне контента iframe ❷. Теперь контент в iframe сможет с помощью шунта вызвать функцию `writeMemo()`. В листинге 7.26 приведен код этого контента.

Листинг 7.26. Вызов функции с помощью шунта

```
<html>
   <body>
         <button onclick="parentSandboxBridge.writeMemo(subject.value,
         ➥message.value)">Save Memo</button>
      <form>
         Subject <input type="text" id="subject" /><br />
         Message <textarea id="message" /><br />
      </form>
   </body>
</html>
```

При щелчке пользователя по кнопке мы вызываем функцию parentSandboxBridge.writeMemo(). При этом шунт позволяет контенту фрейма, загруженному в песочницу не-приложения, вызвать функцию внутри родительской страницы HTML, которая находится в песочнице приложения. В результате мы получаем возможность сохранить файл локально, несмотря на то, что контент этого файла получен со страницы HTML, находящейся вне области приложения.

С помощью переменной childSandboxBridge можно сделать доступными родительской странице значения и функции дочерней страницы, что осуществляется почти идентичным образом. На странице контента вы определяете объект со свойствами, которые содержат значения или ссылки на функции, а затем присваиваете этот объект переменной с именем childSandboxBridge. Теперь с родительской страницы можно обращаться к childSandboxBridge из окна контента iframe, как только он будет загружен.

7.7. Добавление HTML в AirTube

Теперь, узнав о работе с HTML в AIR больше, чем можно было вначале предположить, добавим в наше приложение AirTube новую функцию. После всего нами изученного сделать это будет на редкость просто. Мы хотим лишь дать пользователю возможность открывать веб-страницу YouTube для фильма в окне AIR. Мы добавим в окно видео кнопку, которая будет открывать страницу HTML в окне.

Сначала изменим компоненту HTMLWindow, так чтобы она показывала HTML с помощью компоненты HTML. Для этого откроем файл HTMLWindow.mxml и добавим в него код, показанный жирным шрифтом в листинге 7.27.

Листинг 7.27. Показ контента HTML в компоненте HTMLWindow

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
   width="800" height="800" closing="closingHandler(event);">
   <mx:Script>
      <![CDATA[

         [Bindable]
         private var _url:String;

         public function set url(value:String):void {
            _url = value;
         }

         private function closingHandler(event:Event):void {
            event.preventDefault();
            visible = false;
         }

      ]]>
   </mx:Script>
```

```
    <mx:HTML id="html" location="{_url}" width="100%" height="100%" />
</mx:Window>
```

Как видите, нового кода немного. Мы просто добавили компоненту HTML и связали ее атрибут location со свойством _url. Теперь посмотрим, как задать URL для компоненты HTMLWindow.

Вспомните, как в главе 2, когда мы создавали первоначальный скелет HTMLWindow, мы также поместили в AirTube.mxml метод, открывающий экземпляр HTMLWindow. Этот метод называется launchHTMLWindow() и выглядит так:

```
public function launchHTMLWindow(url:String):void {
   if(_htmlWindow.nativeWindow == null) {
        _htmlWindow.open();
     }
     else {
         _htmlWindow.activate();
     }
}
```

Мы изменим этот метод, добавив в него одну строку. В листинге 7.28 новая строка выделена жирным шрифтом.

Листинг 7.28. Задание свойства url объекта HTMLWindow

```
public function launchHTMLWindow(url:String):void {
   _htmlWindow.url = url;
   if(_htmlWindow.nativeWindow == null) {
        _htmlWindow.open();
     }
     else {
         _htmlWindow.activate();
     }
}
```

Теперь осталось только вызвать метод launchHTMLWindow() при щелчке пользователя по кнопке в VideoWindow. Для этого нужно внести изменения в компоненту VideoWindow. Листинг 7.29 показывает, как выглядит обновленный код, в котором изменения выделены жирным шрифтом.

Листинг 7.29. Добавление кнопки для открытия HTML из окна видео

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml" width="400"
   height="400" type="utility" closing="closingHandler(event);"
   creationComplete="creationCompleteHandler();">
   <mx:Script>
      <![CDATA[
         import com.manning.airtube.services.AirTubeService;
         import com.manning.airtube.data.ApplicationData;

         [Bindable]
         private var _applicationData:ApplicationData;
```

```
        private function creationCompleteHandler():void {
            _applicationData = ApplicationData.getInstance();
        }

        private function closingHandler(event:Event):void {
            event.preventDefault();
            visible = false;
        }

        private function saveOffline():void {
            AirTubeService.getInstance().saveToOffline(
            ➥_applicationData.currentVideo);
        }

        private function togglePlayback():void {
            if(videoDisplay.playing) {
                videoDisplay.pause();
                playPauseButton.label = "Play";
            }
            else {
                videoDisplay.play();
                playPauseButton.label = "Pause";
            }
        }

        private function viewOnYouTube():void {
            AirTube.getInstance().launchHTMLWindow(
            ➥_applicationData.currentVideo.video.url);
            videoDisplay.pause();
            playPauseButton.label = "Play";
        }

    ]]>
</mx:Script>
<mx:VBox>
    <mx:Label text="{_applicationData.currentVideo.video.title}" />
    <mx:VideoDisplay id="videoDisplay"
source="{_applicationData.currentVideo.flvUrl}" width="400" height="300"
/>
    <mx:HBox id="progressContainer" width="100%"
        visible="{_applicationData.downloadProgress > 0}"
        includeInLayout="{progressContainer.visible}">
        <mx:Label text="download progress" />
        <mx:HSlider id="progressIndicator" enabled="false"
         width="100%" minimum="0" maximum="1"
         value="{_applicationData.downloadProgress}" />
    </mx:HBox>
    <mx:HBox>
        <mx:Button id="playPauseButton" label="Pause"
            click="togglePlayback();" />
        <mx:Button id="saveOfflineButton" label="Save Offline"
         visible="{!_applicationData.currentVideo.offline}"
         click="saveOffline();"
```

❶ Открытие окна HTML

❷ Пауза видео

```
        enabled="{!(_applicationData.downloadProgress > 0)}" />
      <mx:Button label="View On YouTube"
        visible="{_applicationData.online}"
        click="viewOnYouTube();" />
      </mx:HBox>
    </mx:VBox>
  </mx:Window>
```

❸ Добавить кнопку

Этот новый код добавляет кнопку ❸, при щелчке по которой вызывается `viewOnYouTube()`. Когда пользователь щелкает по кнопке, мы сначала вызываем метод `launchHTMLWindow()` экземпляра приложения ❶, передавая ему URL страницы YouTube, хранящийся в объекте `AirTubeVideo` для текущего фильма. И на всякий случай мы также приостанавливаем показ ❷.

После этих изменений приложение может выглядеть примерно так, как на рис. 7.9.

***Рис. 7.9.** С помощью новой кнопки можно открывать соответствующую страницу HTML*

Теперь у нас есть почти готовое приложение AirTube. Осталось только добавить в него возможность запускать файлы .atv двойным щелчком, что мы сделаем в следующей главе.

7.8. Резюме

В этой главе мы вихрем промчались по всевозможным деталям работы с HTML в AIR. Мы начали с основ: загрузки и показа контента HTML в приложениях AIR на базе Flash или Flex. Мы изучили, как это делается с помощью класса `HTMLLoader` или компоненты `HTML`. Затем мы научились управлять загрузкой и контентом HTML в AIR, в том числе кэшировать контент и обрабатывать запрос данных регистрации с сервера. Мы также рассмотрели обширную тему перекрестного вызова ActionScript и JavaScript и научились не только управлять элементами JavaScript и HTML из ActionScript, но и обращаться к элементам ActionScript из JavaScript. Затем, прежде чем завершить главу дополнениями к приложению AirTube, мы рассмотрели проблемы безопасности работы с HTML в AIR.

Эта глава знаменует собой определенный рубеж, достигнутый в нашем изложении. Она – последняя, посвященная исключительно добавлению новых функций в приложения AIR. В следующей главе мы свяжем воедино все, чему научились к этому времени, и соберем это в виде приложения, готового к распространению. Затем мы рассмотрим существующие стратегии развертывания и обновления приложений.

В этой главе:

- Понятие о распространении приложений
- Создание значков приложений AIR
- Обновление приложений
- Обработка различных способов вызова приложений

8

Распространение и обновление приложений AIR

Вы теперь очень многое знаете об Adobe AIR. Пользуясь тем, с чем вы познакомились в предыдущих главах, вы, вероятно, уже создали немало приложений (типа AirTube), или, по крайней мере, у вас появились идеи для создания каких-то замечательных приложений. В каком бы состоянии ни находились эти приложения – в завершенном или на стадии замысла – в какой-то момент вы столкнетесь со следующими вопросами:

- Как лучше всего поставлять ваше приложение пользователям?
- Как лучше всего пользователю устанавливать приложение?
- Как гарантировать пользователю работу с последней версией вашего приложения?
- Как распространять обновления?
- Какими способами пользователь может запустить ваше приложение?

В этой главе мы подробно рассмотрим все перечисленные вопросы. Начнем с того, как лучше распространять ваши приложения и устанавливать их у пользователей.

8.1. Распространение приложений

Все приложения AIR должны устанавливаться пользователем на его машине. Это означает, что пользователь должен загрузить файл .air приложения и запустить его. Есть два основных пути, как это может происходить. Один путь – просто разрешить пользователю загрузить ваш файл .air. При этом снабдить пользователя инструкцией, в которой сказано, что нужно загрузить файл .air, а потом сделать на нем двойной

щелчок для установки. Это вполне приемлемая схема для пользователей средней и высокой квалификации, которых она вряд ли смутит. Но для пользователей, которые не знакомы с AIR, данный простой путь может оказаться не самым подходящим. Примите во внимание, что этот простой подход предполагает, что на машине пользователя уже установлена исполнительная среда AIR. В противном случае пользователь может прийти в замешательство, если после двойного щелчка по файлу .air ничего не произойдет или возникнет что-то неожиданное.

Второй подход к распространению приложений AIR требует от разработчика немного больше усилий, но создает более комфортные условия для пользователя. Второй подход называется *гладкой установкой* (*seamless install*), поскольку он автоматически определяет, есть ли у пользователя исполнительная среда AIR, и при необходимости устанавливает сначала ее, а потом приложение AIR. Кроме того, при гладкой установке загрузка и запуск файла .air происходят невидимо для пользователя. В этом разделе мы рассмотрим организацию гладкой установки.

8.1.1. Использование стандартного значка

Гладкая установка начинается на веб-странице. Пользователь запускает ее щелчком по т. н. значку, или *бейджу* (*badge*). Бейдж – это всего лишь файл .swf, содержащий код, который выполняет инсталляцию. Как мы увидим в следующем разделе, можно создавать свои пользовательские значки. Но сначала мы посмотрим, как работать со стандартным значком, который Adobe включила в состав AIR. Файлы значков по умолчанию можно найти в каталоге samples\badge в AIR SDK (находящемся в каталоге AIK установочного каталога Flash CS3 или в каталоге Flex 3 SDK). Рисунок 8.1 показывает, как выглядит стандартный значок с типовой картинкой, входящий в состав AIR SDK.

Главный файл, который вам нужен, – это badge.swf. В этом файле есть весь код, необходимый для гладкой установки. С помощью badge.swf можно провести установку любого приложения AIR, потому что badge.swf поддерживает передачу нужных ему параметров через FlashVars. Этим способом ему можно передать следующие переменные:

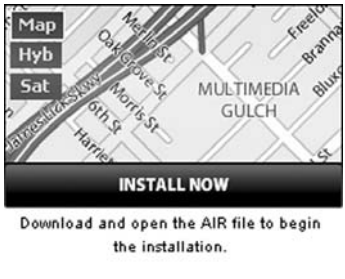


Рис. 8.1. *Так выглядит стандартный значок, с помощью которого пользователи могут выполнять гладкую установку приложений AIR*

- appname – имя приложения AIR, которое должен установить этот бейдж. Это имя бейдж покажет, если пользователю требуется установка исполнительной среды AIR.
- appurl – абсолютный URL файла .air.
- airversion – версия исполнительной среды AIR, требующаяся приложению. Для AIR 1.0 этим значением всегда должна быть строка 1.0.
- imageurl – URL графического файла, который должен загрузить и показать бейдж.
- buttoncolor – по умолчанию цвет кнопки черный, но вы можете задать другой цвет. Значение предается в виде шестнадцатеричной строки, например FF00FF.
- messagecolor – по умолчанию подпись под кнопкой черная. Вы можете задать предпочтительный цвет в виде шестнадцатеричной строки, например FF00FF.

Из всех этих переменных обязательны только две: appurl и airversion. Для остальных есть значения по умолчанию.

Хотя можно воспользоваться образцами HTML и JavaScript, предоставляемыми Adobe наряду со стандартным значком, чтобы встроить значок на страницу HTML, мы настоятельно рекомендуем встроить файл .swf на страницу HTML с помощью SWFObject и задать переменные FlashVars. Если вам незнаком объект SWFObject, вы можете получить о нем сведения и загрузить все необходимое со страницы *http://code.google.com/p/swfobject/*. Следующий код HTML/JavaScript показывает, как встроить badge.swf в страницу HTML с помощью SWFObject, задав appurl, airversion, buttoncolor и imageurl:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
➥"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
 <head>
   <title>AIR Application</title>
   <meta http-equiv="Content-Type" content="text/html;
   ➥charset=iso-8859-1" />

   <script type="text/javascript" src="swfobject.js"></script>

   <script type="text/javascript">
      var flashvars = new Object();
      flashvars.appurl =
      ➥"http://www.example.com/air/applications/example.air";
      flashvars.airversion = "1.0";
      flashvars.buttoncolor = "FF00FF";
      flashvars.imageurl = "image.jpg";
      swfobject.embedSWF("badge.swf", "badgeDiv", "217", "180",
      ➥"9.0.0", null, flashvars);
   </script>

 </head>
 <body>
```

```
    <div id="badgeDiv">
     <p>Alternative content</p>
    </div>
  </body>
</html>
```

Результат работы этого кода выглядит примерно как на рис. 8.2.

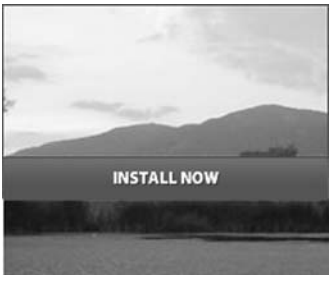


Рис. 8.2. *Вид значка по умолчанию можно настроить в соответствии со своими предпочтениями*

Это все, что необходимо для организации гладкой установки приложения AIR. Если вы разместите веб-страницу и badge.swf на своем веб-сайте, то пользователи смогут установить ваше приложение AIR без глубокого погружения в AIR. Процедура будет следующей:

1. Пользователь заходит на веб-сайт и щелкает по значку.
2. Бейдж определяет, стоит ли у пользователя нужная исполнительная среда AIR. Если установлена среда правильной версии, пользователь сразу попадает на шаг 4. В противном случае ему сообщается о необходимости установки исполнительной среды AIR и дается возможность установить ее, щелкнув по кнопке, как показано на рис. 8.3.
3. Если пользователь щелкнет по кнопке, чтобы установить исполнительную среду, она будет установлена таким образом, что ему не понадобится никуда перемещаться с бейджа. На рис. 8.4 показано, как это выглядит. По завершении установки пользователь автоматически попадает на шаг 4.
4. Автоматически загружается и выполняется файл .air, не требуя никаких действий пользователя вплоть до успешного запуска программы установки. После этого пользователю предъявляется серия экранов помощника, на которых от него требуется принять значения по умолчанию (например, для адреса установки) или указать свои собственные значения.
5. После окончания работы с помощником приложение оказывается успешно установленным.

Мы рассмотрели использование значка по умолчанию и его настройку. Далее мы узнаем, как создать собственный оригинальный значок.

Рис. 8.3. *Пользователь получает возможность установить исполнительную среду AIR, если ее нет на его машине*

Рис. 8.4. *Если пользователь соглашается на установку исполнительной среды, он видит диалоговое окно с индикацией хода загрузки и установки*

8.1.2. Создание собственного значка

Значок по умолчанию пригоден для любых приложений AIR, допуская определенную меру настройки внешнего вида, что делает его полезным во многих случаях. Но наверняка настанет момент, когда у вас возникнет желание или необходимость настолько серьезно переделать значок, что вам придется создать его с чистого листа. В этом разделе мы разберемся, как работают значки, чтобы вы при необходимости могли создать свой собственный.

Все значки должны загружать внешний файл .swf с сайта Adobe и работать с ним. URL этого файла – *http://airdownload.adobe.com/air/browserapi/air.swf*. В этом файле есть несколько методов ActionScript, необходимых значку для выполнения своих функций. Чтобы вызывать эти методы, загружающий .swf должен загрузить внешний .swf в тот же самый `ApplicationDomain`. Для этого нужно создать объект LoaderContext, указав `ApplicationDomain.currentDomain` в качестве значения его свойства `applicationDomain`, и передать этот объект `LoaderContext` методу `load()` объекта `Loader`, используемого для загрузки внешнего .swf. Это может показаться довольно запутанным. На практике тут нет ничего сложного. Нужен только код вроде следующего:

```
_loader = new Loader();
var context:LoaderContext = new LoaderContext();
context.applicationDomain = ApplicationDomain.currentDomain;
_loader.contentLoaderInfo.addEventListener(Event.INIT, initHandler);
_loader.load(new URLRequest("http://airdownload.adobe.com/air/
➥browserapi/air.swf"), context);
```

Когда происходит событие `init`, становятся доступны методы внешнего .swf, и можно вызывать их из свойства `content` объекта `Loader`. Это следующие методы:

- `getStatus()` – этот метод просто возвращает строку `available`, `unavailable` или `installed`. Значение `available` свидетельствует о том, что

исполнительная среда AIR доступна операционной системе, хотя и не установлена в данный момент. Значение `unavailable` указывает, что исполнительная среда AIR недоступна в данной операционной системе. Значение `installed` указывает, что среда AIR в настоящее время установлена.

- `getApplicationVersion()` – этот метод возвращает версию указанного приложения AIR, установленную в системе в данное время. Метод принимает три параметра: ID приложения и ID издателя приложения AIR, которое вы проверяете, а также ссылку на функцию обратного вызова. Так как метод работает асинхронно, ему необходима функция обратного вызова. При возврате результата функция обратного вызова получает в качестве параметра текущую версию установленного приложения. Если не установлено никакой версии, параметр имеет значение null.

Примечание

Вероятно, вы знаете, как выяснить ID приложения AIR. Вспомните: ID приложения указывается в файле дескриптора. Но определение ID издателя может вызвать у вас затруднение, поскольку его нет в файле дескриптора. ID издателя создается при генерации файла .air, когда `adt` создает уникальный ID издателя для сертификата. Следовательно, если используется один и тот же сертификат для нескольких приложений, у всех из них будет один и тот же ID издателя. Но как его получить? Его можно программно прочесть во время выполнения программы с помощью свойства `NativeApplication.nativeApplication.publisherID`.

- `installApplication()` – этот метод устанавливает приложение AIR из файла .air. Ему требуются два параметра: URL файла .air и необходимая версия исполнительной среды AIR. (Ее нужно указать в виде строки.) Установочные программы приложений AIR позволяют запускать приложения прямо из программы установки сразу после ее завершения. Если вы хотите передать запускаемой программе какие-то начальные параметры, их можно передать в качестве третьего параметра методу `installApplication()`. Этот третий параметр может быть массивом значений, которые вы хотите передать запускаемому приложению.
- `launchApplication()` – этот метод запускает приложение AIR (если оно установлено). Методу нужно передать хотя бы два параметра: ID приложения и ID издателя. Можно также задать третий параметр, являющийся массивом значений, которые нужно передать приложению при запуске.

Примечание

Чтобы запускать приложение из броузера, приложение AIR должно указать значение `true` для элемента `allowBrowserInvocation` в своем файле дескриптора.

Не каждый день приходится загружать внешний файл .swf в тот же самый `ApplicationDomain`. И не так легко запомнить методы из файла

air.swf и их параметры. Поэтому для работы с этими методами весьма удобно написать вспомогательный класс, который обеспечит загрузку файла .swf в тот же самый домен ApplicationDomain и предоставит API, который будет понятен IDE, такой как Flex Builder, чтобы среда разработки смогла давать синтаксическую подсказку. В листинге 8.1 приведен пример такого вспомогательного класса, который мы назвали AirBadgeService.

Листинг 8.1. Вспомогательный класс для работы с air.swf

```
package com.manning.airinaction.utilities {
   import flash.display.Loader;
   import flash.system.LoaderContext;
   import flash.system.ApplicationDomain;
   import flash.events.Event;
   import flash.events.EventDispatcher;
   import flash.net.URLRequest;

   public class AirBadgeService extends EventDispatcher {

        private var _loader:Loader;
      private var _service:Object;

      public function AirBadgeService() {

         _loader = new Loader();
         var context:LoaderContext = new LoaderContext();
         context.applicationDomain = ApplicationDomain.currentDomain;

         _loader.contentLoaderInfo.addEventListener(Event.INIT,
                                          initHandler);
         _loader.load(new URLRequest(
         ➥"http://airdownload.adobe.com/air/browserapi/air.swf"),
         ➥context);
      }

      private function initHandler(event:Event):void {
         _service = _loader.content;
         dispatchEvent(new Event(Event.COMPLETE));
      }

      public function getStatus():String {
         return _service.getStatus();
      }

      public function getApplicationVersion(applicationId:String,
     ➥publisherId:String,   callback:Function):void {
         _service.getApplicationVersion(applicationId,
                              publisherId,
                              callback);
      }

      public function installApplication(url:String,
                              runtimeVersion:String,
                              parameters:Array = null):void {
```

```
            _service.installApplication(url, runtimeVersion, parameters);
        }

        public function launchApplication(applicationId:String,
                                publisherId:String,
                                parameters:Array = null):void {
            _service.launchApplication(applicationId,
                               publisherId,
                               parameters);
        }

    }
}
```

Теперь вам остается только создать экземпляр `AirBadgeService`**, перехватить событие** `complete` **и вызвать методы, как в этом примере:**

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
   creationComplete="creationCompleteHandler()">
   <mx:Script>
      <![CDATA[
         import com.manning.airinaction.utilities.AirBadgeService;

         private var _badgeService:AirBadgeService;
         private function creationCompleteHandler():void {
            _badgeService = new AirBadgeService();
            _badgeService.addEventListener(Event.COMPLETE,
                                  completeHandler);
         }

         private function completeHandler(event:Event):void {
            textArea.text = "Detecting AIR runtime: " +
            ➥_badgeService.getStatus();
         }

      ]]>
   </mx:Script>
   <mx:TextArea id="textArea" />
</mx:Application>
```

Теперь вы видите, как просто распространять приложения AIR. Хотя для организации гладкой установки нужно приложить немного труда, все же она происходит легко и быстро. Теперь посмотрим, какие средства предлагает AIR для того, чтобы ваши пользователи всегда могли работать с самой свежей версией вашего приложения AIR.

8.2. Обновление приложений

Есть две основные стратегии обновления приложений: пассивная и активная. При пассивном подходе приложение не берет на себя обязанность проверять наличие более новых версий, уведомлять о них пользователя или содействовать установке обновлений. При активном под-

ходе приложение играет активную роль в предоставлении пользователю для работы новейшей версии приложения. Если придерживаться пассивного подхода, то от вас как от разработчика не требуется никакой дополнительной работы. Вам достаточно предоставлять доступ к новым файлам .air, содержащим новые версии приложений AIR, – пользователи смогут самостоятельно загружать и устанавливать их. Однако активный подход потребует от вас некоторых дополнительных усилий, о чем мы и расскажем в этом разделе.

Чтобы осуществлялось активное обновление версий, ваше приложение должно выполнять следующие действия:

1. Обратиться к сервису, который сообщает, какая версия приложения является последней. Этот сервис должен быть доступен в сети в виде сервиса AMF, REST, SOAP или аналогичного, чтобы ваше приложение AIR могло обратиться к нему с помощью стандартных сетевых функций Flash Player.
2. Сравнить последнюю версию с установленной и выяснить, требуется ли обновление.
3. Если требуется обновление, предложить пользователю его осуществить.
4. При согласии пользователя загрузить файл .air с помощью приемов, описанных в главе 3.
5. С помощью объекта `flash.desktop.Updater` запустить файл .air.

Шаги с 1 по 4 либо выходят за рамки данной книги (поскольку это базовые приемы работы с Flash или Flex), либо о них уже рассказывалось ранее. Сейчас нас интересует только шаг 5. Мы еще не сталкивались с классом `Updater` и сейчас обсудим его устройство и возможности применения.

У класса `Updater` единственная задача: обновлять приложение AIR. Чтобы воспользоваться объектом `Updater`, нужно лишь создать его экземпляр и вызвать метод `update()`, передав ему два параметра: ссылку на файл .air (который должен храниться локально на компьютере) и строку, содержащую версию приложения AIR. Вот пример создания ссылки на локально хранящийся файл .air и обновления с помощью объекта `Updater`:

```
var airFile:File = File.desktopDirectory.resolvePath(
➥"ExampleApplication_v2.air");
var updater:Updater = new Updater();
updater.update(airFile, "2.0");
```

В этом коде предполагается, что на машине пользователя есть файл ExampleApplication_v2.air и в его файле дескриптора указана версия приложения 2.0. Объект обновления предпринимает следующие действия:

1. Завершает работу текущей версии приложения AIR.

2. Проверяет, что ID приложения и ID издателя файла .air такие же, как у версии, которая только что выполнялась.
3. Проверяет, что версия, переданная методу `update()`, такая же, как версия, указанная в файле .air.
4. Устанавливает новую версию (если не произошло сбоя на каком-то из предыдущих шагов).
5. Запускает новую версию.

Если на каком-либо из шагов возникнет ошибка (например, различны строки номеров версий), вместо установки и запуска новой версии открывается прежняя версия.

Теперь мы создадим пример обновляющегося приложения. Это приложение будет состоять в основном из компоненты текстового поля, кнопки и кода для проверки необходимости обновления и запуска обновления, если этого захочет пользователь. Приложение показано на рис. 8.5.

Чтобы создать это приложение, выполните следующие действия:

1. Создайте файл latestversion.txt и запишите в него такой текст:

```
2.0,http://www.yourserver.com/UpdatableApplication_v2.air
```

   С помощью этого файла мы будем определять, какова последняя версия приложения. В данном случае мы указали, что последняя версия 2.0 и файл приложения UpdatableApplication_v2.air расположены на вашем сервере. Предполагается, что вы замените *www.yourserver.com* на доменное имя, к которому у вас есть доступ. Мы указали версию 2.0, чтобы она была больше, чем версия приложения, которое мы сначала напишем, установим и запустим.
2. Разместите файл latestversion.txt на веб-сервере и запомните URL, по которому его можно найти.
3. Создайте новый проект AIR под именем UpdatableApplication.
4. Создайте файл дескриптора приложения, как показано в листинге 8.2. Обратите внимание на номер версии – 1.0.

Рис. 8.5. *Обновляющееся приложение показывает пользователю сообщение и разрешает обновить приложение, если доступна его более новая версия*

Листинг 8.2. Файл дескриптора для приложения UpdatableApplication

```
<application xmlns="http://ns.adobe.com/air/application/1.0">
 <id>com.manning.airinaction.UpdatableApplication</id>
 <filename>UpdatableApplication</filename>
 <name>UpdatableApplication</name>
 <version>1.0</version>
 <initialWindow>
   <content>UpdatableApplication.swf</content>
 </initialWindow>
</application>
```

5. Создайте файл MXML приложения, назовите его UpdatableApplication.mxml и поместите в него код из листинга 8.3. Этот код задает базовую структуру приложения с компонентой текстового поля и кнопкой. Остальной код мы допишем позднее.

Листинг 8.3. Главный файл MXML обновляющегося приложения

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler()">
   <mx:Script>
      <![CDATA[

         private var _latestVersion:String;
         private var _airFileUrl:String;

         private function creationCompleteHandler():void {
         }

         private function updateApplication():void {
         }

      ]]>
   </mx:Script>
   <mx:VBox width="100%" height="100%">
      <mx:TextArea id="textArea" width="100%" height="80%" />
      <mx:Button id="updateButton" label="Update"
         enabled="false" click="updateApplication();" />
   </mx:VBox>
</mx:WindowedApplication>
```

Когда запустится приложение, мы хотим, чтобы оно загрузило с сервера файл latestversion.txt, проанализировало его данные и определило, совпадает ли версия текущего приложения с последней версией. Соответствующий код приведен в листинге 8.4.

Листинг 8.4. Загрузка данных с сервера и их анализ

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler()">
   <mx:Script>
      <![CDATA[
```

```
            private var _latestVersion:String;
            private var _airFileUrl:String;

            private function creationCompleteHandler():void {
               var loader:URLLoader = new URLLoader();
               loader.addEventListener(Event.COMPLETE, completeHandler);
               loader.load(new URLRequest(
               ➥"http://www.yourserver.com/latestversion.txt"));
            }

            private function completeHandler(event:Event):void {
               var loader:URLLoader = event.target as URLLoader;
               _latestVersion = loader.data.split(",")[0];
               var descriptor:XML =
               ➥NativeApplication.nativeApplication.applicationDescriptor;
               var air:Namespace = descriptor.namespaceDeclarations()[0];
               var currentVersion:String = descriptor.air::version;
               if(_latestVersion != currentVersion) {
                  _airFileUrl = loader.data.split(",")[1];
                  textArea.text = "You are running version " +
                  ➥currentVersion + ". However, version " +
                  ➥_latestVersion +
                  ➥" is available. Click the button to update.";
                  updateButton.enabled = true;
               }
               else {
                  textArea.text =
                  ➥"You appear to be running the latest version";
               }
            }

            private function updateApplication():void {
            }

         ]]>

      </mx:Script>
      <mx:VBox width="100%" height="100%">
         <mx:TextArea id="textArea" width="100%" height="80%" />
         <mx:Button id="updateButton" label="Update" enabled="false"
            click="updateApplication();" />
      </mx:VBox>
   </mx:WindowedApplication>
```

❶ Загрузить lastversion.txt

❷ Разобрать строку версии

❸ Сравнить версии

❹ Разобрать URL файла .air

Этот новый код загружает текст latestversion.txt ❶ и анализирует в нем строку версии ❷. Кроме того, мы получаем из дескриптора текущего приложения номер его версии с помощью `NativeApplication.nativeApplication.applicationDescriptor`. Если эти версии разные ❸, выделяем URL файла .air ❹ и активируем кнопку Update.

6. Добавьте код, с помощью которого пользователь сможет загрузить обновление приложения. Он приведен в листинге 8.5.

Листинг 8.5. Загрузка файла .air с помощью объекта URLStream

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler()">
   <mx:Script>
      <![CDATA[

         private var _latestVersion:String;
         private var _airFileUrl:String;

         private function creationCompleteHandler():void {
            var loader:URLLoader = new URLLoader();
            loader.addEventListener(Event.COMPLETE, completeHandler);
           loader.load(new  URLRequest("http://www.rightactionscript.com/
latestversion.txt"));
         }

         private function completeHandler(event:Event):void {
            var loader:URLLoader = event.target as URLLoader;
            _latestVersion = loader.data.split(",")[0];
            var descriptor:XML =
NativeApplication.nativeApplication.applicationDescriptor;
            var air:Namespace = descriptor.namespaceDeclarations()[0];
            var currentVersion:String = descriptor.air::version;
            if(_latestVersion != currentVersion) {
               _airFileUrl = loader.data.split(",")[1];
               textArea.text = "You are running version " +
               ➥currentVersion + ". However, version " + _latestVersion +
                ➥" is available. Click the button to update.";
               updateButton.enabled = true;
            }
            else {
               textArea.text =
               ➥"You appear to be running the latest version";
            }
         }

         private function updateApplication():void {
            var stream:URLStream = new URLStream();
            stream.addEventListener(ProgressEvent.PROGRESS,
                                    progressHandler);
            stream.addEventListener(Event.COMPLETE,
                                    downloadCompleteHandler);
            stream.load(new URLRequest(_airFileUrl));
            textArea.text = "Downloading update";
         }

         private function progressHandler(event:ProgressEvent):void {
         }

         private function downloadCompleteHandler(event:Event):void {
         }
      ]]>
```

```
    </mx:Script>
    <mx:VBox width="100%" height="100%">
       <mx:TextArea id="textArea" width="100%" height="80%" />
       <mx:Button id="updateButton" label="Update" enabled="false"
          click="updateApplication();" />
    </mx:VBox>
</mx:WindowedApplication>
```

Этот код загружает файл .air с помощью объекта URLStream. Мы также добавляем обработчики событий `progress` и `complete`.

7. Добавьте код обработки событий `progress` и `complete`. Он приведен в листинге 8.6.

Листинг 8.6. Показ индикатора загрузки и запуск обновления, когда оно становится возможным

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler()">
   <mx:Script>
      <![CDATA[

         private var _latestVersion:String;
         private var _airFileUrl:String;

         private function creationCompleteHandler():void {
            var loader:URLLoader = new URLLoader();
            loader.addEventListener(Event.COMPLETE, completeHandler);
            loader.load(new URLRequest("http://
www.rightactionscript.com/latestversion.txt"));
         }

         private function completeHandler(event:Event):void {
            var loader:URLLoader = event.target as URLLoader;
            _latestVersion = loader.data.split(",")[0];
            var descriptor:XML =
NativeApplication.nativeApplication.applicationDescriptor;
            var air:Namespace = descriptor.namespaceDeclarations()[0];
            var currentVersion:String = descriptor.air::version;
            if(_latestVersion != currentVersion) {
               _airFileUrl = loader.data.split(",")[1];
               textArea.text = "You are running version " +
      ➥currentVersion + ". However, version " + _latestVersion +
      ➥" is available. Click the button to update.";
      updateButton.enabled = true;
   }
   else {
      textArea.text =
      ➥"You appear to be running the latest version";
   }
}

private function updateApplication():void {
   var stream:URLStream = new URLStream();
```

```
        stream.addEventListener(ProgressEvent.PROGRESS,
                        progressHandler);
        stream.addEventListener(Event.COMPLETE,
                        downloadCompleteHandler);
        stream.load(new URLRequest(_airFileUrl));
        textArea.text = "Downloading update";
    }

    private function progressHandler(event:ProgressEvent):void {
        textArea.text = "Downloading update " +
        ➥event.bytesLoaded + " of " + event.bytesTotal + " bytes";
    }

    private function downloadCompleteHandler(event:Event):void {
        textArea.text = "Download complete";
        var urlStream:URLStream = event.target as URLStream;
        var file:File =
        ➥File.applicationStorageDirectory.resolvePath(
        ➥"newVersion.air");
        var fileStream:FileStream = new FileStream();
        fileStream.open(file, FileMode.WRITE);
        var bytes:ByteArray = new ByteArray();
        urlStream.readBytes(bytes);
        fileStream.writeBytes(bytes);
                fileStream.close();
                var updater:Updater = new Updater();
                updater.update(file, _latestVersion);
            }
        ]]>

        </mx:Script>
        <mx:VBox width="100%" height="100%">
            <mx:TextArea id="textArea" width="100%" height="80%" />
            <mx:Button id="updateButton" label="Update" enabled="false"
                click="updateApplication();" />
        </mx:VBox>
    </mx:WindowedApplication>
```

Этот новый код демонстрирует пользователю ход загрузки, и когда файл доступен, с помощью `File` и `FileStream` мы записываем его на диск. Затем с помощью объекта `Updater` осуществляем обновление до последней версии.

8. Создайте файл .air для приложения `UpdatableApplication` и инсталлируйте это приложение на своей машине. После запуска оно должно сообщить, что вы работаете с версией 1.0, но есть версия 2.0, и предложить вам загрузить и установить обновление. Обновлять не нужно, потому что мы еще не создали обновленной версии.
9. Создайте новый проект AIR с именем UpdatableApplication_v2.
10. Создайте файл дескриптора этого приложения, показанный в листинге 8.7. Заметьте, что он точно такой же, как для версии 1.0, за исключением строки версии. Важно повторить ID приложения.

Листинг 8.7. Файл дескриптора для UpdatableApplication_v2

```
<application xmlns="http://ns.adobe.com/air/application/1.0">
 <id>com.manning.airinaction.UpdatableApplication</id>
 <filename>UpdatableApplication</filename>
 <name>UpdatableApplication</name>
 <version>2.0</version>
 <initialWindow>
   <content>UpdatableApplication.swf</content>
 </initialWindow>
</application>
```

11. Создайте файл приложения UpdatableApplication_v2.mxml и скопируйте в него код из UpdatableApplication.mxml. Конечно, обновления для приложений должны содержать какие-то отличия, но нас сейчас интересует только проверка работы функции обновления, а не добавление новых функций.
12. Создайте файл .air для UpdatableApplication_v2. Воспользуйтесь тем же сертификатом, с помощью которого вы создавали UpdatableApplication, потому что у новой версии должен быть тот же самый ID издателя.
13. Загрузите файл .air на свой веб-сервер по адресу, указанному в latestversion.txt.
14. Запустите UpdatableApplication версии 1.0, которая уже стоит у вас на машине, и когда появится предложение обновить версию, щелкните по кнопке Update. Вы увидите, как загрузится файл, а затем выполнится обновление и запустится новая версия.

Мы рассмотрели процедуру обеспечения функции активного обновления в приложении AIR. Даже если пользователь запустит обновление не из самого приложения (например, с помощью значка на веб-странице), вы все равно сможете управлять обновлением из текущей версии, установленной на вашей машине, о чем будет сказано в следующем разделе.

8.3. Запуск приложений AIR

Может показаться, что запуск приложений AIR – слишком элементарный вопрос для обсуждения. Неужели для такой книги не нашлось более серьезных тем? Что еще можно сообщить о запуске приложения AIR, кроме того, что нужно сделать двойной щелчок по его значку, выбрать опцию меню или щелкнуть по значку на веб-странице? Но нас интересует другой вопрос: как сделать, чтобы приложение AIR по-разному реагировало в зависимости от способа, которым оно было запущено. Например, если пользователь запускает приложение двойным щелчком по файлу ассоциированного типа, может потребоваться, чтобы приложение AIR автоматически открывало этот файл или, в другом случае, читало его содержимое. Мы разберемся в том, как приложение AIR может выяснить способ, которым оно было запущено.

8.3.1. Обработка события invoke

При запуске приложения AIR объект NativeApplication (и WindowedApplication) посылает событие invoke. Когда пользователь запускает приложение двойным щелчком по его значку, событие invoke содержит мало информации. Но в других случаях в событии invoke есть дополнительные сведения, которыми может воспользоваться приложение AIR. А именно, если приложение AIR запускается двойным щелчком по файлу ассоциированного типа, в событии invoke есть данные о файле, по которому щелкнул пользователь: объект File, указывающий на этот файл. Можно воспользоваться этой информацией каким-то разумным образом. Например, если пользователь запускает приложение двойным щелчком по файлу, это может означать необходимость считать содержимое этого файла в приложение при его запуске.

Событие invoke имеет тип flash.events.InvokeEvent, и дополнительная информация, если она есть, хранится в свойстве arguments этого объекта. Свойство arguments представляет собой массив значений. Если пользователь запустил приложение щелчком по файлу, путь к файлу будет храниться в объекте File как элемент свойства arguments соответствующего события invoke. В следующем разделе мы увидим, как воспользоваться событием invoke, чтобы открыть файл в приложении AirTube.

8.3.2. Запуск AirTube через ассоциированный файл

Если помните, в главе 4 мы ввели в AirTube функцию перетаскивания, с помощью которой пользователь мог перетащить видео в файловую систему (например, на рабочий стол) и сохранить файл .atv, являющийся индивидуальным форматом, созданным нами для хранения ID фильма. Теперь мы хотим, чтобы пользователь двойным щелчком по файлу .atv запускал AirTube и открывал видеофайл.

1. Откройте файл дескриптора для AirTube и приведите его в соответствие с листингом 8.8. Обратите внимание: мы создаем ассоциирование по типу файла для файлов с расширением .flv.

Листинг 8.8. Задание ассоциированных типов файлов для AirTube

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://ns.adobe.com/air/application/1.0">
   <id>AirTube</id>
   <filename>AirTube</filename>
   <name>AirTube</name>
   <version>1.0</version>
   <initialWindow>
      <content>AirTube.swf</content>
   </initialWindow>
   <fileTypes>
       fileType>
         <name>AirTubeVideo</name>
         <extension>atv</extension>
```

```
        </fileType>
    </fileTypes>
</application>
```

2. Измените AirTubeService.as, добавив открытый метод `getVideoById()`. Этот метод ищет фильмы в автономном режиме по ID. Этот метод показан в листинге 8.9, и его нужно добавить в класс AirTubeService.

Листинг 8.9. Добавление метода getVideoById( ) в AirTubeService

```
public function getVideoById(id:String):void {
    var sql:SQLStatement = new SQLStatement();
    sql.addEventListener(SQLEvent.RESULT, getOfflineVideosResultHandler);
    sql.sqlConnection = _connection;
    sql.itemClass = Video;
    sql.text = "SELECT * FROM videos WHERE id = @id";
    sql.parameters["@id"] = id;
    sql.execute();
}
```

3. Измените код AirTube.mxml. Сначала модифицируем метод `creationCompleteHandler()`, добавив в него код, который установит AirTube в качестве приложения по умолчанию для файлов .atv. Новый код приведен в листинге 8.10.

Листинг 8.10. Регистрация AirTube в качестве приложения по умолчанию для файлов .atv

```
private function creationCompleteHandler():void {
    _service = AirTubeService.getInstance();
    _service.key = "AhWz9YtBmWM";
    _videoWindow = new VideoWindow();
    _htmlWindow = new HTMLWindow();
    _instance = this;
    registerClassAlias("com.manning.airtube.data.AirTubeVideo",
    ➥AirTubeVideo);
    if(!NativeApplication.nativeApplication.
    ➥isSetAsDefaultApplication("atv")) {
        NativeApplication.nativeApplication.
      ➥setAsDefaultApplication("atv");
    }
}
```

4. Добавьте атрибут `invoke` к тегу `WindowedApplication`, что вынудит приложение AIR вызывать метод `invokeHandler()` при возникновении события `invoke`:

```
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    creationComplete="creationCompleteHandler();" width="800"
height="600"
    closing="closingHandler();" invoke="invokeHandler(event);">
```

5. Определите `invokeHandler()`. Этот метод принимает параметр `InvokeEvent`. Определяем, есть ли в свойстве `arguments` какие-нибудь элементы. Если есть, то проверяем, является ли первый элемент путем к файлу с расширением .atv. Если да, читаем этот файл и вызываем `getVideoById()`, чтобы извлечь фильм и отобразить в результирующем списке. Это делается в предположении, что файл сохранен локально. Листинг 8.11 показывает новый метод `invokeHandler()`, который мы добавляем в AirTube.mxml.

Листинг 8.11. Метод invokeHandler( ) в AirTube.mxml

```
private function invokeHandler(event:InvokeEvent):void {
   if(event.arguments.length > 0) {
      var file:File = new File(event.arguments[0]);
      var fileName:Array = file.name.split(".");
      if(fileName[1] != undefined) {
         if(fileName[1] == "atv") {
            var reader:FileStream = new FileStream();
            reader.open(file, FileMode.READ);
            reader.position = 0;
            var id:String = reader.readUTF();
            _service.getVideoById(id);
         }
      }
   }
}
```

6. Экспортируйте файл .air для AirTube, инсталлируйте его и запустите. Проверьте работу новых функций.

Теперь мы умеем обрабатывать событие `invoke` и даже воспользовались этим в AirTube. Далее рассмотрим, как обрабатывать аналогичные события `invoke`, исходящие от броузера.

8.3.3. Перехват событий броузера

Как мы выяснили, когда запускается приложение – непосредственно или двойным щелчком по файлу ассоциированного типа, – оно посылает событие `invoke`. Аналогично, при запуске приложения из броузера оно посылает событие `browserInvoke`. Событие `invoke` имеет тип `InvokeEvent`, а событие `browserInvoke` имеет тип `flash.events.BrowserInvokeEvent`. Такие события, как и событие `invoke`, содержат свойство `arguments`, являющееся массивом параметров, переданных приложению. Для событий `browserInvoke` параметрами являются любые значения, переданные в третьем параметре метода `launchApplication()`.

В отличие от событий `invoke`, зарегистрировать обработчик события `browserInvoke` непосредственно из `WindowedApplication` нельзя. Регистрировать обработчик этого события нужно из экземпляра `NativeApplication`, независимо от того, как вы строите приложение – с помощью Flash или Flex. Ниже показано, как зарегистрировать обработчик события `browserInvoke`:

```
NativeApplication.nativeApplication.addEventListener(
➥BrowserInvokeEvent.BROWSER_INVOKE, browserInvokeHandler);
```

Вероятно, проще всего разобраться с событием `browserInvoke` на конкретном примере. Мы сейчас построим простое приложение, которое демонстрирует работу этого события:

1. Создайте новый проект AIR с именем BrowserInvoke.
2. Создайте главный файл MXML приложения с именем BrowserInvoke.mxml и поместите в него код листинга 8.12. Этот код сначала показывает ID приложения и ID издателя. Получив событие `browserInvoke`, он показывает все параметры, которые были ему переданы.

Листинг 8.12. Главный файл приложения показывает переданные ему параметры

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
   layout="absolute" creationComplete="creationCompleteHandler()">
   <mx:Script>
      <![CDATA[

         private function creationCompleteHandler():void {
            textArea.text = "application ID: " +
            ➥NativeApplication.nativeApplication.applicationID;
            textArea.text += "\npublisher ID: " +
            ➥NativeApplication.nativeApplication.publisherID;
            NativeApplication.nativeApplication.addEventListener(
            ➥BrowserInvokeEvent.BROWSER_INVOKE, browserInvokeHandler);
         }

         private function browserInvokeHandler(
        ➥event:BrowserInvokeEvent):void    {
            textArea.text += "\n* arguments: " +
            ➥event.arguments.length;
            for(var i:Number = 0; i < event.arguments.length; i++) {
               textArea.text += "\n\t" + event.arguments[i];
            }
         }

      ]]>
   </mx:Script>
   <mx:TextArea width="100%" height="100%" id="textArea" />
</mx:WindowedApplication>
```

3. Создайте файл дескриптора согласно листингу 8.13. Обратите внимание на значение `true`, заданное для `allowBrowserInvocation`.

Листинг 8.13. Задайте для allowBrowserInvocation значение true в файле дескриптора

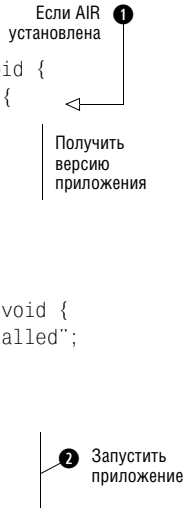
```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://ns.adobe.com/air/application/1.0">
   <id>com.manning.airinaction.BrowserInvoke</id>
   <filename>BrowserInvoke</filename>
```

```
    <name>Initialize</name>
    <version>1.0</version>
    <initialWindow>
       <content>BrowserInvoke.swf</content>
    </initialWindow>
    <allowBrowserInvocation>true</allowBrowserInvocation>
</application>
```

4. Экспортируйте файл .air и установите его.
5. Запустите приложение и скопируйте ID издателя. Он понадобится вам для запуска приложения из броузера.
6. Создайте новый веб-проект с именем LaunchFromBrowser.
7. Создайте главный документ MXML приложения с именем LaunchFromBrowser.mxml и поместите в него код из листинга 8.14. В этом коде используется `AirBadgeService` из листинга 8.1.

Листинг 8.14. Веб-приложение для запуска приложения AIR

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
creationComplete="creationCompleteHandler()">
   <mx:Script>
      <![CDATA[
         import com.manning.airinaction.utilities.AirBadgeService;

         private var _badgeService:AirBadgeService;

         private function creationCompleteHandler():void {
            _badgeService = new AirBadgeService();
            _badgeService.addEventListener(Event.COMPLETE,
                                 completeHandler);
         }

         private function completeHandler(event:Event):void {
            if(_badgeService.getStatus() == "installed") {
               _badgeService.getApplicationVersion(
               ➥"com.manning.airinaction.BrowserInvoke",
               ➥" YourPublisherID ", versionHandler);
               launchButton.enabled = true;
            }
         }

         private function versionHandler(version:String):void {
            textArea.text = "version " + version + " installed";
         }

         private function launchApplication():void {
            _badgeService.launchApplication(
            ➥"com.manning.airinaction.BrowserInvoke",
            ➥"YourPublisherID", ["a", "b", "c", "d"]);
         }

      ]]>
   </mx:Script>
```



```
    <mx:TextArea id="textArea" />
    <mx:Button id="launchButton" label="Launch"
       click="launchApplication();" enabled="false" />
</mx:Application>
```

В этом коде мы проверяем, установлена ли среда AIR ❶, и если да, то активируем кнопку запуска приложения. (Мы также дополнительно показываем номер установленной версии.) Когда пользователь щелкает по кнопке, мы вызываем `launchApplication()` ❷ и передаем приложению четыре параметра: a, b, c и d. Обратите внимание, что в двух местах вы должны подставить свой `YourPublisherID`. Воспользуйтесь тем ID издателя, который вы скопировали при запуске `BrowserInvoke`.

8. Закройте приложение BrowserInvoke, если оно еще работает. Запустите LaunchFromBrowser в броузере и щелкните по кнопке Launch. Вы увидите, как запустится BrowserInvoke и отобразит четыре параметра, которые вы передали ему из броузера.

Теперь вам известны не только стандартные способы запуска приложения двойным щелчком по его окну или выбором пункта меню, но еще два дополнительных способа. Вы теперь умеете запускать приложение с помощью файлов ассоциированного типа, а также из броузера, передавая ему параметры. С такими знаниями вы становитесь специалистом по запуску приложений AIR, и мы готовы завершить свою книгу.

8.4. Резюме

В этой главе мы научились устанавливать, обновлять и запускать приложения AIR. Вы теперь знаете, что значки (badges) – это размещенные в сети файлы .swf, с помощью которых пользователи могут проводить гладкую установку приложений AIR, и умеете создавать значки несколькими способами. Затем вы научились применять класс `Updater` для обновления приложений. Кроме того, вы знаете, как обрабатывать события вызова (`invoke`), генерируемые при двойном щелчке по файлам ассоциированных типов или щелчке по значку запуска в сети. Ряд этих идей был реализован в нескольких примерах, включая приложение AirTube.

На этом завершается наше изучение AIR. Поверьте, мы действительно достигли конца книги. Однако, как вы наверняка знаете по своему опыту, конец одного – это, в действительности, начало другого. В данном случае, это конец вашего ознакомления с Adobe AIR, но это лишь начало открывающихся перед вами возможностей создания приложений. Возвращайтесь вновь к этой книге, если вам потребуется освежить какую-то тему, но не ограничивайтесь тем, что в ней представлено. Мы показали вам лишь верхушку айсберга. В отношении AIR ваши возможности ограничены только вашим воображением. Желаем вам успехов во всех ваших начинаниях.

Алфавитный указатель

В

Г

Д

Ж

З

И

К

Л

М

О

П

Р

С

Т

У

Ф

Х

Ч

Ш