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

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.

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.

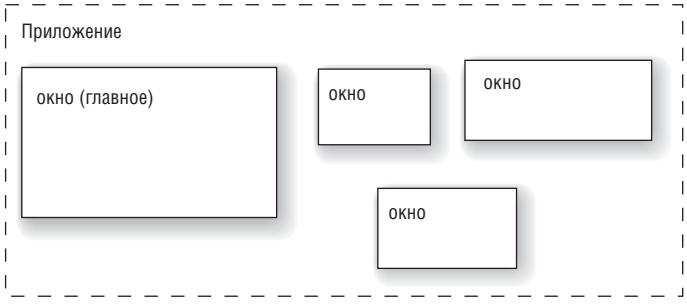

***Рис. 2.1.** В каждом приложении AIR есть один объект application и один или несколько объектов window*

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

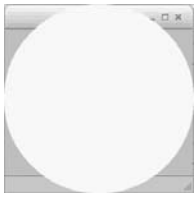

***Рис. 2.6.** Окно Flex с установленным параметром systemChrome*

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.

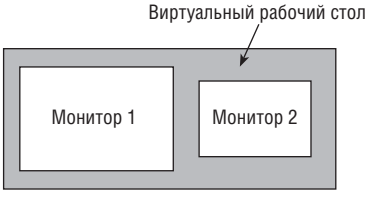

Рис. 2.7. *Приложения AIR рассматривают все мониторы как один виртуальный рабочий стол*

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
         }
      }

   }

}
```