

*Программирование на Java
для нового поколения мобильных устройств*

2-Е ИЗДАНИЕ



Программирование под

Android

Зигард Медникс

Лайрд Дорнин

Блэйк Мик

Масуми Накамура

O'REILLY®

 ПИТЕР®

Zigurd Mednieks, Laird Dornin
G. Blake Meike, and Masumi Nakamura

Programming
Android

SECOND EDITION

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Зигард Медникс, Лайрд Дорнин
Блэйк Мик, Масуми Накамура

Программирование под
Android

ВТОРОЕ ИЗДАНИЕ



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

2013

Медникс З., Дорнин Л., Мик Б., Накамура М.

Программирование под Android

2-е издание

Серия «Бестселлеры O'Reilly»

Перевел с английского *О. Сивченко*

Заведующий редакцией
Ведущий редактор
Художник
Корректор
Верстка

*Д. Винуцкий
Е. Каляева
Л. Адуевская
Е. Павлович
М. Моисеева*

ББК 32.973.2-018.2

УДК 004.451

Медникс З., Дорнин Л., Мик Б., Накамура М.

П78 Программирование под Android. 2-е изд. — СПб.: Питер, 2013. — 560 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-00526-5

В этой книге подробно рассказано о последних наработках в области программирования для Android. Тщательно изучив ее, вы научитесь создавать современные пользовательские интерфейсы как для мобильных телефонов, так и для планшетов. Книга рассказывает об инструментарии Android и важнейших практиках программирования для этой системы, в частности рассматривает оптимальные способы использования API для Android 4.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1449316648 англ.

Copyright © 2012 Zigurd Mednieks, Laird Dornin, Blake Meike, and Masumi Nakamura. All rights reserved. Authorized Russian translation of the English edition of Programming Android, 2nd Edition (ISBN 9781449316648). This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-496-00526-5

© Перевод на русский язык ООО Издательство «Питер», 2013
© Издание на русском языке, оформление ООО Издательство «Питер», 2013

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 18.03.13. Формат 70×100/16. Усл. п. л. 45,150. Тираж 2000. Заказ 0000.

Отпечатано с готовых диапозитивов в ГППО «Псковская областная типография».

180004, Псков, ул. Ротная, 34.

Краткое содержание

Предисловие	11
От издательства	16

Часть I. Инструментарий и основы разработки

Глава 1. Установка Android SDK и необходимые предпосылки	19
Глава 2. Java для Android	49
Глава 3. Составные части приложения Android	94
Глава 4. Передача программы пользователю	151
Глава 5. Среда Eclipse для разработки программ Android	171

Часть II. Фреймворк Android

Глава 6. Создание вида	193
Глава 7. Фрагменты и многоплатформенная поддержка	229
Глава 8. Рисование двухмерной и трехмерной графики	249
Глава 9. Обращение с данными и их долговременное хранение	289

Часть III. Скелет приложения Android

Глава 10. Каркас работоспособного приложения	323
Глава 11. Создание пользовательского интерфейса	345
Глава 12. Использование поставщиков содержимого	374
Глава 13. Поставщики содержимого как фасад для веб-сервисов RESTful	400

Часть IV. Продвинутые темы

Глава 14. Поиск	433
Глава 15. Геолокация и картография	449
Глава 16. Мультимедиа	471
Глава 17. Сенсоры, коммуникация ближнего поля, речь, жесты и доступность	483
Глава 18. Коммуникация, личные данные, синхронизация и социальные сети	505
Глава 19. Комплект для нативной разработки в Android (NDK)	542

Оглавление

Предисловие	11
Как построена эта книга	11
Условные сокращения, используемые в данной книге.	12
Работа с примерами кода.	13
Как с нами связаться	13
Благодарности.	13
Об авторах	15
От издательства.	16

Часть I. Инструментарий и основы разработки

Глава 1. Установка Android SDK и необходимые предпосылки . . .	19
Установка комплекта разработки ПО (SDK) Android и необходимые условия.	19
Проверка работоспособности.	28
Компоненты комплекта для разработки ПО	37
Обеспечение актуальности	44
Примеры кода	47
О чтении кода	48
Глава 2. Java для Android	49
Android и видоизменение клиентской разновидности Java	49
Система типов Java	50
Область видимости	73
Идиомы программирования в Java	77
Глава 3. Составные части приложения Android	94
Сравнение Android и традиционных моделей программирования	94
Активности, намерения и задачи	95
Другие компоненты Android	98
Жизненные циклы компонентов.	103

Статические ресурсы приложения и его контекст	106
Среда времени исполнения приложения Android	115
Шаблон приложения Android	119
Параллелизм в Android	126
Сериализация	142
Глава 4. Передача программы пользователю	151
Подписывание приложения	151
Размещение программы на Android Market для распространения	161
Альтернативные способы распространения	163
Ключи к интерфейсу программирования приложений (API) для работы с картами Google.	167
Обеспечение совместимости на уровне интерфейса программирования приложений	169
Совместимость с экранами нескольких разновидностей	169
Глава 5. Среда Eclipse для разработки программ Android.	171
Концепции и терминология Eclipse	172
Виды и перспективы Eclipse	177
Написание кода Java в Eclipse	181
Eclipse и Android.	182
Предотвращение ошибок и поддержание чистоты кода	183
Характерные особенности Eclipse и альтернативные инструменты	189

Часть II. Фреймворк Android

Глава 6. Создание вида	193
Архитектура графического пользовательского интерфейса в Android.	193
Сборка графического интерфейса	198
Подключение контроллера.	203
Меню и панель действий	223
Отладка и оптимизация видов	226
Глава 7. Фрагменты и многоплатформенная поддержка	229
Создание фрагмента	230
Жизненный цикл фрагмента	233
Менеджер фрагментов.	234
Транзакции фрагмента	236
Пакет поддержки.	240
Фрагменты и макет	241

Глава 8. Рисование двухмерной и трехмерной графики	249
Создание собственных виджетов	249
Украшения.	274
Глава 9. Обращение с данными и их долговременное	
хранение	289
Обзор реляционной базы данных.	289
SQLite	290
Язык SQL.	291
SQL и модель построения архитектуры вокруг базы данных	
в приложениях Android	302
Классы базы данных в Android	303
Разработка базы данных для приложений Android	304
API базы данных на примере MJAndroid	308

Часть III. Скелет приложения Android

Глава 10. Каркас работоспособного приложения.	323
Визуализация жизненных циклов.	324
Визуализация жизненного цикла фрагмента.	337
Методы жизненного цикла класса Application	341
Глава 11. Создание пользовательского интерфейса	345
Общий дизайн интерфейса	346
Визуальное редактирование пользовательских интерфейсов	349
Начнем с чистого листа	349
Сворачивание и разворачивание масштабируемого пользовательского	
интерфейса.	357
Делегирование задач классам фрагментов.	362
Обеспечение совместной работы активности, фрагмента, панели	
действий и нескольких макетов.	365
Другая активность.	369
Глава 12. Использование поставщиков содержимого	374
Понятие о поставщиках содержимого.	376
Определение общедоступного API поставщика содержимого	379
Написание и интеграция поставщика содержимого	384
Управление файлами и двоичные данные	386
Модель MVC в Android и наблюдение за содержимым.	388
Полный код поставщика содержимого: поставщик	
SimpleFinchVideoContentProvider.	390
Объявление вашего поставщика содержимого	399

Глава 13. Поставщики содержимого как фасад	
для веб-сервисов RESTful	400
Разработка приложений Android с передачей состояния	
представления (RESTful)	402
Сетевой вариант «Модель-вид-контроллер»	402
Общая характеристика достоинств	404
Пример кода: динамическое построение списка и кэширование	
видеокартин YouTube	406
Структура исходного кода для примера с Finch-видео при работе	
с YouTube	408
Пошаговая разработка поискового приложения	409
Этап 1. Пользовательский интерфейс собирает пользовательский	
ввод	409
Этап 2. Контроллер прослушивает события	410
Этап 3. Контроллер запрашивает данные у поставщика	
содержимого/модели при помощи метода <code>managedQuery</code>	410
Этап 4. Реализация запроса с передачей состояния представления	410

Часть IV. Продвинутые темы

Глава 14. Поиск	433
Поисковый интерфейс	433
Варианты завершения запроса	443
Глава 15. Геолокация и картография	449
Геолокационные сервисы	450
Работа с картами	451
Активность для работы с картами Google	451
MapView и MapActivity	452
Работа с MapView	453
Инициализация MapView и MapLocationOverlay	453
Приостановление и возобновление работы MapActivity	457
Управление картой при помощи клавиш меню	458
Управление картой с клавиатуры	460
Геолокация без использования карт	461
StreetView	469
Глава 16. Мультимедиа	471
Аудио и видео	471
Воспроизведение аудио и видео	472
Запись аудио и видео	476
Сохраненный медийный контент	482

Глава 17. Сенсоры, коммуникация ближнего поля, речь, жесты и доступность	483
Сенсоры	483
Коммуникация ближнего поля (NFC)	488
Ввод жестов	501
Доступность	503
Глава 18. Коммуникация, личные данные, синхронизация и социальные сети	505
Контакты учетной записи	505
Аутентификация и синхронизация	508
Bluetooth	525
Глава 19. Комплект для нативной разработки в Android (NDK)	542
Нативные методы и вызовы нативного интерфейса Java (JNI)	542
Комплект для нативной разработки в Android (Android NDK)	544
Нативные библиотеки и заголовки, предоставляемые в NDK	548
Создание собственных пользовательских библиотечных модулей	550
Нативные активности	554

Предисловие

Цель этой книги — помочь вам создавать качественно организованные приложения Android, возможности которых намного превышают функционал маленьких приложений-примеров.

Данное издание рассчитано на специалистов, приступающих к программированию на Android и имеющих при этом самый разный опыт. Если вы разрабатывали приложения на языке Objective-C для iPhone или Mac OS, то вам будет полезно познакомиться с инструментарием Android и функциями языка Java, предназначенными для программирования под Android. Прочитав об этом в данной книге, вы сможете адаптировать свои знания в области мобильной разработки к работе с Android. Если вы опытный Java-программист, то вам будет интересно описание архитектуры приложения Android, которое позволит вам применить ваш опыт разработки на Java в этом новом, актуальном направлении разработки клиентских приложений на Java. Можно сказать, что это издание предназначено для специалистов, имеющих значительный опыт разработки с применением объектно-ориентированных языков, создания мобильных приложений, приложений с передачей состояния представления (REST) и в смежных областях.

Как построена эта книга

Мы хотели, чтобы вы могли как можно быстрее приступить к работе. В главах, составивших первую часть книги, поэтапно объясняются способы работы с инструментарием, входящим в комплект разработки ПО (SDK). При изучении этого материала вы сможете пользоваться примерами кода из данной книги и из SDK, а также подробнее изучить инструментарий SDK, язык Java и архитектуру базы данных. Инструменты и базовые концепции, рассматриваемые в части I, могут быть достаточно хорошо вам знакомы, и, возможно, вы захотите сразу перейти к части II, в которой приведена основная информация, необходимая для разработки более крупных приложений Android.

Основным компонентом данной книги является пример приложения, которое использует веб-сервисы для передачи информации пользователю — подобный функционал имеется в ядре очень многих приложений. Мы опишем архитектуру приложения, а также новаторский подход к использованию классов

фреймворка Android, который поможет более эффективно решать задачи именно такого рода. Вы сможете применять это приложение как основу для создания собственных программ, а также как наглядное пособие для обучения программированию в Android.

Кроме того, мы исследуем интерфейсы программирования приложений (API) в конкретных прикладных областях: мультимедиа, определение местоположения, сенсоры, обмен информацией. Все это должно помочь вам приступить к созданию приложений именно в той области, которая вас особенно интересует.

Мы хотели бы, чтобы после прочтения этой книги вы обладали глубокими знаниями, которые не ограничиваются сведениями из справочного материала и несколькими примерами. Мы надеемся, что у вас появится собственная точка зрения по вопросу: «Как разрабатывать отличные приложения Android?»

Условные сокращения, используемые в данной книге

В данной книге применяются следующие условные обозначения.

Шрифт для названий

Используется для обозначения URL, адресов электронной почты, а также сочетаний клавиш и названий элементов интерфейса.

Шрифт для команд

Применяется для обозначения программных элементов — переменных и названий функций, типов данных, переменных окружения, операторов и ключевых слов и т. д.

Шрифт для листингов

Используется в листингах программного кода.

Полужирный шрифт для листингов

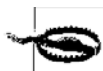
Указывает команды и другой текст, который должен воспроизводиться пользователем буквально.

Курсивный шрифт для листингов

Обозначает текст, который должен быть заменен значениями, сообщаемыми пользователем, или значениями, определяемыми в контексте.



Данный символ означает совет, замечание практического характера или общее замечание.



Этот символ означает предостережение.

Работа с примерами кода

Данная книга написана для того, чтобы помочь вам при работе. В принципе, вы можете использовать код, содержащийся в этой книге, в ваших программах и в документации. Можете не связываться с нами и не спрашивать разрешения, если собираетесь воспользоваться небольшим фрагментом кода. Например, если вы пишете программу и кое-где вставляете в нее код из данной книги, разрешения для этого не требуется. Однако, если вы запишете на диск примеры из книг издательства O'Reilly и начнете раздавать или продавать такие диски, на это необходимо получить разрешение. Если вы цитируете книгу, отвечая на вопрос, или воспроизводите код из нее в качестве примера, на это не требуется разрешения. Если вы включаете значительный фрагмент кода из книги в документацию по вашему продукту, на это требуется разрешение.

Если вам кажется, что вы пользуетесь кодом, указанным в книге, каким-то необычным способом, и хотите получить от нас разрешение на это, не стесняйтесь и пишите нам по адресу permissions@oreilly.com.

Как с нами связаться

Все вопросы и комментарии отправляйте издателю:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США или Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

У нас есть веб-страница, посвященная этой книге, где мы перечисляем ошибки, приводим примеры и дополнительную информацию. Вы можете посетить ее, перейдя по адресу http://oreil.ly/prog_android_2e.

Для получения более подробной информации о наших книгах, курсах, конференциях и новостях посетите веб-сайт <http://www.oreilly.com>.

Мы в сети Facebook: <http://facebook.com/oreilly>.

Мы в Twitter: <http://twitter.com/oreillymedia>.

Наш канал на YouTube: <http://www.youtube.com/oreillymedia>.

Благодарности

Авторы использовали в этой книге фрагменты из ее предыдущего издания — Android Application Development (издательства O'Reilly).

Черновики оригинала этой книги выкладывались в Интернете в системе OFPS (Open Feedback Publishing System, система открытой публикации для получения обратной связи) издательства O'Reilly. Так мы хотели проверить, насколько нам удастся достигать целей, поставленных при написании книги. Мы будем очень

благодарны всем пользователям OFPS за исправление наших ошибок и улучшение нашего стиля. Открытая публикация черновиков сохранится и при подготовке следующих изданий. Нам интересны любые ваши мнения о тех или иных сторонах книги.

Зигурд Медникс

Я бесконечно благодарен Терри, моей жене, а также Майе и Чарльзу, моим детям, которые помогли мне выкроить время на написание данного издания. Эта книга обязана своим появлением нашему агенту, Кэрол Джелен из Waterside Productions, которая смогла правильно оформить предложенный нами материал, а также Майку Хендриксону, запустившему этот проект в издательстве O'Reilly. Наши редакторы, Брайан Джепсон и Энди Орам, помогли всем авторам слаженно работать для достижения общих целей и результатов. Благодарю Йохана ван дер Хевена, который выполнил обзорные комментарии и помог сделать книгу значительно точнее и яснее.

Спасибо всем читателям, приславшим свои комментарии через систему OFPS, также поучаствовавшим в создании этой книги.

Лайрд Дорнин

Спасибо моей чудесной Норе за то, что сподвигла меня поучаствовать в этом проекте, несмотря на то что сначала даже представить было невозможно, сколько сил на это уйдет. Да здравствуют вылазки в Акадию и Нью-Гэмпшир и бессонные ночи, проведенные за работой. Я счастлив, что эта книга не застопорила наш самый важный проект — появление на свет нашей прекрасной дочурки Клэр. Спасибо нашему редактору Энди, а также моим соавторам, благодаря которым у меня появилась возможность принять участие в такой работе. Спасибо Ларри за критику и помощь в работе над этим проектом. Наконец, спасибо нашим основным рецензентам, Виджаю и Йохану, вы оба замечательно справились с улучшением всего текста.

Дж. Блэйк Мик

Благодарю нашего агента, Кэрол Джелен из Waterside Productions, без которой эта книга навсегда осталась бы просто хорошей идеей. Спасибо нашим редакторам Брайану Джепсону и Энди Ораму. Все читатели этой книги пользуются также плодами труда Йохана ван дер Хевена и Виджая Йеллапрагады, технических рецензентов; Сумиты Мухерджи, Адама Заремба и остальной команды издательства O'Reilly. Благодарю всех тех, кто воспользовался системой O'Reilly OFPS и просмотрел первые, совершенно неудобоваримые черновики, дал четкие комментарии и отследил самые вопиющие ошибки. Спасибо, ребята! И конечно, особой благодарности заслуживают Зигурд, Лайрд и Масуми — работать с ними было для меня честью и истинным удовольствием. Последнее, самое большое спасибо скажу моей жене Кэтрин, которая мобилизует меня и поддерживает в тяжелые моменты.

Масуми Накамура

Хотелось бы поблагодарить моих друзей и семью за их поддержку во время моей работы над этим и другими проектами. Особенно благодарю Джессамин за то, что она рядом со мной уже столько лет. Хотелось бы поблагодарить Брайана и Энди за то, что поддерживали нас на самых сложных этапах написания и публикации книги, а также моих соавторов за то, что я смог поучаствовать вместе с ними в этой работе. Кроме того, кратко благодарю весь коллектив WHERE.Inc., оказавший мне посильную поддержку в моих технологических странствиях. Наконец, спасибо вам, дорогие читатели, а также всем разработчикам, которые неустанно трудятся, чтобы Android оставалась великолепной платформой для работы и к тому же очень приятной в использовании.

Об авторах

Зигард Медникс — консультант, работающий с ведущими производителями комплектного оборудования, корпорациями и другими предприятиями, работающими в области новых технологий и занимающимися созданием систем на базе Android.

Лайрд Дорнин — старший инженер в фирме — крупном мобильном операторе. Имеет богатейший опыт работы с Java, Android, J2ME, SavaJe и WebKit.

Дж. Блэйк Мик — разработчик-ветеран с обширным опытом написания приложений Java для различных мобильных и серверных платформ.

Масуми Накамура — главный архитектор в группе по работе с большими данными и рекомендациям, работает в компании WHERE.Inc.

От издательства

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

Мы будем рады узнать ваше мнение!

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

ЧАСТЬ I

Инструментарий и основы разработки

- Глава 1.** Установка Android SDK и необходимые предпосылки
- Глава 2.** Java для Android
- Глава 3.** Составные части приложения Android
- Глава 4.** Передача программы пользователю
- Глава 5.** Среда Eclipse для разработки программ Android

В части I рассказано, как установить и использовать необходимые инструменты, что следует знать о языке Java, чтобы писать качественный код для Android, как создавать и использовать базы данных SQL, являющиеся центральным компонентом модели приложения в Android. Здесь описаны система долговременного хранения информации и основные паттерны разработки, применимые к программам Android.

1 Установка Android SDK и необходимые предпосылки

В этой главе рассказано, как установить комплект для разработки ПО (SDK) для платформы Android, а также все остальные программы, которые вам могут понадобиться при работе. В конце главы вы сможете запустить в эмуляторе простую программу Hello, World!. Разработка приложений для Android может происходить в операционных системах Windows, Mac OS X и Linux. Мы скачаем программы, рассмотрим, каковы функции отдельных инструментов, входящих в SDK, а также покажем вам образцы исходного кода.

На протяжении всей книги, и особенно в главе 1, мы будем ссылаться на размещенные на различных сайтах инструкции по установке и обновлению тех инструментов, которыми вы будете пользоваться при написании программ для Android. Самый важный ресурс, на котором следует искать информацию и ссылки на инструменты, — это сайт разработчиков Android: <http://developer.android.com>.

Эта глава посвящена в основном процессу установки и объясняет, как сочетаются и взаимодействуют компоненты системы Android и инструменты для их разработки. Здесь также рассмотрены изменения, которые могут происходить в той или иной части системы.

Установка комплекта разработки ПО (SDK) Android и необходимые условия

Для успешной установки SDK Android требуется еще два комплекта программ, не входящих в его состав: комплект для разработки на языке Java (JDK) и интегрированная среда разработки (IDE) Eclipse. Две эти системы не входят в комплект для разработки ПО в системе Android, потому что с их помощью создаются программы не только для Android, а также потому, что они могут уже быть установлены в вашей системе, а при дополнительной установке данных систем могут возникнуть конфликты версий.

Android SDK совместим с рядом последних версий JDK и интегрированной среды разработки Eclipse. Как правило, следует устанавливать последнюю версию каждого из этих инструментов. Подробные спецификации изложены на странице

System Requirements (Системные требования) на сайте разработчиков Android <http://developer.android.com/sdk/requirements.html>. При разработке программ для системы Android можно использовать и другие среды, кроме Eclipse. Информация о применении других интегрированных сред разработки содержится в документации Android по адресу <http://developer.android.com/guide/developing/other-ide.html>. В этой книге мы выбрали в качестве среды разработки именно Eclipse, так как в Eclipse поддерживается максимальное количество инструментов из состава Android SDK, а также работают разнообразные плагины (подключаемые модули). Кроме того, Eclipse — наиболее распространенная интегрированная среда разработки, используемая при работе с Java. В качестве альтернативы можно назвать IntelliJ IDEA, которую предпочитают многие специалисты по разработке на Java.

Комплект для разработки ПО на Java (JDK)

Если в вашей системе установлена актуальная версия комплекта JDK, не нужно ее переустанавливать. В JDK есть инструменты, в частности компилятор Java, применяемые в интегрированных средах и инструментариях для разработки программ на Java. В JDK также содержится среда времени исполнения Java (JRE), обеспечивающая работу программ Java, например Eclipse, в вашей системе.

Если вы работаете на Macintosh в одной из версий Mac OS X, поддерживающей комплект для разработки ПО в Android, то JDK у вас уже установлен.

Если вы работаете с Linux или Windows либо вам требуется установить JDK с сайта Oracle по какой-то другой причине, то вы можете найти этот файл по адресу <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

Комплект для установки в Windows, который вы скачаете, — это исполняемый файл. Запустите его, чтобы установить JDK.

Пользователям Linux потребуется извлечь каталог с JDK в свой домашний каталог и выполнить для установки JDK следующие шаги. При этом предполагается, что в качестве среды времени исполнения Java вы собираетесь использовать стандартный Oracle JDK.

Скачайте архив или пакет, соответствующий вашей системе. (Если это пакет, используйте для завершения установки менеджер пакетов. В противном случае выполните следующие шаги.)

```
tar -xvf archive-name.tar.gz
```

Архив с JDK будет извлечен в каталог `./jdk-name`. Теперь переместите каталог с JDK в `/usr/lib`:

```
sudo mv ./jdk-name /usr/lib/jvm/jdk-name
```

Переместив JDK в это место, вы создаете его конфигурируемую разновидность в вашей среде Linux. Это полезно, если у вас есть проекты или программы, требующие других версий JRE или JDK. Теперь запустите:

```
sudo update-alternatives --install "/usr/bin/java" "java" \  
    "/usr/lib/jvm/jdk-name/bin/java" 1  
sudo update-alternatives --install "/usr/bin/javac" "javac" \  
    "/usr/lib/jvm/jdk-name.0/bin/javac" 1
```

```
sudo update-alternatives --install "/usr/bin/javaws" "javaws" \
    "/usr/lib/jvm/jdk-name/bin/javaws" 1
sudo update-alternatives --config java
```

Вы увидите примерно такой вывод:

There are 3 choices for the alternative java (providing /usr/bin/java).

Selection	Path	Priority	Status
* 0	/usr/lib/jvm/java-6-openjdk/jre/bin/java	63	auto mode
1	/usr/lib/jvm/java-6-openjdk/jre/bin/java	63	manual mode
2	/usr/lib/jvm/java-6-sun/jre/bin/java	63	manual mode
3	/usr/lib/jvm/jdk1.7.0/jre/bin/java	1	manual mode

Press enter to keep the current choice[*], or type selection number:

Когда вы выберете устанавливаемый JDK, вы увидите следующий вывод:

```
update-alternatives: using /usr/lib/jvm/jdk1.7.0/jre/bin/java to provide
    /usr/bin/java (java) in manual mode.
```

Повторите приведенный выше процесс вывода для javac:

```
sudo update-alternatives --config javac
```

И для javaws:

```
sudo update-alternatives --config javaws
```

В зависимости от различных вариантов реализации Java, которые могут быть установлены в вашей системе, и от версии JDK, актуальной на момент чтения вами этой книги, номера версий могут отличаться от приведенных в примерах и выводе команд.

В любой операционной системе вы можете проверить установленную версию Java. Это делается при помощи следующей команды:

```
java -version
```

Выведенная данной командой версия должна соответствовать версии, которую вы установили. В противном случае повторите установку и убедитесь, что в процессе не возникло никаких ошибок.

Интегрированная среда разработки Eclipse

Eclipse — это универсальная платформа для работы с несколькими технологиями. Она находит разнообразное применение при создании интегрированных сред разработки для нескольких языков, а также при создании специализированных сред разработки для конкретных SDK.

Кроме того, она не сводится к поддержке инструментария для разработки программ и предоставляет, в частности, платформу для полнофункциональных клиентских приложений (RCP) в системе Lotus Notes, а также применяется в нескольких других контекстах.

Обычно Eclipse используется в качестве интегрированной среды разработки и обеспечивает написание, тестирование и отладку программ, особенно программ на Java. Кроме того, в системе присутствуют производные IDE (интегрированные среды разработки) и SDK (комплекты для разработки ПО) для различных вариантов разработки программ на Java, где Eclipse выступает в качестве основы. В данном случае берется широко распространенный вариант Eclipse и к нему подключается плагин, необходимый для разработки программ под Android. Нужно скачать пакет Eclipse, расположенный по адресу <http://www.eclipse.org/downloads>, и установить его.

На этой странице представлена подборка наиболее активно используемых пакетов Eclipse. В Eclipse пакетом называется комплект готовых модулей, благодаря которым Eclipse оптимизируется под разработку программ определенного рода. Как правило, работа с Eclipse начинается с установки одного из пакетов, доступных для загрузки на этой странице, после чего этот пакет дополняется плагинами. В вашем случае таким плагином будет ADT (инструментарий для разработки в Android). На странице System Requirements (Системные требования) на сайте разработчиков Android перечисляются три варианта пакета Eclipse, выступающих в качестве основы комплекта, необходимого для разработки приложений Android:

- Eclipse Classic (версия Eclipse 3.5 или выше);
- интегрированная среда разработки Eclipse для работы с Java;
- Eclipse для полнофункциональных клиентских приложений (RCP)/разработки плагинов.

Любой из этих вариантов будет работать, но, если вы не занимаетесь разработкой плагинов для Eclipse, целесообразно выбрать либо классический пакет, либо пакет для разработчиков на Java (EE — версия для предприятий или Standard — стандартная версия). Мы начинали работать с пакетом разработки Java EE. Именно в ней сделаны скриншоты, используемые в данной книге.

На сайте загрузки Eclipse автоматически определяется, какие именно версии подходят для вашей операционной системы — в частности, учитывается, является ли конкретная система 32- или 64-битной. Скачиваемый файл — это архив. Для установки среды Eclipse откройте архив и скопируйте каталог eclipse в ваш домашний каталог. Исполняемый файл для запуска Eclipse находится в данной папке.



Установка происходит именно так, как мы описали, то есть Eclipse устанавливается в ваш домашний каталог (или другом каталоге, который является «вашим собственным»). Это особенно актуально, если в вашей системе настроено несколько пользовательских аккаунтов (учетных записей). Не пользуйтесь менеджером пакетов системы. Ваш вариант Eclipse представляет собой лишь один из многих возможных комплектов плагинов Eclipse. Кроме того, установленная система Eclipse, скорее всего, потребует дополнительной пользовательской настройки. А управление плагинами Eclipse и их обновлениями происходит отдельно от управления другими программами вашей системы.

Если вы работаете с Ubuntu или с другим дистрибутивом Linux, не следует устанавливать Eclipse из репозитория вашего дистрибутива, а если среда уже установлена таким образом, программу нужно удалить и переустановить Eclipse

так, как было описано выше. Наличие пакета Eclipse в репозиториях Ubuntu является одной из черт, унаследованных этим дистрибутивом от Debian, на основе которого создана система Ubuntu. Такой метод установки и использования Eclipse не очень распространен, так как в большинстве случаев в таких репозиториях содержатся устаревшие версии Eclipse.

Чтобы убедиться, что Eclipse установлена правильно и что в вашей системе стоит версия JRE (среды времени исполнения Java), поддерживающая Eclipse, запустите исполняемый файл в каталоге Eclipse. Появится экран приглашения, показанный на рис. 1.1.

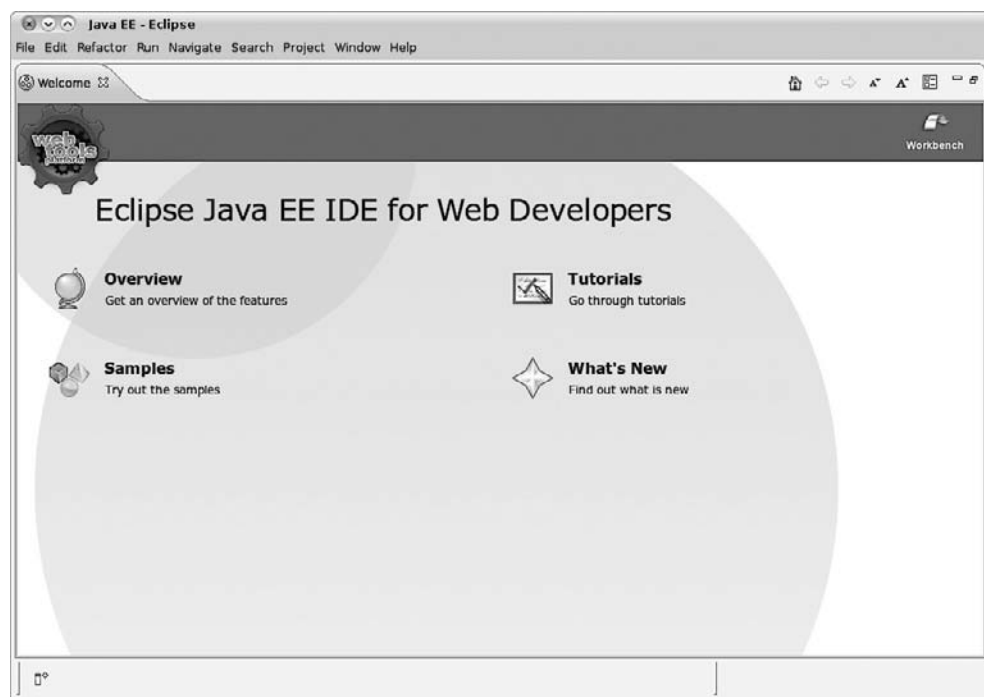


Рис. 1.1. Экран приглашения, который отображается при первом запуске Eclipse

Eclipse написана на Java, поэтому она требует наличия JRE. Пакет для разработки ПО на Java (JDK), установленный вами ранее, содержит JRE. Если Eclipse не запускается, проверьте, правильно ли установлен JDK.

Комплект разработки ПО для Android

Если у вас установлены JDK и Eclipse, в вашей системе соблюдены все условия для работы Android SDK и вы готовы к установке этого комплекта для разработки ПО. Android SDK — это коллекция файлов: в ее состав входят библиотеки, исполняемые файлы, скрипты, документация и т. д. Под установкой SDK понимается скачивание версии SDK, предназначенной для вашей платформы, и размещение файлов

SDK в одной из папок вашего домашнего каталога. Установочный сценарий отсутствует. Позже вы сконфигурируете плагин Eclipse так, чтобы он смог обнаружить, куда вы поместили SDK. Внешний вид, функционал и требования инструментария Android изменяются очень быстро. Описанный ниже процесс можно считать рекомендацией, которая не всегда будет соответствовать практике. Новейшую документацию по этим вопросам вы найдете по следующему адресу: <http://developer.android.com/tools/index.html>.

Для установки SDK скачайте с сайта <http://developer.android.com/sdk/index.html> пакет SDK, соответствующий вашей системе.

Скачанный файл — это архив. Откройте архив и извлеките содержащуюся в нем папку в домашний каталог.



Если вы работаете с 64-битной версией Linux, то вам, возможно, понадобится установить пакет `ia32-libs`. Чтобы проверить, нужен ли вам этот пакет, попробуйте запустить команду `adb` (`~/android-sdk-linux_*/platform-tools/adb`). Если система сообщает, что `adb` не удается найти (несмотря на то что команда находится прямо в директории `platform-tools`), это, вероятно, означает, что актуальная версия `adb`, а возможно, и другие инструменты не будут работать без установки пакета `ia32-libs`. Команда для установки пакета `ia32-libs` такова:

```
sudo apt-get install ia32-libs
```

В SDK содержится одна или две папки для инструментов: одна называется `tools`, а другая, присутствующая в версии SDK 8 и выше, — `platform-tools`. Эти каталоги должны быть указаны в пути к файлу, который представляет собой список каталогов, просматриваемых системой в поисках исполняемых файлов. Это происходит, когда вы запускаете исполняемый файл из командной строки. В системах Mac и Linux установка переменной окружения `PATH` производится в файле `.profile` (Ubuntu) или `.bash_profile` (Mac OS X) домашнего каталога. Добавьте в указанный файл строку кода, задающую переменную `PATH`, чтобы включить папку `tools` в SDK (разделительным знаком между записями служит двоеточие). Например, можно использовать следующую строку (но нужно заменить оба экземпляра `~/android-sdk-ARCH` полным путем к вашему экземпляру Android SDK):

```
export PATH=$PATH:~/android-sdk-ARCH/tools:~/android-sdk-ARCH/platform-tools
```

В системах Windows нажмите Пуск, далее правой кнопкой мыши щелкните на строке Компьютер и выберите в раскрывающемся меню пункт Свойства. После этого выберите Дополнительные параметры системы и нажмите кнопку Переменные среды. Дважды щелкните на системной переменной `path` и добавьте путь к каталогу в самом конце значения этой переменной (не меняйте никаких заданных настроек!). Кроме того, добавьте в конце два пути, разделив их точками с запятой, но не ставя между указанными путями пробелов. Например:

```
;C:\android-sdk-windows\tools;C:\android-sdk-windows\platform-tools
```

Отредактировав этот путь в Windows, Mac или Linux, закройте и откройте заново все запущенные экземпляры командной строки или терминалы, чтобы была принята новая настройка `PATH` (в Ubuntu может потребоваться выйти из системы и снова войти в нее, если терминал не настроен как интерактивная командная

оболочка с регистрацией — login shell, то есть не выполняет при запуске стартовые скрипты пользователя).

Добавление целевых платформ для сборки в SDK

Прежде чем приступить к написанию приложения для Android или даже перейти к созданию проекта, который попытается собрать приложение Android, нужно задать одну или несколько целевых платформ для сборки. Для этого используется SDK и менеджер виртуальных устройств Android (AVD). Данный инструмент позволяет устанавливать в SDK пакеты, которые будут поддерживать несколько версий операционной системы Android и несколько уровней API (интерфейсов программирования приложений).

После установки в Eclipse плагина ADT (об этом мы поговорим далее) пакет SDK и менеджер AVD можно запускать внутри Eclipse. Кроме того, запуск может происходить из командной строки — именно так мы будем поступать. Для активации SDK и AVD из командной строки используется команда `android`.

Скриншот на рис. 1.2 демонстрирует комплект для разработки ПО (SDK) в Android и менеджер виртуальных устройств Android (AVD) со всеми доступными версиями SDK, выбранными для установки.

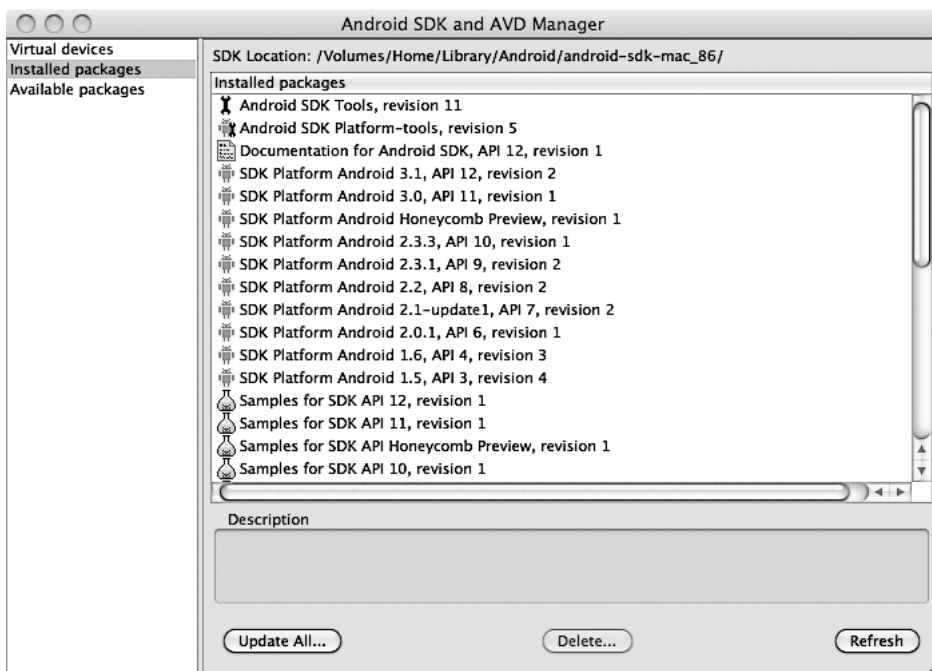


Рис. 1.2. SDK и менеджер виртуальных устройств Android, обеспечивающий установку различных уровней интерфейсов программирования приложений (API) для системы Android

Пакеты, обозначенные **SDK platform** (Платформа SDK), поддерживают создание приложений, совместимых с API (интерфейсами программирования приложений), которые используются в той или иной версии Android. Нужно установить как минимум самую новую версию (с наивысшим номером), но также не помешает установить все доступные уровни API и все дополнительные пакеты с API Google — ведь, возможно, в перспективе вы будете разрабатывать приложения и для других версий Android. Кроме того, нужно как минимум установить новейшие версии пакета с образцами приложений. Необходимо также установить пакет Platform-Tools из Android SDK.

Плагин для разработки в Android (ADT) для работы в среде Eclipse

Теперь, когда у вас установлены нужные файлы SDK, а также среда разработки Eclipse и комплект JDK, **требуется установить еще один важнейший инструмент**: плагин для разработки в Android (ADT). Плагин ADT добавляет в Eclipse функционал, специфичный для разработки под Android.

Программы из этого плагина позволяют создавать в Eclipse приложения для Android, запускать эмулятор Android, подключаться к службам отладки, входящим в состав эмулятора, редактировать XML-файлы Android, редактировать и компилировать файлы на языке определения интерфейса Android (AIDL), создавать пакеты приложений Android (файлы APK) и выполнять специфичные для Android задачи.

Использование мастера установки новых программ для скачивания и установки плагина ADT

Чтобы запустить мастер установки новых программ, нужно выполнить в Eclipse команду **Help ► Install New Software** (Помощь ► Установить новую программу) (рис. 1.3).

Для установки плагина ADT необходимо записать следующую ссылку: <https://dl-ssl.google.com/android/eclipse/> — в поле **Work With** (Использовать) и нажать **Enter** (рис. 1.4).



Процедура установки плагина ADT при помощи мастера новых программ подробнее описана на сайте разработчиков Android по адресу <http://developer.android.com/sdk/eclipse-adt.html#downloading>.

Документация Eclipse по этому мастеру доступна на сайте Eclipse по следующему адресу: <http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.platform.doc.user/tasks/tasks-124.htm>.

Добавив этот URL в список сайтов для получения новых плагинов, вы увидите запись **Developer Tools** (Инструменты разработчика) в списке **Available Software** (Доступные программы).

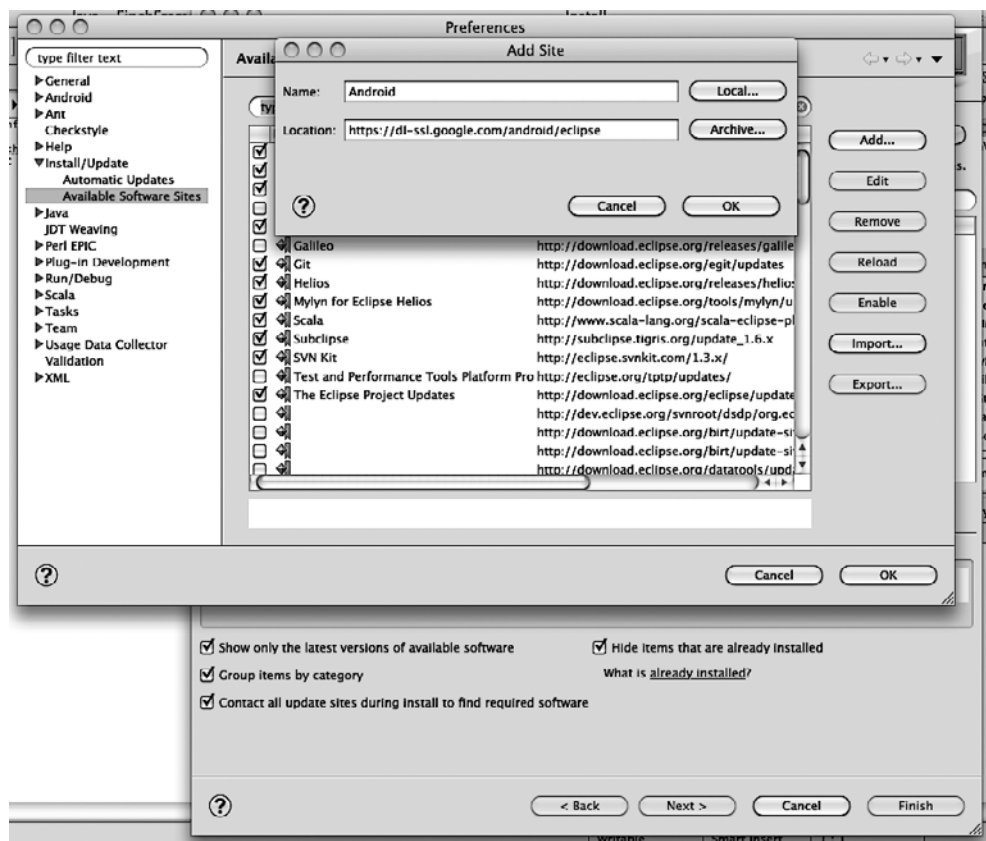


Рис. 1.3. Окно Add Site (Добавить сайт) в Eclipse

Выберите элемент **Developer Tools** (Инструменты разработчика), поставив флажок рядом с ним, потом нажмите кнопку **Next** (Далее). На следующем экране вам будет предложено принять условия лицензионного соглашения на эту программу. Приняв их, нажмите **Finish** (Готово), и плагин ADT будет установлен. Для завершения установки потребуется перезапустить Eclipse.

Конфигурирование плагина ADT

До завершения инсталляции остается еще один шаг. После того как вы установите плагин ADT, его еще нужно сконфигурировать. После установки плагина в различных частях Eclipse появятся новые диалоговые окна, специфичные для разработки программ в Android, новые команды меню и другие инструменты, в том числе диалоговое окно, в котором настраивается плагин ADT. Откройте диалоговое окно **Preferences** (Настройки), выполнив команду **Window** ► **Preferences** (Окно ► Настройки) в Windows и Linux или **Eclipse** ► **Preferences** (Eclipse ► Настройки) в Mac. Щелкните на элементе **Android** в левой области окна **Preferences** (Настройки).

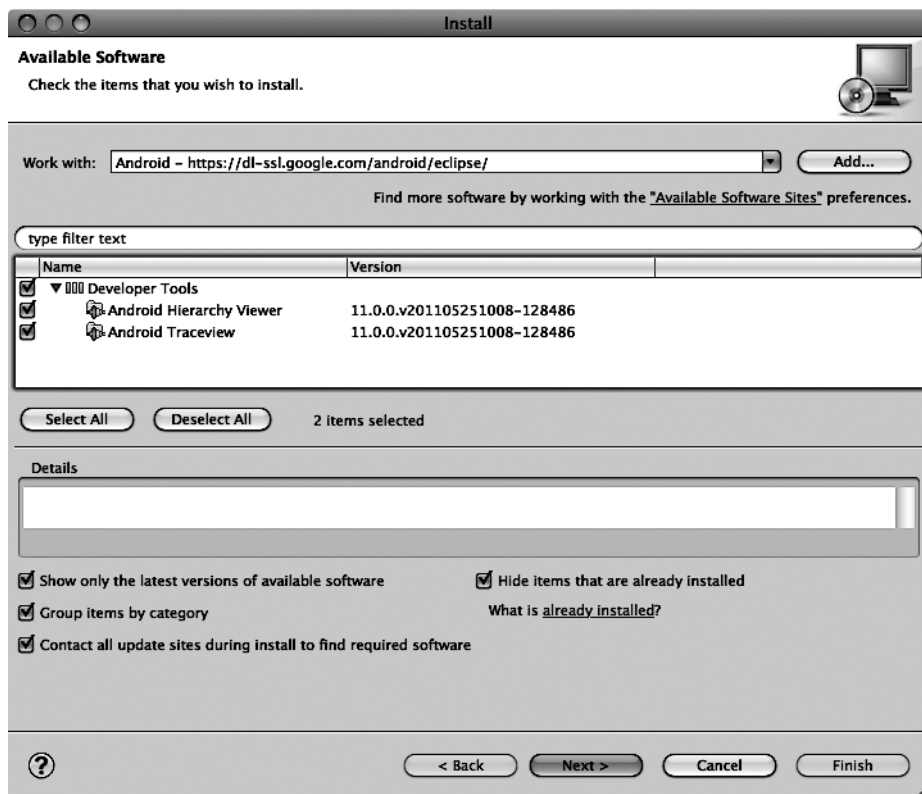


Рис. 1.4. Установочное окно Eclipse. Плагин для просмотра иерархии Android отображается как доступный



В первый раз, когда вы заходите в этот раздел настроек, там появляется еще одно диалоговое окно с вопросом, не хотите ли вы отправлять в Google определенную статистику использования программы. Сделайте выбор и нажмите Proceed (Продолжить).

На рис. 1.5 показано диалоговое окно с настройками Android. В верхней части этого окна расположено поле SDK location (Размещение SDK). В нем нужно указать путь к каталогу, в котором находится SDK, вручную либо при помощи диспетчера файлов. Нажмите Apply (Применить). Обратите внимание на то, что здесь же указаны выбранные вами целевые сборки, установка которых описана в подразделе «Добавление целевых платформ для сборки в SDK» выше.

Теперь установка комплекта для разработки ПО в Android завершена.

Проверка работоспособности

Если вы точно выполнили все этапы установки, описанные выше, и изучили соответствующие справочные материалы, установка Android уже должна быть завер-

шена. Чтобы убедиться, что все установленные компоненты работают, создадим простое приложение Android.

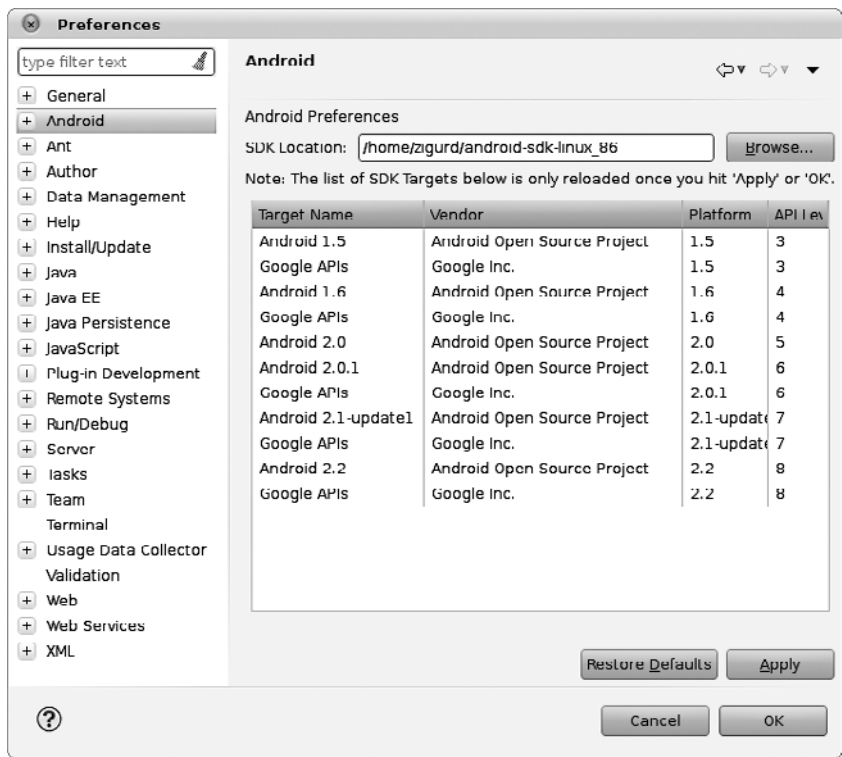


Рис. 1.5. Настройка расположения SDK в плагине Eclipse ADT при помощи диалогового окна конфигурации Android

Создание проекта Android

Первый этап при разработке Android-приложения заключается в создании проекта Android. Работа в Eclipse организована в виде проектов. Если вы выбираете проект Android, вы тем самым сообщаете Eclipse, что при работе над этим проектом будут использоваться плагин ADT и другие инструменты Android, связанные с данной задачей.



Справочная информация и подробные онлайн-инструкции о том, как создать проект Android, содержатся по адресу <http://developer.android.com/guide/developing/eclipse-adt.html>.

Чтобы начать новый проект, выберите команду меню **File** ► **New** ► **Android Project** (Файл ► Создать ► Проект Android). В диалоговом окне **New Project** (Новый проект) выберите в разделе **Android** вариант **Android Project** (Проект Android) и нажмите **Next** (Далее). Появится диалоговое окно **New Project** (Новый проект) (рис. 1.6).

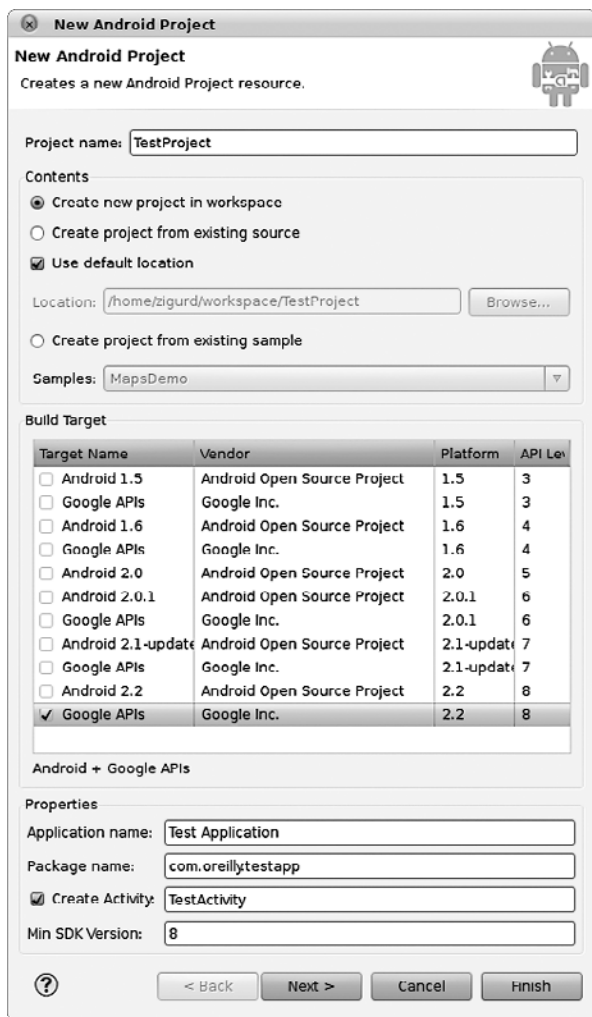


Рис. 1.6. Диалоговое окно для создания нового проекта Android

Для создания проекта Android необходимо указать следующую информацию.

- **Project name** (Имя проекта) — название проекта (но не приложения), которое появится в Eclipse. Введите TestProject, как показано на рис. 1.6.
- **Workspace** (Рабочее пространство) — каталог, содержащий набор проектов Eclipse. Новый проект можно создать в текущем рабочем пространстве или задать в файловой системе иное расположение для проекта, которое и будет его рабочим пространством. Если нет необходимости расположить проект в строго определенном месте, используйте настройки, заданные по умолчанию (Create new project in workspace (Создать новый проект в рабочем пространстве) и Use default workspace (Использовать рабочее пространство, заданное по умолчанию)).

- **Target Name** (Имя целевой сборки) — образы системы Android, установленные вами в SDK, показаны в списке целевых версий сборки. Можно выбрать один из этих образов и соответствующего производителя, платформу (номер версии ОС Android) и **уровень интерфейса программирования приложений (API)** в качестве показателей, на которые будет рассчитано создаваемое вами приложение. Важнейшими параметрами здесь являются платформа и уровень API. Интерфейсы программирования приложений с более высоким уровнем API, чем тот, что вы задали в данном разделе, не смогут использоваться вашей программой. На данном этапе следует выбрать новейшую установленную у вас версию ОС Android и самый высокий уровень API.

- **Application name** (Имя приложения) — название приложения, которое будет видеть пользователь. Введите здесь `Test Application`.

- **Package name** (Имя пакета) — имя пакета создает пространство имен **Java**, которое уникально идентифицирует пакеты в приложении, а также должно уникально идентифицировать приложение Android среди всех остальных приложений, установленных в системе. Имя состоит из уникального доменного имени (это доменное имя того, кто опубликовал приложение) и имени конкретного приложения. В Java не все имена пакетов являются уникальными, но правила, используемые при назывании приложений Android, значительно снижают вероятность конфликтов. Например, мы воспользовались именем `com.oreilly.testapp`, но вы можете выбрать вариант, более подходящий для вашего домена.

Вы также можете воспользоваться именем `com.example.testapp`, так как доменное имя `example.com` зарезервировано для применения в подобных примерах.

- **Activity** (Активность) — элемент интерактивного пользовательского интерфейса в приложении Android. Обычно под активностью понимается группа элементов пользовательского интерфейса, которая занимает целый экран. При создании проекта можно использовать активность-каркас (*skeleton activity*), которая автоматически создается в системе. Если вы создаете приложение (а не служебную программу, которая может не иметь пользовательского интерфейса¹), то удобно начинать работу именно с активности-каркаса. В данном примере мы создадим активность под названием `TestActivity`.

- **Minimum SDK Version** (Минимальная версия SDK) — может содержать целое число, соответствующее номеру минимальной версии SDK, **требуемой для работы приложения**. Это поле используется для инициализации атрибута `uses-sdk` в описании² приложения. В файле описания сохраняются атрибуты приложения (см. пункт «Редактор описаний Android» подраздела «Компоненты плагина ADT для Eclipse» раздела «Компоненты комплекта для разработки ПО» данной главы). Как правило, этот номер должен быть таким же, как и номер уровня API для выбранной вами целевой версии сборки. Этот номер отображается в крайнем справа столбце таблицы со списком целевых версий сборки, показанной на рис. 1.6.

¹ В англоязычных источниках программы, не имеющие пользовательского интерфейса, называются *headless*, дословно — «без головы». — *Примеч. пер.*

² В английском языке этот файл называется *application manifest*. Встречаются также переводы «манифест» и «файл манифеста». — *Примеч. пер.*

Нажмите **Finish** (Готово), а не **Next** (Далее), чтобы создать проект Android. Этот проект появится в списке в левой части окна интегрированной среды разработки Eclipse (рис. 1.7).

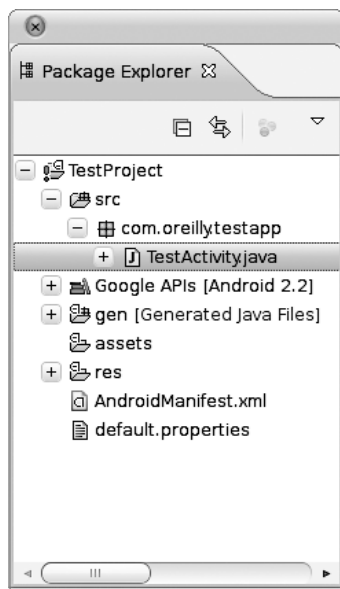


Рис. 1.7. Вид обозревателя пакетов (Package Explorer), в котором отображаются файлы и их компоненты, являющиеся составными частями вашего проекта

Если раскрыть вид с иерархией проекта (для этого нужно щелкнуть на знаке «+» в Windows или на треугольнике в Mac или Linux) рядом с именем проекта, то увидите различные составляющие проекта Android. Откройте каталог **src**, в нем вы увидите пакет Java с именем, которое ранее было указано в мастере установки. Раскройте этот пакет, в нем будет класс **Activity**, автоматически созданный мастером установки. Дважды щелкните на нем и просмотрите код на языке **Java**, относящийся к нашей первой программе Android:

```
package com.oreilly.demo.pa.ch01.testapp;
```

```
import android.app.Activity;
import android.os.Bundle;
import com.oreilly.demo.pa.ch01.R;
```

```
public class TestActivity extends Activity {
    /** Вызывается при первом создании активности. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Если параллельно с чтением книги вы выполняете все описываемые операции на компьютере и теперь видите на экране этот код, то установленный у вас SDK, по-видимому, работает правильно. Но давайте убедимся в этом и исследуем SDK чуть более подробно. Для этого запустим нашу первую программу в эмуляторе и на устройстве Android, если оно есть у вас под рукой.

Создание виртуального устройства Android (AVD)

В SDK для Android предоставляется эмулятор, имитирующий устройство с процессором ARM, на котором работает операционная система Android. Такой эмулятор предназначен для запуска программ Android на ПК. Виртуальное устройство Android (Android Virtual Device, AVD) — это набор параметров для эмулятора, который, опираясь на них, конфигурируется для использования конкретного образа системы (то есть определенной версии системы Android), а также задает другие параметры. К их числу относятся размер экрана, объем памяти и другие характеристики эмулируемого оборудования. Подробная документация о виртуальных устройствах Android приводится по адресу <http://developer.android.com/guide/developing/tools/avd.html>, а развернутая документация об эмуляторе находится здесь: <http://developer.android.com/guide/developing/tools/emulator.html>.

Поскольку мы просто собираемся убедиться, что установленный нами SDK работает, мы не будем подробно рассматривать ни виртуальные устройства Android, ни эмулятор. Здесь мы воспользуемся SDK Android и менеджером виртуальных устройств Android (рис. 1.8), чтобы запустить в таком виртуальном устройстве программу, которую только что создали при помощи мастера.



Рис. 1.8. SDK и менеджер виртуальных устройств Android

Вам потребуется создать виртуальное устройство Android с образом системы, причем этот образ должен не уступать по актуальности той версии сборки, которую

вы определили для проекта в качестве целевой. Нажмите кнопку **New** (Новое). Вы увидите диалоговое окно **Create New Android Virtual Device (AVD)** (Создать новое виртуальное устройство Android (AVD)), где нужно указать параметры нового AVD (рис. 1.9).

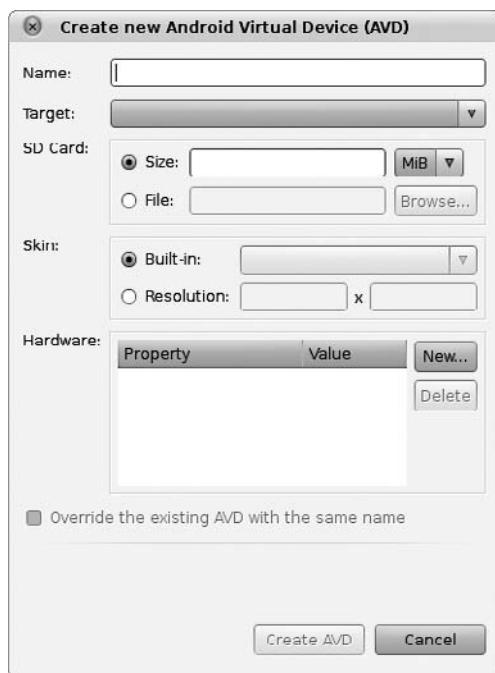


Рис. 1.9. Создание нового виртуального устройства Android

- **Name** (Имя) — имя виртуального устройства Android. Для обозначения AVD можно использовать любое имя, но рекомендуется выбирать такое имя, в котором упоминается используемый устройством номер системы.
- **Target** (Цель) — указывает, какой образ системы будет использоваться в данном виртуальном устройстве Android. Он должен быть не менее актуален, чем показатель, заданный при выборе целевой сборки для первого проекта Android.
- **SD Card** (Карта памяти) — для работы некоторых приложений требуется карта памяти, чтобы имеющийся в распоряжении объем памяти превышал объем флеш-памяти, встроенной в устройство Android. Если вы не собираетесь использовать при создании приложений и, соответственно, записывать на карту памяти достаточно большой объем информации (например, медиафайлы), то можно создать небольшую виртуальную карту памяти, например, объемом 100 Мбайт. При этом необходимо отметить, что большинство современных мобильных телефонов оснащено картами памяти объемом несколько гигабайт.
- **Skin** (Скин) — оболочка (скин) виртуального устройства Android обычно задает размер экрана. Чтобы проверить, работает ли установленная вами версия SDK, для этого параметра можно оставить значение, заданное по умолчанию. Однако

полезно использовать несколько эмуляторов, в которых указаны различные размеры экранов, чтобы гарантировать, что применяемые вами компоновки (макеты) страниц будут работать на различных устройствах.

- **Hardware (Оборудование)** — при конфигурации AVD позволяет задавать параметры, указывающие, какие разновидности оборудования доступны для испытаний. В данном проекте лучше оставить значения, заданные по умолчанию.

Заполните поля **Name (Имя)**, **Target (Цель)** и **SD Card (Карта памяти)**, а затем создайте новое виртуальное устройство Android, нажав кнопку **Create AVD (Создать виртуальное устройство Android)**. Вы не сможете запустить свою программу, если не создадите образ системы с не менее или более актуальной версией целевой сборки, чем та, что была задана в проекте Android.

Запуск программы на виртуальном устройстве Android

Теперь, когда у вас создан проект для сборки приложения, а также настроено виртуальное устройство Android с образом системы, совместимым с целевой версией сборки приложения и заданным уровнем API, **вы можете запустить свое приложение** и подтвердить, что SDK создал приложение Android и способен его открыть.

Для запуска приложения щелкните правой кнопкой мыши на созданном проекте и в появившемся контекстном меню выберите **Run as ► Android Application (Запустить как ► Приложение Android)**.

Если созданное виртуальное устройство Android **совместимо с данным приложением**, то это устройство запустится, на нем загрузится операционная система Android, после чего запустится приложение (рис. 1.10).



Рис. 1.10. Созданное приложение, работающее на виртуальном устройстве Android

Если вы сконфигурировали более одного совместимого AVD, то появится окно выбора устройства Android (**Android Device Chooser**), в котором будет предложено выбрать виртуальное устройство Android из числа уже работающих или из числа прикрепленных к вашей системе устройств Android, если такие имеются. Кроме того, здесь вы сами можете выбрать виртуальное устройство Android для запуска. На рис. 1.11 показано окно выбора виртуальных устройств Android с одним работающим устройством AVD и с одним, которое можно запустить.

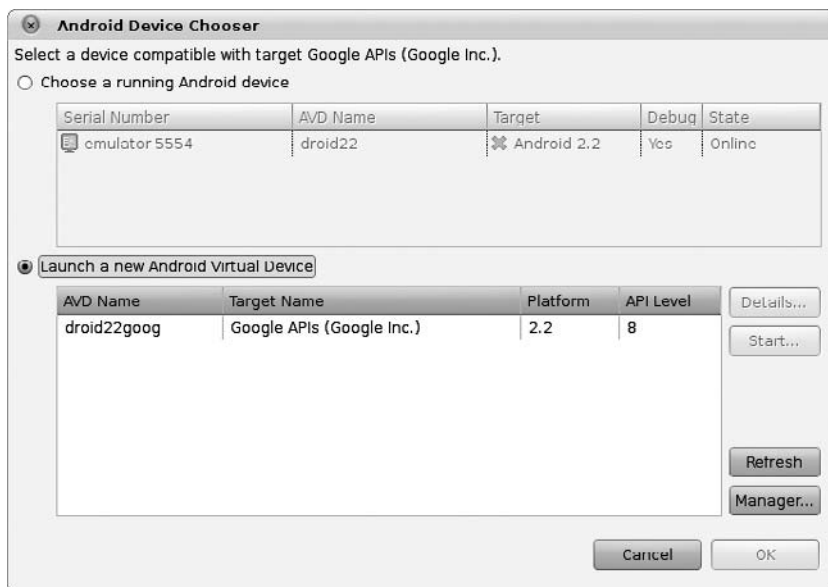


Рис. 1.11. Окно для выбора устройства Android

Запуск программы на реальном устройстве Android

Кроме того, созданную вами программу можно запустить и на большинстве реальных устройств Android.

Нужно подключить устройство к ПК с помощью USB-кабеля, при необходимости установить драйвер и задать права доступа к устройству, подключенному через USB.

Инструкции для подключения устройства к системе Windows, а также необходимые драйверы доступны по адресу <http://developer.android.com/sdk/win-usb.html>.

Если вы работаете с Linux, то для нового устройства Android нужно создать отдельный файл `rules`.

При работе с Mac OS X никакого дополнительного конфигурирования не требуется.

Подробная справочная информация об отладке USB содержится по следующему адресу: <http://developer.android.com/guide/developing/device.html>.

Кроме того, USB нужно настроить и на самом устройстве Android. Как правило, для этого следует запустить приложение **Settings** (Настройки), выбрать в нем **Applications** (Приложения), а затем **Development** (Разработка). Здесь вы увидите настройку, позволяющую включить или отключить отладку USB.

Если устройство AVD сконфигурировано или работает, то появится окно выбора устройств Android, в котором отобразится и подключенное к компьютеру через USB устройство Android, и виртуальное устройство Android.

Выберите устройство — и приложение для Android загрузится на него и запустится.

Устранение проблем с комплектом разработки ПО: отсутствие данных о целевой версии сборки

Если не получается создать новый проект или импортировать образец проекта из комплекта для разработки ПО (SDK), то, возможно, вы забыли задать целевые версии сборки для вашего SDK. Перечитайте подраздел «Добавление целевых платформ для сборки в SDK» раздела «Установка комплекта разработки ПО (SDK) Android и необходимые условия» данной главы и убедитесь, что в области Android окна **Preferences** (Настройки) перечислены те целевые версии сборки, которые вы задали в SDK (см. рис. 1.5).

Компоненты комплекта для разработки ПО

Комплект для разработки ПО Android (**Android SDK**) состоит в основном из стандартных компонентов, а также небольшого количества специализированных компонентов. Как правило, различные конфигурации, плагины и дополнения адаптируют эти компоненты к Android. Android SDK вобрал в себя весь опыт создания современного и полного комплекта для разработки ПО. Компания Google создавала этот комплект с расчетом на скорейшее укрепление Android на рынке. Вы сами в этом убедитесь, когда займетесь исследованием компонентов Android SDK. Средства Eclipse, язык Java, эмулятор QEMU и другие появившиеся ранее платформы, технологии и инструменты являются важнейшими частями Android SDK.

Создавая простую программу, которая подтверждает, что вы выполнили установку SDK правильно, вы уже использовали многие компоненты SDK. В данном разделе мы рассмотрим компоненты SDK, применяемые при написании данной программы, а также другие составляющие, с которыми вы будете работать.

Утилита Android Debug Bridge (adb)

adb — это программа, одновременно контролирующая и эмуляторы и устройства, а также запускающая оболочку, которая позволяет выполнять команды в окружении эмулятора или устройства. Утилита adb особенно удобна для установки или удаления программ с эмулятора или устройства. Документация по adb находится по адресу <http://developer.android.com/guide/developing/tools/adb.html>.

Инструмент отладки Dalvik Debug Monitor Server (DDMS)

Dalvik Debug Monitor Server (DDMS) — это регулировщик трафика, действующий между одиночным портом (Eclipse или другие инструменты отладки Java ищет этот порт для подключения к виртуальной машине Java) и несколькими другими портами. Данные порты предназначены для каждого реального или виртуального устройства Android. DDMS также предоставляет функции, доступные через автономный пользовательский интерфейс или через интерфейс, встраиваемый в Eclipse как часть плагина ADT.

При активизации DDMS из командной строки вы увидите нечто подобное тому, что показано на рис. 1.12.

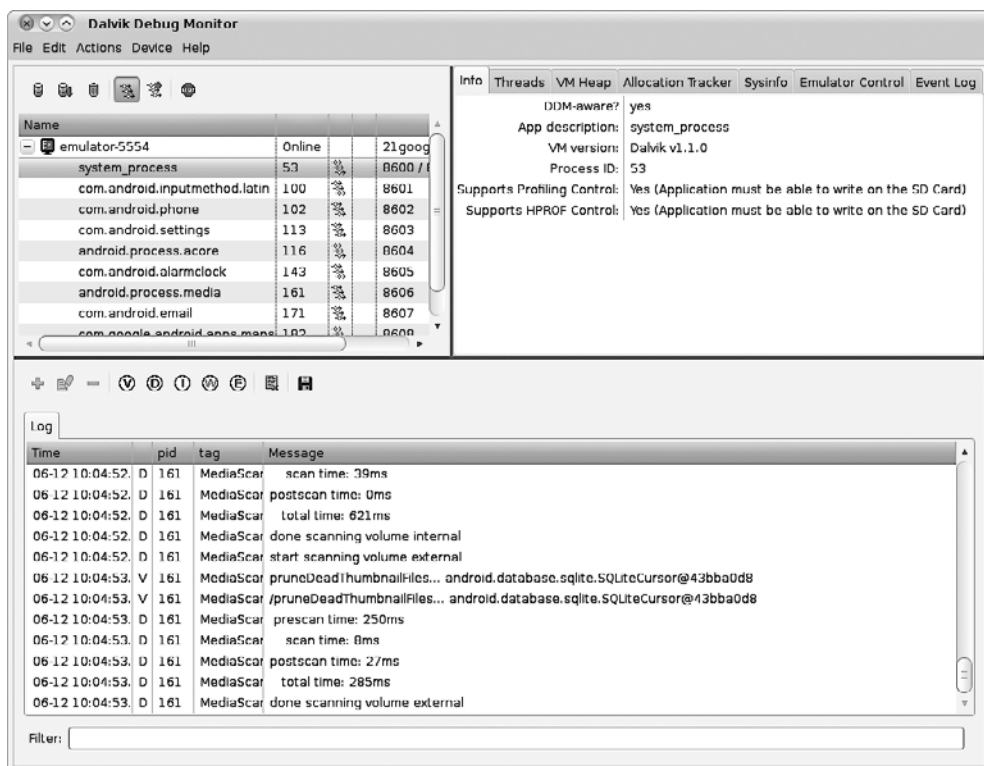


Рис. 1.12. Инструмент Dalvik Debug Monitor Server, работающий в автономном режиме

Через пользовательский интерфейс DDMS вы получаете доступ к следующим компонентам.

- Список реальных и виртуальных устройств, а также виртуальных машин, работающих на этих устройствах, — в верхней левой области окна DDMS вы видите список устройств Android, подключенных к ПК, а также все работающие на

компьютере виртуальные устройства Android. Под каждым устройством или виртуальным устройством перечислены задачи, решаемые виртуальной машиной Dalvik.

- Информация о виртуальной машине — при выборе одной из виртуальных машин Dalvik, работающих на устройстве или виртуальном устройстве, информация о ней выводится в верхней правой области окна.
- Информация о потоках — информация о потоках внутри каждого процесса содержится на вкладке **Threads** (Потоки) в верхней правой области окна DDMS.
- Менеджер файловой системы — файловую систему реального или виртуального устройства можно исследовать при помощи специального менеджера DDMS. Чтобы перейти в этот менеджер, нужно выбрать элемент **File Explorer** (Файловый менеджер) в меню **Device** (Устройство). В этом менеджере отображается иерархия файлов (рис. 1.13).

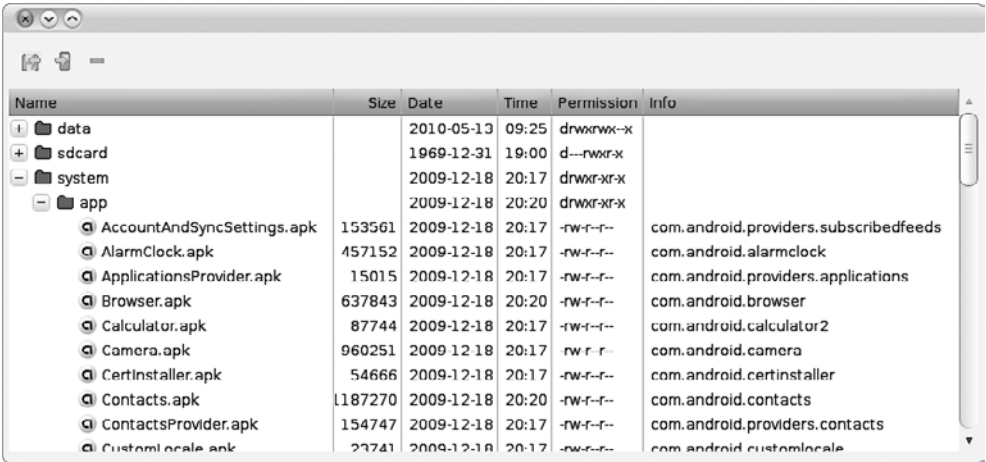


Рис. 1.13. Файловый менеджер системы DDMS

- Имитация телефонных звонков — вкладка **Emulator Control** (Управление эмулятором) в верхней правой области окна DDMS позволяет имитировать поступление на эмулятор телефонных вызовов или текстовых сообщений.
- Снимок экрана — команда **Screen Capture** (Снимок экрана) в меню **Device** (Устройство) делает скриншот с выбранного реального или виртуального устройства Android.
- Журналирование — в нижней области окна DDMS отображаются записи журнала о процессах, работающих на выбранном устройстве или виртуальном устройстве. Отображение записей можно фильтровать с помощью кнопок, расположенных на панели инструментов над записями журнала.
- Дампирование состояния устройств, приложений и мобильного радио — в меню **Device** (Устройство) содержится набор команд, позволяющий реальному или

виртуальному устройству дампитировать состояние целого устройства, одного из приложений или мобильного радио.

Подробная документация по DDMS доступна по адресу <http://developer.android.com/guide/developing/tools/ddms.html>.

Компоненты плагина ADT для Eclipse

Eclipse позволяет создавать проекты определенных типов, в том числе несколько типов проектов для Java. Плагин ADT обеспечивает возможность создавать и использовать проекты Android. Запуская новый проект Android, вы одновременно создаете иерархию проекта, а также все файлы, необходимые для минимальной сборки корректно работающего проекта Android. В проектах Android плагин ADT позволяет Eclipse применять компоненты, входящие в состав этого плагина, для редактирования, сборки, запуска и отладки проекта Android.

В некоторых случаях компоненты SDK можно применять как вместе с Eclipse, так и в автономном режиме. Но в большинстве примеров разработки приложений, рассмотренных в этой книге, наиболее целесообразным будет использование этих компонентов внутри Eclipse или в связи с ней.

Плагин ADT обладает многочисленными отдельными компонентами, и, несмотря на то что он считается подключаемым модулем и, соответственно, не очень серьезным дополнением к программе, этот плагин содержит существенный объем кода. Ниже будут описаны все важнейшие части плагина ADT, с которыми вам придется столкнуться при разработке программ Android с применением Eclipse.

Редактор макетов Android

Макеты (layouts) для пользовательских интерфейсов приложений Android можно создавать на языке XML. Плагин ADT дополнительно предоставляет визуальный редактор, удобный для компоновки и предварительного просмотра макетов Android. Когда вы открываете файл шаблона, плагин ADT автоматически запускает этот редактор для просмотра и редактирования файла. Вкладки, расположенные по нижнему краю области редактирования, позволяют переключаться между визуальным редактором и XML-редактором.

В более ранних версиях Android SDK редактор макетов Android был очень ограничен с функциональной точки зрения, поэтому использовался редко. Но в настоящее время рекомендуется применять визуальное редактирование макетов Android как основной метод работы. Автоматическая спецификация макетов повышает вероятность того, что макеты будут корректно работать на самых разнообразных устройствах Android.

Редактор описаний Android

При создании проекта Android в его состав вместе с программным кодом и ресурсами включается файл описания (manifest file). Этот файл сообщает системе Android, как следует устанавливать и использовать программы из архива, в котором содержится собранный проект. Файл описания создается на XML, а в плагине ADT предоставляется специальный XML-редактор для изменения такого описания.

Другие компоненты плагина ADT Eclipse, например компоновщики приложений, также могут изменять файл описания.

XML-редакторы для работы с другими XML-файлами в системе Android

В Android есть и другие XML-файлы, которые предназначены, например, для хранения информации (спецификации меню, ресурсы, в частности строковые) или для организации графических ресурсов приложения. Для таких файлов предусмотрены специализированные XML-редакторы, которые открываются при открытии таких файлов.

Сборка приложений Android

Сборка проектов Eclipse обычно происходит автоматически. Это означает, что не приходится специально объединять исходный код проекта и его ресурсы в готовый продукт, пригодный к развертыванию. В Android требуется выполнять специфичные для этой ОС этапы, позволяющие собрать файл, который затем можно будет развернуть в эмуляторе или на устройстве Android, а в плагине ADT есть программы, обеспечивающие выполнение этих этапов. В Android конечным результатом сборки проекта является файл APK. Выше в этой главе мы уже создали такой файл для тестового проекта, он находится в подкаталоге bin (в иерархии файлов проекта, расположенных в рабочем пространстве Eclipse).

Специфичные для Android компоновщики, предоставляемые в плагине ADT, позволяют применять для создания программ Android язык Java. При этом данные программы запускаются на виртуальной машине Dalvik, обрабатывающей собственные байт-коды. Это означает, что среди прочего такие компоновщики превращают байт-кодовый вывод Java, создаваемый компилятором Java, в байт-коды Dalvik. Они также создают APK-файлы, отличающиеся по структуре и содержанию от JAR-файлов.

Запуск и отладка приложений Android

При запуске или отладке проекта Android из Eclipse или внутри Eclipse файл APK этого проекта развертывается и запускается на виртуальном или реальном устройстве Android при помощи инструментов adb и DDMS для обмена информацией с AVD или реальным устройством Android. Еще в этом процессе участвует среда времени исполнения Dalvik, которая запускает код проекта. Плагин ADT доустанавливает в систему компоненты, позволяющие Eclipse выполнять эти операции.

DDMS

Инструмент отладки DDMS (служба наблюдения и отладки Dalvik) был описан выше. Мы рассмотрели службу наблюдения Dalvik и научились вызывать пользовательский интерфейс DDMS из командной строки. Кроме того, пользовательский интерфейс DDMS доступен и внутри Eclipse. Чтобы перейти в него, выберите команду Window ► Open Perspective ► DDMS (Окно ► Открыть перспективу ► DDMS) в меню Eclipse. Можно также по отдельности получить доступ к каждому из видов, из которых состоит перспектива DDMS. Это делается при помощи меню

Window ► Show View (Окно ► Отобразить вид). Далее нужно выбрать, например, вид LogCat.

Виртуальные устройства Android

Виртуальные устройства Android (AVD) создаются на основе QEMU-подобных эмуляторов, имитирующих аппаратное обеспечение устройства Android, а также образов системы Android, состоящих из программ Android, собранных для работы на эмулированном оборудовании. Для конфигурирования виртуальных устройств Android используется диспетчер SDK и AVD, задающий такие параметры, как объем эмулируемых запоминающих устройств и параметры экрана. Кроме того, он позволяет указывать, какой образ системы Android будет использоваться с каким эмулируемым устройством.

Виртуальные устройства Android обеспечивают тестирование программ в довольно широком диапазоне системных параметров. Для обеспечения такого широкого диапазона потребовалось бы достаточно большое количество реальных устройств, достать которые для тестирования может быть затруднительно. Поскольку QEMU-подобные эмуляторы оборудования, образы систем и параметры AVD допускают всевозможные изменения, вы можете тестировать в эмуляторе устройства и образных систем, реальные аналоги которых пока недоступны.

QEMU

QEMU — это основа виртуальных устройств Android. Но QEMU — универсальный инструмент, используемый в разнообразных системах для эмуляции, в том числе вне Android SDK. Когда вы конфигурируете QEMU опосредованно, через SDK и менеджер виртуальных устройств, иногда требуется скорректировать эмуляцию такими способами, которые не поддерживаются инструментарием SDK. А возможно, вы просто заинтересуетесь широтой и пределами возможностей, которые есть в QEMU. К счастью, QEMU собрал вокруг себя крупное и активное сообщество разработчиков и пользователей, с которым можно познакомиться на сайте <http://www.qemu.org>.

Диспетчер SDK и AVD

QEMU — это универсальная система эмуляции. Android SDK обеспечивает управление конфигурацией QEMU. Это целесообразно при создании эмуляторов, запускающих образы системы Android. Диспетчер SDK и AVD имеет пользовательский интерфейс, который позволяет управлять виртуальными устройствами Android на основе QEMU.

Другие инструменты SDK

Кроме основных инструментов, без которых, вероятно, вам не удастся обойтись при решении текущих задач в ходе большинства проектов, связанных с разработкой, в SDK есть и некоторые другие инструменты. Здесь описаны те из них, которые используются или активируются непосредственно самими разработчиками. Еще более полный набор компонентов SDK перечислен в статье **Tools (Инструменты)**,

размещенной на сайте документации Android по адресу <http://developer.android.com/guide/developing/tools/index.html>.

Инструмент просмотра иерархии

Инструмент просмотра иерархии отображает иерархию видов открытой в данный момент активности выбранного устройства Android и обеспечивает анализ этой иерархии. Таким образом, он позволяет видеть и диагностировать проблемы, возникающие с иерархией видов, прямо в ходе работы приложения либо проверять иерархии видов других приложений и смотреть, как они построены. Кроме того, этот инструмент дает возможность смотреть на дисплей с увеличением и с применением направляющих. Таким образом, становится проще обнаруживать проблемы, связанные с компоновкой активностей.

Layoutopt

Layoutopt — это статический анализатор, работающий с файлами компоновки, написанными на языке XML. Он может диагностировать некоторые проблемы, связанные с компоновкой элементов Android. **Подробная информация о Layoutopt** приводится по адресу <http://developer.android.com/guide/developing/tools/layoutopt.html>.

Monkey

Monkey — это инструмент автоматизации тестирования, работающий на эмуляторе или устройстве. Инструмент Monkey активируется при помощи утилиты adb, входящей в состав SDK. Adb позволяет запустить на эмуляторе или устройстве оболочку, а Monkey активируется из этой оболочки:

```
adb shell monkey --wait-dbg -p your.package.name 500
```

При активации Monkey таким образом в указанное приложение посылается 500 случайных событий (указанием на приложение служит имя пакета), после чего ожидается подключение отладчика. Подробная информация о Monkey содержится по адресу <http://developer.android.com/guide/developing/tools/monkey.html>.

sqlite3

Android использует базу данных SQLite в качестве основы для многих системных баз данных и предоставляет API для приложений, применяющих SQLite. Такой подход очень удобен для хранения и представления данных. **SQLite также оснащено** интерфейсом для работы с командной строкой, а команда sqlite3 позволяет разработчику дампитировать схемы баз данных, а также совершать с базами данных Android другие операции.

Разумеется, эти базы данных содержатся на реальном или виртуальном устройстве Android, поэтому команда sqlite3 доступна в оболочке adb. Подробное руководство, как получить доступ к командной строке sqlite3 из оболочки adb, дается по адресу <http://developer.android.com/guide/developing/tools/adb.html#shellcommands>. Вводная информация о работе с командой sqlite3 приводится в подразделе «Пример работы с базой данных (используется sqlite3)» раздела «Язык SQL» главы 9.

Keytool

Инструмент `keytool` генерирует ключи шифрования и используется плагином ADT для создания временных отладочных ключей, которыми подписывается код перед отладкой. Как правило, этот инструмент используется для подписывания сертификатов перед выпуском приложения. Этот процесс описан в пункте «Создание самозаверяющего сертификата» подраздела «Подписывание приложения» раздела «Подписывание приложения» главы 4.

Zipalign

`Zipalign` обеспечивает оптимизированный доступ к данным в готовых к выпуску версиях приложений Android. Оптимизация должна производиться после того, как приложение подписано для выпуска на рынок, поскольку подпись влияет на выравнивание байтов. Подробная информация о `Zipalign` содержится по адресу <http://developer.android.com/guide/developing/tools/zipalign.html>.

Draw9-patch

`9-patch` — это специфический ресурс Android, состоящий из девяти изображений, и полезный, например, когда вы хотите, чтобы кнопки увеличивались в размерах, а радиус их углов не изменялся. `Draw9-patch` — это специальная программа для рисования, позволяющая создавать и предварительно просматривать ресурсы таких типов. Подробная информация о `draw9-patch` содержится по адресу <http://developer.android.com/guide/developing/tools/draw9patch.html>.

android

Команда `android` может использоваться для активации диспетчера SDK и AVD из командной строки — этот процесс был описан в разделе об установке SDK («Комплект разработки ПО для Android») выше. Кроме того, данная команда может применяться для создания проекта Android из командной строки. При использовании таким образом эта команда обеспечивает генерирование всех каталогов проекта, файлов описания, свойств сборки, а также сценария компоновки проекта. Использование команды `android` подробно описано по адресу <http://developer.android.com/guide/developing/other-ide.html#CreatingAProject>.

Обеспечение актуальности

Комплект для разработки ПО на Java (JDK), среда Eclipse и комплект разработки ПО для Android (Android SDK) — результат работы разных производителей. Инструменты, которыми вы пользуетесь для разработки программ Android, могут стремительно меняться. Вот почему в нашей книге, и особенно в этой главе, мы рекомендуем вам сверяться с сайтом разработчиков Android, где размещается информация о новейших совместимых версиях ваших инструментов. Поддержание инструментария в совместимом и актуальном виде — задача, которую вам придется решать уже сейчас, вместе с обучением разработке программ для Android.

В Windows, Mac OS X и Linux имеются системные механизмы обновления, позволяющие поддерживать актуальность ваших программ. Но в силу тех принципов, по которым составляется Android SDK, вам придется одновременно поддерживать актуальность различных программных систем, причем использовать для этого разные механизмы.

Обеспечение актуальности комплекта для разработки ПО Android

SDK Android не является частью операционной системы вашего ПК, как и не входит в состав плагина Eclipse. Поэтому содержимое каталога SDK не обновляется ни операционной системой, ни Eclipse. В SDK имеется собственный механизм обновления, для работы с которым применяется пользовательский интерфейс диспетчера SDK и AVD. Как показано на рис. 1.14, выберите в левом подокне команду **Installed Packages** (Установленные пакеты), чтобы отобразить список компонентов SDK, установленных в вашей системе. Нажмите кнопку **Update All** (Обновить все), чтобы начать процесс обновления. В результате система покажет список доступных обновлений.



Рис. 1.14. Обновление SDK в диспетчере SDK и AVD

Как правило, требуется установить все доступные обновления.

Обеспечение актуальности Eclipse и плагина ADT

В то время как SDK требует обновления, происходящего вне рамок как вашей операционной системы, так и Eclipse, плагин ADT, а также все остальные компоненты Eclipse обновляются посредством собственной системы Eclipse для управления

обновлениями. Для обновления всех компонентов, входящих в состав среды Eclipse, в том числе плагина ADT, используется команда **Check for Updates** (Проверить наличие обновлений) в меню **Help** (Помощь). Эта команда отображает доступные обновления (рис. 1.15).

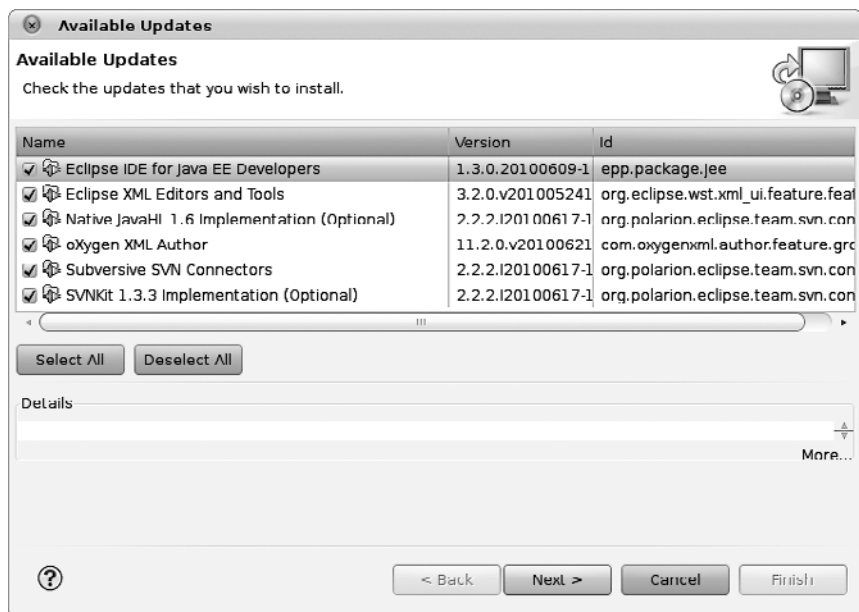


Рис. 1.15. Обновление компонентов Eclipse и плагина ADT

Как правило, для установки всех доступных обновлений используется кнопка **Select All** (Выделить все). Список обновлений зависит от того, какие модули Eclipse у вас установлены, и от того, не обновляли ли вы Eclipse в последнее время.

Обеспечение актуальности комплекта для разработки ПО на Java (JDK)

Java требуется обновлять значительно реже, чем SDK, плагин ADT и другие плагины Eclipse. Прежде чем переходить к обновлению JDK, ознакомьтесь со страницей **System Requirements** (Системные требования) на сайте разработчиков Android: <http://developer.android.com/sdk/requirements.html>.

Если требуется обновление, а вы работаете в системе Mac или Linux, проверьте наличие доступных обновлений для вашей системы, в частности, не появилась ли новая версия JDK.

Если комплект JDK был установлен в вашей системе поставщиком или вы установили этот комплект из какого-либо репозитория Linux, обновления будут осуществляться через специальный механизм обновлений, предусмотренный в вашей системе.

Примеры кода

Установив Android SDK и убедившись, что этот комплект работает, можно приступать к исследованию. Даже если вам не приходилось сталкиваться с классами фреймворка Android, а также если вы не имеете опыта работы с Java, вам стоит познакомиться с примерами кода, которые помогут дополнительно закрепить материал, связанный с установкой SDK, перед тем как переходить к другим частям этой книги.

Примеры кода SDK

Удобнее всего брать образцы кода прямо из SDK. Можно создать новый проект, основываясь на примерах кода из SDK, как показано на рис. 1.16. Выбранный вами образец появляется в левой области окна Eclipse. Там вы можете просмотреть файлы, из которых состоит образец, а также запустить этот образец и посмотреть, как он работает. Если вы умеете использовать интегрированные среды разработки для отладки кода, то, возможно, захотите задать определенные контрольные точки в образце кода и посмотреть, когда выполняются те или иные методы.

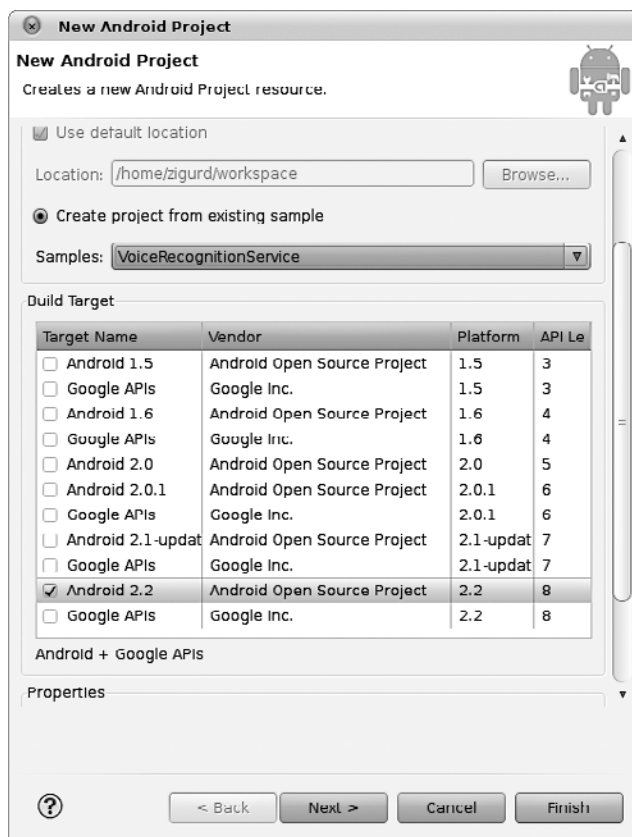


Рис. 1.16. Создание нового проекта с применением образца кода из SDK



В диалоговом окне, изображенном на рис. 1.16, нужно сначала выбрать целевую версию сборки, и лишь потом — образец. Образцы организованы в соответствии с уровнями API, и если не выбрать целевую версию сборки, раскрывающийся список будет пуст.

Для всех образцов приложений, которые содержатся в SDK, есть соответствующие статьи на сайте разработчиков Android. Здесь можно ознакомиться с более подробным описанием каждого из образцов. Все образцы перечислены на странице документации по адресу <http://developer.android.com/resources/samples/index.html>.

Есть более десятка приложений, одно из которых содержит демонстрационные версии API и представляет собой развернутое исследование большинства интерфейсов программирования приложений, используемых в Android. Если создать несколько проектов на основе данного кода, можно познакомиться с принципами работы конкретных программ. Кроме того, вы лучше усвоите материал следующих частей книги, даже если сейчас не совсем понимаете, о чем пойдет речь в определенных разделах.

Примеры кода из этой книги

Примеры кода из этой книги можно скачать с сайта книги по адресу http://oreil.ly/prog_android_2e.

О чтении кода

Хороший программист читает много кода. Примеры кода, которые мы привели в этой книге, задумывались и просто как образцы качественного кода на языке Java, и как примеры, демонстрирующие широту возможностей платформы Android.

Некоторые примеры, которые встретятся вам в этой книге, не дотягивают до уровня, который необходимо поддерживать для создания первоклассных образцов коммерческого программного обеспечения, отлично приспособленного и для расширяемости, и для технической поддержки. Многие примеры приложений являются удачными вариантами для случаев, когда программист ставит перед собой цель создать пример использования отдельного класса Java. Зачастую приложения Android являются развернутыми версиями кода из примеров, из-за чего они становятся неудобными для чтения и технической поддержки. Но это не означает, что вам не следует читать примеры, которые в большей мере, чем крупное приложение, ориентированы на конкретную ситуацию.

В следующей главе мы исследуем язык Java. В этой главе требуется научиться оценивать код примеров, усвоить хорошие практики разработки и проектирования. Мы хотели бы, чтобы вы научились брать пример и улучшать его, а также применять идеи, заложенные в образцах, для создания высококачественных продуктов.

2 Java для Android

Обучение языку Java не является целью этой книги, но в данной главе мы поможем вам понять, как именно Java используется в системе Android. Эта глава может быть интересна самым разным читателям: студентам, которые уже изучали язык Java, но еще не сталкивались с дилеммами, характерными для практического программирования на этом языке, программистам, которым доводилось работать с другими мобильными платформами и использовать иные версии Java, а теперь стоящим перед необходимостью заново изучить некоторые аспекты этого языка в контексте программирования для Android, и, наконец, программистам, профессионально работающим с Java, но незнакомым с некоторыми правилами и требованиями, которые действуют именно в системе Android.

Если вам покажется, что материал этой главы излагается слишком стремительно, рекомендуем почитать вводное пособие по Java. Если вы будете легко воспринимать материал, но та или иная концепция, описанная в этой главе, будет непонятна, можете познакомиться с руководством по Java, расположенным по адресу http://download.oracle.com/docs/cd/E17409_01/javase/tutorial/index.html.

Android и видоизменение клиентской разновидности Java

В настоящее время парадигма Android уже является наиболее распространенным вариантом создания интерактивных клиентов на языке Java. Хотя Java используется и в некоторых других библиотеках классов для пользовательского интерфейса (AWT, SWT, Swing, J2ME Canvas и т. д.), ни один из этих подходов не получил такого распространения, как Android. Любому программисту, работающему с Java, не помешает изучить пользовательский интерфейс Android просто для того, чтобы представлять себе, как будут выглядеть пользовательские интерфейсы, создаваемые на языке Java.

Нельзя сказать, что инструментарий Android отличается необоснованными изменениями Java в каких-то неожиданных направлениях. В данном случае мы сталкиваемся со значительным разнообразием размеров и форм дисплеев; отсутствует мышь (хотя может присутствовать сенсорный экран); ввод текста требует

трех касаний (triple-tap) и т. д. Кроме того, вероятно, придется иметь дело с многочисленными периферийными устройствами, в частности с сенсорами движения, навигационными блоками (GPS), камерами, разнообразными радиоустройствами и т. д. Наконец, Android постоянно работает от батареи, поэтому необходимо учитывать проблемы, связанные с питанием. Закон Мура применим к мощности процессоров и объему памяти (удвоение этих показателей примерно каждые два года), но он, к сожалению, неприменим к сроку службы аккумулятора. Когда процессоры были медленными, разработчики сталкивались с проблемами в области скорости и эффективности работы процессора. В свою очередь, разработчики ПО для мобильных устройств вынуждены учитывать расход энергии.

В этой главе мы быстро повторим обычный язык Java. Библиотеки, специфичные для Android, подробно рассмотрены в главе 3.

Система типов Java

В Java существуют две различные фундаментальные разновидности типов: объекты и примитивы. В Java безопасность типов (type safety) обеспечивается благодаря статической типизации, в соответствии с которой каждая переменная должна быть объявлена вместе с типом и только потом использоваться. Например, переменная с именем `i`, объявленная как переменная типа `int` (примитивное 32-битное целое число), выглядит так:

```
int i;
```

Этот механизм значительно отличается от языков с нестатической типизацией, где объявление переменной необязательно. Хотя при явном объявлении типа объем кода увеличивается, при таком подходе компилятору удастся избежать многочисленных программных ошибок (в частности, случайного создания переменных, обусловленного ошибками в названиях переменных, вызовом несуществующих методов и т. д.) с того самого момента, как создается действующий код. Подробное описание системы типов Java приводится в спецификации языка Java по адресу http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html¹.

Примитивные типы

Примитивные типы Java не являются объектами и не поддерживают операции с объектами, описанные ниже в этой главе. Изменить примитивный тип можно при помощи лишь ограниченного количества заранее заданных операторов: `+`, `-`, `&`, `|`, `=` и т. д. К примитивным типам Java относятся:

- `boolean` — булев тип, может иметь значения `true` (истинно) или `false` (ложно);
- `byte` — 8-разрядное целое число в дополнительном коде;
- `short` — 16-разрядное целое число в дополнительном коде;

¹ Перевод этого документа на русский язык — <http://www.uni-vologda.ac.ru/java/jls/index.html>. — *Примеч. пер.*

- `int` — 32-разрядное целое число в дополнительном коде;
- `long` — 64-разрядное целое число в дополнительном коде;
- `char` — 16-разрядное беззнаковое целое, представляющее собой символ UTF-16;
- `float` — 32-разрядное число IEEE 754 с плавающей точкой;
- `double` — 64-разрядное число IEEE 754 с плавающей точкой.

Объекты и классы

Java — это объектно-ориентированный язык. Следовательно, его основными составляющими являются не примитивы, а объекты — комбинации данных и процедуры для совершения операций над этими данными. Класс определяет поля (данные) и методы (процедуры), составляющие объект. В Java такое определение — шаблон, на основании которого создается объект, — является само по себе отдельным типом объекта и называется `Class`. В Java классы образуют основу системы типов, позволяющей разработчику описывать сколь угодно сложные объекты со сложным, специализированным состоянием и поведением.

В Java, как и в большинстве других объектно-ориентированных языков, типы могут наследовать свойства от других типов. Класс, наследующий свойства от другого, называется подтипом или подклассом родительского. В свою очередь, родительский класс может называться супертипом, или суперклассом. Класс, у которого есть несколько различных подклассов, может называться базовым типом этих подклассов.

И методы, и поля имеют глобальную область видимости (`global scope`) внутри класса. Они могут быть видимы и извне объекта, посредством ссылки на экземпляр класса.

Ниже приведено определение очень простого класса с одним полем — `ctr` — и одним методом — `incr`:

```
public class Trivial {  
    /** поле: его областью видимости является весь класс */  
    private long ctr;  
  
    /** изменение поля */  
    public void incr() { ctr++; }  
}
```

Создание объекта

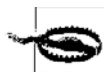
Новый объект, экземпляр определенного класса, создается при помощи ключевого слова `new`:

```
Trivial trivial = new Trivial();
```

Слева от оператора присваивания `=` в этом утверждении определяется переменная, которая называется `trivial`. Эта переменная имеет тип `Trivial`, поэтому ей могут быть присвоены лишь объекты типа `Trivial`. В правой части утверждения присваивания происходит выделение памяти для нового экземпляра класса `Trivial`

и инициализируется экземпляр. Оператор присваивания присваивает переменной ссылку на только что созданный объект.

Определение `ctr` в `Trivial` является совершенно безопасным, несмотря на то что явной инициализации не происходит. В Java гарантируется, что эта переменная будет инициализирована со значением 0, а также то, что все поля автоматически инициализируются при создании объекта: `boolean` инициализируется со значением `false`, числовые примитивные типы — со значением 0, а все объектные типы — со значением `null`.



Это касается только полей объектов. Локальные переменные необходимо инициализировать до того, как на них ставится ссылка!

Можно и в большей степени контролировать инициализацию объекта, добавив к определению его класса конструктор. Определение конструктора выглядит как метод, не считая того, что в нем не указывается тип возвращаемой переменной. Имя конструктора должно точно совпадать с именем конструируемого им класса:

```
public class LessTrivial {
    /** поле: его областью видимости является весь класс */
    private long ctr;

    /** конструктор: инициализация полей */
    public LessTrivial(long initCtr) { ctr = initCtr; }

    /** изменение поля */
    public void incr() { ctr++; }
}
```

На практике у каждого класса Java есть конструктор. Компилятор Java автоматически создает конструктор без аргументов, если не указано других конструкторов. Далее, если конструктор не вызывает автоматически какой-нибудь конструктор суперкласса, компилятор Java автоматически добавляет неявный вызов безаргументного конструктора суперкласса в качестве самого первого утверждения. Определение `Trivial`, данное выше (в котором отсутствует явное указание конструктора), на самом деле содержит конструктор, который выглядит так:

```
public Trivial() { super(); }
```

Поскольку в классе `LessTrivial` происходит явное определение конструктора, Java не создает конструктор по умолчанию автоматически. Это означает, что при попытке создания объекта `LessTrivial` без аргументов происходит ошибка:

```
LessTrivial fail = new LessTrivial(); // ОШИБКА!!
LessTrivial ok = new LessTrivial(18); // ... работает
```

Очень важно различать две концепции: безаргументный конструктор (`no-arg constructor`) и конструктор, задаваемый по умолчанию (`default constructor`). Конструктор по умолчанию — это такой конструктор, который Java автоматически добавляет к вашему классу, причем неявно и в том случае, если вы не определяете никаких других конструкторов. Бывает, что по умолчанию задается и безаргумент-

ный конструктор. Но, с другой стороны, безаргументный конструктор — это просто конструктор, для которого не заданы никакие параметры. Класс не обязательно должен иметь безаргументный конструктор. Определять такой конструктор также не обязательно, если это не требуется вам для какой-то определенной цели.



С безаргументным конструктором возможен особый случай, и его необходимо обсудить отдельно. Некоторые библиотеки требуют наличия возможности создавать новые объекты по общему принципу и по вашей команде. Например, во фреймворке JUnit необходимо иметь возможность создавать новые тестовые примеры, независимо от того, что именно на них тестируется. Библиотеки, которые формируют пакеты кода (выполняют маршалинг) и распаковывают их в постоянное хранилище данных или транслируют по сети (оба последних случая — варианты демаршалинга), также должны иметь такую возможность. Поскольку во время исполнения таким библиотекам будет крайне сложно определить точный протокол вызова отдельно взятого объекта, для них, как правило, требуется безаргументный конструктор.

Если класс обладает несколькими конструкторами, целесообразно каскадировать эти конструкторы, то есть гарантировать, что фактически только одна копия кода инициализирует экземпляр, а все остальные конструкторы его вызывают. Например, для удобства можно добавить конструктор к классу `LessTrivial`, чтобы случай получился более стандартным:

```
public class LessTrivial {
    /** поле: его областью видимости является весь класс */
    private long ctr;

    /** конструктор: счетчик init установлен в значение 0 */
    public LessTrivial() { this(0); }

    /** конструктор: инициализация полей */
    public LessTrivial(long initCtr) { ctr = initCtr; }

    /** изменение поля */
    public void incr() { ctr++; }
}
```

Каскадное расположение методов — это стандартная идиома Java для задания определенных значений по умолчанию. Весь код, который инициализирует объект, находится в одном самодостаточном методе или конструкторе, а остальные методы или конструкторы просто вызывают его. Особенно целесообразно применять такую идиому с конструкторами, которые должны выполнять явный вызов суперконструктора.

Конструкторы должны быть просты и выполнять не больше работы, чем необходимо для перевода объекта в стабильное исходное состояние. Можно представить себе, например, конструкцию объекта, обеспечивающего соединение с базой данных или сетью. Он может создать соединение, инициализировать его, проверить соединяемость — и все это сделать в конструкторе. На первый взгляд такой подход кажется совершенно оправданным, но на практике у вас получится код, которому не хватает модульности и который плохо поддается отладке и внесению изменений.

При более грамотном дизайне конструктор просто инициализирует состояние соединения как закрытое и передает соединение явному методу `open`, обеспечивающему выход в сеть.

Класс `Object` и его методы

Класс `Java Object` — `java.lang.Object` — это корневой предок любого класса. Любой объект `Java` относится к классу `Object`. Если при определении класса не происходит автоматического указания суперкласса, то описываемый класс является прямым подклассом класса `Object`. Класс `Object` определяет стандартную реализацию для некоторых ключевых поведений, свойственных любому объекту. Если они не переопределяются суперклассом, то поведения наследуются непосредственно от класса `Object`.

Методы `wait`, `notify` и `notifyAll`, относящиеся к классу `Object`, участвуют в поддержке параллелизма (`concurrency support`), реализованной в `Java`. Эти методы рассматриваются в подразделе «Управление потоками с помощью методов `wait()` и `notify()`» раздела «Идиомы программирования в `Java`» данной главы.

При помощи метода `toString` объект создает собственное строковое представление. Интересно отметить, что метод `toString` используется для сцепления со строкой: со строкой может быть сцеплен любой объект. В следующем примере показано два варианта вывода на печать одного и того же сообщения: выполнение обоих методов является идентичным. В обоих случаях создается новый экземпляр класса `Foo`, активируется его метод `toString`, а потом происходит сцепление результата со строкой литералов. Потом результат печатается:

```
System.out.println(
    "This is a new foo: " + new Foo());
System.out.println(
    "This is a new foo: ".concat((new Foo()).toString()));
```

Реализация `toString` в классе `Object` возвращает не слишком полезную строку, которая основана на расположении объекта в куче. Переопределение `toString` в своем коде — хороший первый шаг к облегчению отладки кода.

Методы `clone` и `finalize` можно назвать реликтами. Среда времени исполнения `Java` вызывает метод `finalize`, только если он переопределяется в подклассе. Если же класс явно определяет `finalize`, то он вызывается для объекта этого класса, но сразу же после этого объект подбирается сборщиком мусора. В `Java` не только нет однозначного ответа о том, когда это может произойти, но даже не гарантируется, что это вообще произойдет. Кроме того, вызов метода `finalize` может восстановить объект! Фокус вот в чем: сборщик мусора подбирает объекты, к которым не идут живые ссылки. Однако реализация `finalize` запросто создает новую живую ссылку, например добавляя финализируемый объект в какой-либо список! Поэтому существование метода `finalize` во многом препятствует оптимизации определяющего класса. Применяя при работе `finalize`, вы приобретаете проблемы, не сопоставимые с возможным положительным эффектом.

Метод `clone` создает объекты в обход конструкторов. Хотя `clone` и определяется применительно к `Object`, вызов его к объекту вызовет исключение, если только

объект не реализует интерфейс `Cloneable`. Метод `clone` — это, в сущности, оптимизация, которая может быть полезна, если на создание объекта уходит существенное количество ресурсов. В то время как при умелом обращении `clone` может быть полезен в некоторых случаях, есть еще копирующий конструктор — принимающий существующий экземпляр в качестве единственного аргумента. Как правило, он действует гораздо более прямолинейно и в большинстве случаев затратами на его использование можно пренебречь.

При помощи еще двух методов класса `Object` — `hashCode` и `equals` — «вызывающий» узнает, является ли вызываемый объект таким же, как и тот, с которым происходит сравнение.

При определении метода `equals` в документации по интерфейсам программирования приложений для класса `Object` оговаривается соглашение, которому должна подчиняться любая реализация метода `equals`. При корректной реализации метод `equals` имеет следующие атрибуты, и перечисленные ниже связанные с ними утверждения всегда должны выполняться:

```
reflexive
    x.equals(x)
symmetric
    x.equals(y) == y.equals(x)
transitive
    (x.equals(y) && y.equals(z)) == x.equals(z)
consistent
```

Если `x.equals(y)` является истинным в любой момент жизненного цикла программы, то он всегда является истинным при условии, что значения `x` и `y` не изменяются.

Нужно постараться, чтобы все это сделать правильно, и все равно процесс может оказаться удивительно сложным. Обычная ошибка, нарушающая возвратность, — это определение нового класса, который иногда оказывается равен уже существующему классу. Предположим, программа использует существующую библиотеку, которая определяет класс `EnglishWeekdays`. Теперь предположим, что вы дополнительно определяете класс `FrenchWeekdays`. Возникает очевидный соблазн определить для класса `FrenchWeekdays` метод `equals`, возвращающий `true`, сравнивающий один из элементов (дней) из класса `EnglishWeekdays` с его французским эквивалентом. Не поступайте так! Имеющийся «английский» класс ничего «не знает» о том, что вы создали новый класс, поэтому никогда не распознает экземпляры нового класса как равные экземплярам старого. Вы нарушаете принцип возвратности!

Методы `hashCode` и `equals` необходимо воспринимать в паре: если вы переопределяете один из них, то переопределяйте и другой. Во многих библиотечных процедурах `hashCode` считается оптимизированным инструментом приблизительной оценки того, равны ли (`equal`) два объекта. Сначала в таких библиотеках сравниваются хеш-коды двух объектов. Если два кода отличаются, то библиотека полагает, что нет необходимости производить какие-то еще, более затратные сравнения, так как объекты определенно не равны. Но суть расчета хеш-кодов заключается в том, чтобы рассчитать что-то очень быстро, и это хорошо подходит для метода `equals`. Если необходимо проверить каждую ячейку в большом массиве, чтобы рассчитать

хеш-код, то эта операция, вероятно, будет более длительной, чем точное сравнение. Другой крайностью является быстрый выход — возвращать 0, который будет единственным вариантом расчета хеш-кода. Это просто будет не слишком полезно.

Объекты, наследование и полиморфизм

Java поддерживает полиморфизм, одну из основных концепций объектно-ориентированного программирования. Язык считается полиморфным, если объекты одного типа могут проявлять различные поведения. Это происходит, когда подтипы данного класса могут присваиваться переменной, имеющей тип базового класса. Рассмотрим это подробнее.

При объявлении подтипов Java используется ключевое слово `extends`. Вот пример наследования в Java:

```
public class Car {
    public void drive() {
        System.out.println("Going down the road!");
    }
}

public class Ragtop extends Car {
    // изменить определение родителя
    public void drive() {
        System.out.println("Top down!");

        // можно использовать метод суперкласса
        super.drive();

        System.out.println("Got the radio on!");
    }
}
```

Ragtop — это подтип Car. Выше мы отмечали, что Car, в свою очередь, является подклассом Object. Ragtop изменяет определение метода `drive` из класса Car. В данном случае принято говорить, что он переопределяет (**override**) `drive`. И Car и Ragtop — оба имеют тип Car (но они оба не имеют тип Ragtop!), и поведения метода `drive`, закрепленные за ним в каждом из этих классов, различаются.

Продемонстрируем полиморфное поведение:

```
Car auto = new Car();
auto.drive();
auto = new Ragtop();
auto.drive();
```

Этот фрагмент кода будет компилироваться без ошибок (несмотря на присвоение Ragtop переменной, имеющей тип Car). Кроме того, он выполнится без ошибок и даст следующий вывод:

```
Going down the road!
Top down!
Going down the road!
Got the radio on!
```

В различные моменты жизненного цикла переменная `auto` содержит ссылки на два различных объекта типа `Car`. Один из этих объектов относится не только к типу `Car`, но также к подтипу `Ragtop`. Точное поведение утверждения `auto.drive()` зависит от того, на какой из объектов — первый или второй — переменная содержит ссылку в данный момент. Это пример полиморфного поведения.

Как и многие другие объектно-ориентированные языки, Java поддерживает приведение типов (type casting), что обеспечивает преобразование объявленного типа переменной в любой из типов, в сочетании с которыми переменная проявляет полиморфизм:

```
Ragtop funCar;
```

```
Car auto = new Car();  
funCar = (Ragtop) auto; //ОШИБКА! auto является Car, но не является Ragtop!  
auto.drive();
```

```
auto = new Ragtop();  
Ragtop funCar = (Ragtop) auto; // Работает! auto является Ragtop.  
auto.drive();
```

Приведение бывает необходимым, но если оно используется чрезмерно активно, это свидетельствует о том, что код несовершенен. Очевидно, что по правилам полиморфизма все переменные могут быть объявлены как переменные типа `Object`, а затем при необходимости с ними можно выполнять приведение. Но, поступая так, мы отказываемся от применения важнейшего механизма статической типизации.

В Java аргументы метода (то есть его актуальные параметры) должны быть полиморфны формальным аргументам. Аналогично методы возвращают значения, которые являются полиморфными объявленному типу возвращаемого значения. Продолжая наш автомобильный пример, рассмотрим следующий фрагмент кода, который будет компилироваться и возвращаться без ошибок:

```
public class JoyRide {  
    private Car myCar;  
  
    public void park(Car auto) {  
        myCar = auto;  
    }  
  
    public Car whatsInTheGarage() {  
        return myCar;  
    }  
  
    public void letsGo() {  
        park(new Ragtop());  
        whatsInTheGarage().drive();  
    }  
}
```

Метод `park` объявляется как способный принимать в качестве единственного параметра объект типа `Car`. Однако в методе `letsGo` он вызывается с объектом типа

Ragtop, который является подтипом типа Car. Аналогично переменной myCar присваивается значение типа Ragtop, и это значение возвращает метод whatsInTheGarage. Объект является Ragtop: если вы вызываете его метод drive, он сообщит вам и о top, и о radio. С другой стороны, поскольку он также является Car, он может использоваться везде, где применялся бы Car. Такая возможность замены подтипов является принципиальным примером мощности полиморфизма, а также его взаимодействия с безопасностью типов. Даже на этапе компиляции ясно, по назначению используется объект или нет. Безопасность типов позволяет компилятору находить ошибки на раннем этапе, в то время как их было бы гораздо сложнее искать, если бы они возникали во время исполнения.

Объявления final и static

В языке Java существует 11 ключевых слов-модификаторов, которые могут применяться к объявлению. Эти модификаторы изменяют поведение объявленного объекта, иногда довольно существенно. В предыдущих примерах некоторые модификаторы, например public и private, уже использовались без пояснений. Эти, а также некоторые другие модификаторы управляют областями действия и видимости. Далее мы рассмотрим их подробнее. В данном разделе мы поговорим еще о двух модификаторах, важных для полного понимания системы типов Java: final и static.

Объявление с ключевым словом final не может быть изменено. Классы, методы, поля, параметры и локальные переменные — все они могут быть final.

Применительно к классу final означает, что любая попытка определить подкласс вызовет ошибку. Например, класс String является final, поскольку строки должны быть постоянными (это значит, что после того, как строка создана, ее содержание должно оставаться неизменным). Немного поразмыслив над этой ситуацией, приходим к выводу, что выполнение такого условия можно *гарантировать* лишь в случае, если от String не смогут образовываться подтипы. Если бы от класса String можно было образовывать подтипы, то какая-нибудь коварная библиотека могла бы создать подкласс String — DeadlyString, — передать его экземпляр в ваш код и изменить его значение с fred на DROP TABLE. (Проиллюстрирована попытка внедрения в ваш код инородного SQL, который способен удалить части базы данных.) Это могло бы произойти сразу после того, как ваш код проведет проверку (валидацию) содержимого этой строки!

Применительно к методу final означает, что данный метод не может быть переопределен в подклассе. Разработчики используют final-методы для проектирования наследования, когда супертип должен сообщить подклассу поведение, которое сильно зависит от конкретной реализации, и супертип не может разрешить, чтобы это поведение изменялось. Фреймворк, в котором реализован универсальный (generic) кэш, может определить базовый класс под названием, например, CacheableObject. Программист, использующий фреймворк, создает подтип этого класса для каждого нового кэшируемого типа объекта. Однако для поддержания целостности фреймворка классу CacheableObject может потребоваться рассчитать ключ кэша, который должен быть единообразным в объектах любых типов. В данном случае метод computeCacheKey может быть объявлен как final.

Применительно к переменной — полю, параметру или локальной переменной — `final` означает, что значение присваивается переменной только один раз и после этого больше не изменяется. Данное ограничение дополнительно ужесточается компилятором: мало того, что значение не меняется, компилятор также должен иметь возможность убедиться, что оно в принципе не может быть изменено. В случае с полем это означает, что значение должно присваиваться либо в рамках объявления, либо в каждом конструкторе. Если не удастся инициализировать поле `final` в рамках объявления или в конструкторе либо при попытке присваивания значения где-нибудь еще возникнет ошибка.

В случае с параметрами `final` означает, что в рамках метода параметр всегда имеет только то значение, которое было передано при вызове. Попытка присваивания параметра с модификатором `final` вызовет ошибку. Разумеется, поскольку значение параметра в большинстве случаев является ссылкой на тот или иной объект, существует возможность того, что объект изменится. Применение ключевого слова `final` к параметру просто означает, что этот параметр не может быть заново присвоен.



В языке Java параметры передаются по значению: аргументы метода являются новыми копиями значений, которые были переданы при вызове. С другой стороны, очень многие элементы Java являются ссылками на объекты и Java копирует только ссылку, а не целый объект! Ссылки передаются по значению!

Переменная `final` может быть присвоена не более одного раза. Поскольку использование переменной без инициализации также является ошибкой, в языке Java переменная `final` присваивается только один раз без исключений. Присваивание может осуществляться где угодно в рамках внешнего (вышестоящего) блока перед использованием.

Объявление `static` относится к классу, в котором оно описывается, а не к экземпляру этого класса. `static` — это модификатор «статичности», противоположный «динамичности». Подразумевается, что, если сущность не объявлена как статическая, она является динамической. Проиллюстрируем это на примере:

```
public class QuietStatic {
    public static int classMember;
    public int instanceMember;
}

public class StaticClient {
    public void test() {
        QuietStatic.classMember++;
        QuietStatic.instanceMember++; // ОШИБКА!!

        QuietStatic ex = new QuietStatic();
        ex.classMember++; // ВНИМАНИЕ!
        ex.instanceMember++;
    }
}
```

В данном примере `QuietStatic` — это имя класса, а `ex` — ссылка на экземпляр этого класса. Статический член `classMember` — это атрибут класса. Чтобы сослаться на него, его нужно просто квалифицировать именем класса. С другой стороны, `instanceMember` является членом экземпляра класса. Попытка сослаться на него через ссылку класса вызовет ошибку. И это объяснимо. Существует много различных переменных с названием `instanceMember`, и каждая из них относится к отдельному экземпляру `QuietStatic`. Если явно не указать ту переменную, о которой вы говорите, Java не сможет понять, какая из этих переменных имеется в виду.

Как видно из второй пары утверждений, Java на самом деле допускает ссылки на (статические) переменные класса через ссылку на экземпляр. Но такая практика не приветствуется, так как приводит к путанице. В большинстве компиляторов и интегрированных сред разработки вы получите предупреждение, если попытаетесь сделать что-то подобное.

Неявные черты, отличающие статические объявления от динамических, могут быть довольно тонкими. Опять же если при статическом определении мы имеем дело ровно с одной копией, то при динамическом определении на каждый экземпляр приходится по копии. Статические члены класса обеспечивают сохранение информации, которая находится в общем пользовании у всех членов класса. Вот пример кода:

```
public class LoudStatic {
    private static int classMember;
    private int instanceMember;

    public void incr() {
        classMember++;
        instanceMember++;
    }

    @Override public String toString() {
        return "classMember: " + classMember
            + ", instanceMember: " + instanceMember;
    }

    public static void main(String[] args) {
        LoudStatic ex1 = new LoudStatic();
        LoudStatic ex2 = new LoudStatic();
        ex1.incr();
        ex2.incr();
        System.out.println(ex1);
        System.out.println(ex2);
    }
}
```

И его вывод:

```
classMember: 2, instanceMember: 1
classMember: 2, instanceMember: 1
```

Исходным значением переменной `classMember` в предыдущем примере является 0. Оно увеличивается на 1 каждым из двух отдельных экземпляров. Теперь оба экземпляра видят новое значение — 2. Значение переменной `instanceMember` также начинается с 0 в каждом экземпляре. Каждый экземпляр выполняет приращение собственной копии и просматривает значение собственной переменной, равное 1.

Статические определения классов и методов подобны в том, что в обоих случаях статический объект видим по своему квалифицированному имени, а динамический объект видим только по ссылке на экземпляр. Но, кроме сходства, существуют и гораздо более неочевидные различия.

Существенной разницей в поведении методов, объявленных статически и динамически, является то, что статически объявленные методы не могут быть переопределены в подклассе. Например, компиляция следующего кода не удастся:

```
public class Star {
    public static void twinkle() { }
}

public class Arcturus extends Star {
    public void twinkle() { } // ОШИБКА!!
}

public class Rigel {
    // а вот это работает
    public void twinkle() { Star.twinkle(); }
}
```

В современном языке **Java практически нецелесообразно использовать статические методы**. В ранних реализациях Java динамическое назначение методов протекало значительно медленнее, чем статическое. Разработчики предпочитали пользоваться статическими методами для оптимизации кода. В Android, где среда Dalvik обеспечивает динамическую компиляцию, в оптимизации такого рода больше нет нужды. Чрезмерное использование статических методов обычно считается признаком некачественной архитектуры.

Разница между статически и динамически объявленными классами особенно тонкая. Большинство классов, из которых состоит приложение, являются статическими. Как правило, класс определяется на *верхнем уровне* — за пределами внешнего блока. Предполагается, что все такие объявления должны быть статическими. С другой стороны, большинство других объявлений происходит внутри внешнего блока определенного класса, то есть по умолчанию они являются динамическими. В то время как большинство полей по умолчанию являются динамическими и требуют модификатора, чтобы стать статическими, большинство классов сразу являются статическими.



Блок — это фрагмент кода между двумя фигурными скобками: { и }. Все, что угодно (переменные, типы, методы и т. д.), определенное внутри блока, видимо внутри этого блока и внутри статически вложенных блоков. Кроме случая со специальным блоком, в котором определяется класс, все компоненты, определенные в блоке, невидимы за пределами блока.

На самом деле система очень стройная. Согласно тому, как мы определили признак `static` — принадлежность к классу, а не к экземпляру этого класса, — объявления верхнего уровня должны быть статическими (поскольку они не относятся ни к какому классу). Однако при объявлении внутри внешнего блока, например внутри определения класса верхнего уровня, определение класса по умолчанию также становится динамическим. Чтобы создать динамически объявленный класс, просто определите его внутри какого-нибудь другого класса.

Мы подошли к разнице между статическим и динамическим классами. Динамический класс имеет доступ к членам экземпляра вышестоящего класса (так как относится к этому экземпляру). Статический класс — не имеет. Вот пример кода:

```
public class Outer {
    public int x;

    public class InnerOne {
        public int fn() { return x; }
    }

    public static class InnerTube {
        public int fn() {
            return x; // ОШИБКА!!!
        }
    }
}

public class OuterTest {
    public void test() {
        new Outer.InnerOne(); // ОШИБКА!!!
        new Outer.InnerTube();
    }
}
```

Немного поразмышляйте — и сразу поймете, что здесь происходит. Поле `x` — это член экземпляра класса `Outer`. Иными словами, существует множество переменных `x`, по одной для каждого экземпляра времени исполнения (runtime instance) класса `Outer`. Класс `InnerTube` входит в состав класса `Outer`, но не какого-либо экземпляра класса `Outer`. Он никак не может идентифицировать `x`. С другой стороны, класс `InnerOne` относится к экземпляру `Outer`, поскольку является динамическим. Можно представить, что здесь мы имеем отдельный класс `InnerOne` для каждого экземпляра `Outer` (хотя фактически это не так). Следовательно, `InnerOne` имеет доступ к членам того экземпляра `Outer`, к которому этот `InnerOne` относится.

`OuterTest` позволяет убедиться, что, как и с полями, можно использовать статическое внутреннее определение (в данном случае — создать образец класса), просто воспользовавшись его классифицированным именем. Однако динамическое определение полезно только в контексте экземпляра.

Абстрактные классы

При объявлении класса в языке Java можно обойтись без реализации одного или нескольких методов, объявив класс и нереализованные методы как `abstract`:

```
public abstract class TemplatedService {

    public final void service() {
        // Подклассы готовят сервис каждый по-своему,
        prepareService();
        // ... но все они работают с одним и тем же сервисом.
        runService()
    }

    public abstract void prepareService();

    private final void runService() {
        // реализация сервиса ...
    }
}

public class ConcreteService extends TemplatedService {
    void prepareService() {
        // настройка сервиса
    }
}
```

Абстрактный класс не может быть инстанцирован. Подтипы абстрактного класса должны либо представлять определения всех абстрактных методов в суперклассе, либо сами быть объявлены как абстрактные.

Из примера видно, что абстрактные классы полезны при реализации общего паттерна, который предоставляет фрагмент кода, пригодный для многократного использования и допускающий необходимую адаптацию в определенные моменты в ходе выполнения. Многократно используемые фрагменты кода реализуются как абстрактный класс. Подтипы адаптируют шаблон, реализуя абстрактные методы.

Более подробно об абстрактных классах рассказано в руководстве по Java, расположенном по адресу <http://download.oracle.com/javase/tutorial/java/IandI/abstract.html>.

Интерфейсы

В других языках программирования (например, C++, Python и Perl) существует возможность, называемая множественным наследованием реализации (multiple implementation inheritance). При этом объект может наследовать реализации методов более чем от одного родительского класса. Такие иерархии наследования могут быть очень сложны и проявлять неожиданные свойства (например, наследовать две одноименные переменные поля от двух различных суперклассов). Разработчики Java решили отказаться от множественного наследования реализации в пользу простоты языка. В отличие от указанных языков, в Java класс может дополнять только один суперкласс.

Однако взамен многократного наследования реализации Java позволяет классу наследовать свойства от нескольких типов, используя при этом концепцию интерфейса. Интерфейсы позволяют определять тип, не определяя при этом его реализацию. Интерфейс можно считать абстрактным классом, все методы которого тоже

абстрактны. Количество интерфейсов, которые может реализовать тот или иной класс, не ограничено.

```
public interface Growable {
    // объявление сигнатуры, но не реализации
    void grow(Fertilizer food, Water water);
}

public interface Eatable {
    // еще одна сигнатура без реализации
    void munch();
}

/**
 * реализующий класс должен реализовать все методы интерфейса
 */
public class Beans implements Growable, Eatable {

    @Override
    public void grow(Fertilizer food, Water water) {
        // ...
    }

    @Override
    public void munch() {
        // ...
    }
}
```

Кроме того, интерфейсы предоставляют способ определения типа отдельно от реализации этого типа. Подобное разделение можно представить себе и в повседневной жизни. Например, если вы с коллегой хотите приготовить мохито¹, то можно распределить задачи и послать коллегу за мятой. Когда вы начинаете смешивать ингредиенты в стакане, уже неважно, купил ли коллега мяту в магазине или сходил во двор и просто нарвал свежих листьев. Важно, что у вас есть мята.

В качестве другого примера, иллюстрирующего потенциал интерфейсов, рассмотрим программу, которая должна отображать список контактов, отсортированных по адресам электронной почты. Как вы понимаете, в библиотеках Android содержатся стандартные процедуры для сортировки объектов. Однако поскольку эти процедуры являются стандартными, в них не заложена идея о том, как должно происходить упорядочение элементов конкретного класса. Чтобы воспользоваться библиотечными процедурами сортировки, классу нужен способ определения такого порядка. В Java классы решают эту задачу при помощи интерфейса `Comparable`.

Объекты типа `Comparable` реализуют метод `compareTo`. Один объект принимает другой похожий объект в качестве аргумента и возвращает целое число, которое указывает, является ли объект-аргумент большим, равным или меньшим, чем це-

¹ Кубинский коктейль из светлого рома и листьев мяты. — *Примеч. пер.*

левой. Библиотечные процедуры способны сортировать все, что является `Comparable`. Программный тип `Contact` должен быть `Comparable` и реализовывать `compareTo`, чтобы контакты могли быть отсортированы:

```
public class Contact implements Comparable<Contact> {
    // ... другие поля
    private String email;

    public Contact(
        // другие параметры...
        String emailAddress)
    {
        // ... init других полей от соответствующих параметров
        email = emailAddress;
    }

    public int compareTo(Contact c) {
        return email.compareTo(c.email);
    }
}

public class ContactView {
    // ...

    private List<Contact> getContactsSortedByEmail(
        List<Contact> contacts)
    {
        // получить отсортированный список контактов
        // не составляет никакого труда
        return Collections.sort(contacts);
    }

    // ...
}
```

С внутрисистемной точки зрения процедуре `Collections.sort` известно лишь о том, что `contacts` — это список элементов, относящихся к типу `Comparable`. Процедура активирует относящийся к классу метод `compareTo`, чтобы решить, как упорядочить эти элементы.

Как видно из этого примера, интерфейсы позволяют разработчику многократно использовать стандартные процедуры, которые могут отсортировать любой список объектов, реализующий `Comparable`. Если выйти за рамки этого простого примера, то можно сказать, что интерфейсы **Java позволяют претворять на практике различные паттерны программирования**, хорошо описанные в других источниках.

Исключения

В языке Java исключения используются как удобный инструмент, позволяющий справляться с необычными ситуациями. Зачастую такие условия сводятся к возникновению ошибок.

Код, пытающийся произвести синтаксический разбор веб-страницы, не может, например, продолжить работу, если у него не получается считать страницу из сети. Разумеется, можно проверить результаты попытки такого считывания, и продолжить работу лишь в том случае, если попытка окажется удачной, как показано в следующем примере:

```
public void getPage(URL url) {
    String smallPage = readPageFromNet(url);
    if (null != smallPage) {
        Document dom = parsePage(smallPage);
        if (null != dom) {
            NodeList actions = getActions(dom);
            if (null != action) {
                // здесь обрабатывается действие...
            }
        }
    }
}
```

С применением исключений код становится более красивым и надежным:

```
public void getPage(URL url)
    throws NetworkException, ParseException, ActionNotFoundException
{
    String smallPage = readPageFromNet(url);
    Document dom = parsePage(smallPage);
    NodeList actions = getActions(dom);
    // здесь обрабатывается действие...
}

public String readPageFromNet(URL url) throws NetworkException {
    // ...
}
public Document parsePage(String xml) throws ParseException {
    // ...
}
public NodeList getActions(Document doc) throws ActionNotFoundException {
    // ...
}
```

В этом варианте кода каждый метод, вызываемый от `getPage`, использует исключения для моментального «короткого замыкания» всей дальнейшей обработки, если вдруг что-то пойдет неправильно. Принято говорить, что методы *выбрасывают* исключения. Например, метод `getActions` может выглядеть примерно так:

```
public NodeList getActions(Document dom)
    throws ActionNotFoundException
{
    Object actions = XPathFactory.newXPath().compile("//node/@action")
        .evaluate(dom, XPathConstants.NODESET);
    if (null == actions) {
        throw new ActionNotFoundException("Action not found");
    }
    return (NodeList) actions;
}
```

При выполнении утверждения `throw` обработка сразу же прекращается и начинается со следующего блока перехвата (`catch block`). Вот пример блока «попытка-перехват» (`try-catch`):

```
for (int i = 0; i < MAX_RETRIES; i++) {
    try {
        getPage(theUrl);
        break;
    }
    catch (NetworkException e) {
        Log.d("ActionDecoder", "network error: " + e);
    }
}
```

Этот код выполняет повторные попытки при отказе сети. Обратите внимание, что он даже не находится в методе `readPageFromNet`, который выбросил исключение `NetworkException`. Когда мы говорим, что обработка возобновляется с ближайшего блока «попытка-перехват», мы подходим к обсуждению интересного способа, которым Java делегирует ответственность за исключения.

Если отсутствует блок «попытка-перехват», который окружал бы утверждение `throw` внутри метода, из-за выброшенного исключения может показаться, что возврат метода происходит мгновенно. Больше никакие инструкции не выполняются и никакое значение не возвращается. В частности, в предыдущем примере ни один фрагмент кода, следующий за попыткой получить страницу из сети, не учитывает той проблемы, что предпосылка — страница была прочтена — не выполнена. Принято говорить, что метод был *резко завершен*, и в данном случае управление возвращается к `getActions`. Поскольку `getActions` также не содержит блок «попытка-перехват», он тоже резко завершается. Управление передается обратно (вверх по стеку) к вызывающей программе.

В данном случае при выбросе `NetworkException` управление возвращается к первому утверждению внутри блока перехвата (`catch`), приведенного в качестве примера. Это утверждение является инструкцией занесения сетевой ошибки в журнал. Принято говорить, что исключение было *перехвачено* первым утверждением `catch`, чей аргумент относится к тому же типу, что и тип выброшенного исключения, либо к супертипу этого исключения. Обработка возобновляется с первого утверждения в блоке перехвата и после этого продолжается в обычном режиме.

В данном случае сетевая ошибка, возникающая при попытке считать страницу из сети, вызовет резкое завершение как `ReadPageFromNet`, так и `getPage`. После того как блок перехвата запишет в журнал сообщение о неудаче, цикл `for` вновь попытается получить страницу, и так до достижения показателя `MAX_RETRIES`.

Полезно иметь ясное представление о том, как организован корень дерева классов исключений в Java (рис. 2.1).

Все исключения являются подклассами `Throwable`. Практически не существует причин, по которым стоило бы ставить в коде ссылку на `Throwable`. Считайте его просто абстрактным базовым классом с двумя подклассами: `Error` и `Exception`. `Error` и его подклассы предназначены для решения проблем, возникающих с самой средой времени исполнения Dalvik. Вы, конечно, можете написать код, который, казалось бы, сможет перехватывать `Error` (или `Throwable`), но на самом деле отлавливать

их вы не сможете. В частности, очевидным доказательством этого является пример с ужасающим OOME — так сокращенно называют ошибку `OutOfMemoryException`¹. Когда в системе Dalvik заканчивается память, она, возможно, не сможет завершить выполнение даже мельчайшего кода операции! Если вы напишете хитрый код, который попытается перехватить OOME, а потом выделить определенный блок заранее зарезервированной памяти, это может сработать — или не сработать. Если вы пишете код, который пытается перехватить `Throwable` или `Error`, считайте, что вы таскаете воду решетом.



Рис. 2.1. Базовые классы исключений

Java требует, чтобы в сигнатуру метода входили исключения, которые этот метод может выбрасывать. В предыдущем примере `getPage` объявляет, что он выбрасывает три исключения, поскольку использует три метода, каждый из которых может выбросить одно. Методы, вызывающие `getPage`, должны, в свою очередь, объявлять все три исключения, выбрасываемые `getPage`, а также и все другие исключения, которые могут выбрасываться любыми другими методами, которые может вызывать `getPage`.

Как вы понимаете, эта ситуация оказывается обременительной для методов, расположенных достаточно высоко в дереве вызовов. Возможно, методу верхнего уровня потребуется объявить десятки видов самых разных исключений просто потому, что он вызывает методы, способные их выбрасывать. Для облегчения этой проблемы можно создать дерево исключений, конгруэнтное дереву приложения. Помните, что от метода требуется только объявлять супертипы для всех исключений, которые он выбрасывает. Если создать базовый класс под названием `MyApplicationException`, а потом сделать из него подклассы `MyNetworkException`

¹ Дословный перевод — «Исключение, вызванное тем, что израсходована память». — Примеч. пер.

и `MyUIException` соответственно для подсистем, занятых работой с сетью и с пользовательским интерфейсом, то в коде верхнего уровня понадобится обрабатывать только `MyApplicationException`.

Но на самом деле это половинчатое решение. Предположим, выполнение сетевого кода не удастся где-нибудь по пути к точке, расположенной глубоко в недрах приложения. Например, не удастся установить сетевое соединение. Исключение продолжает выскакивать, несмотря на все повторные попытки и альтернативы. В какой-то момент оно становится совершенно размытым и сводится к тому, что «что-то пошло не так». Например, конкретное исключение базы данных не имеет никакого значения для кода, который пытается автоматически подставить набираемый телефонный номер. При добавлении исключения к сигнатуре метода именно на этом этапе возникает ситуация, которая попросту неудобна: вы можете и прямо объявить, что все ваши методы выбрасывают `Exception`.

`RuntimeException` — это особый подкласс `Exception`. К подклассам `RuntimeException` относятся непроверяемые исключения, которые не обязательно объявлять. Например, следующий код будет скомпилирован без ошибок:

```
public void ThrowsRuntimeException() {  
    throw new RuntimeException();  
}
```

В сообществе разработчиков Java идут серьезные дискуссии о том, когда следует и когда не следует использовать непроверяемые исключения. Очевидно, в приложении можно пользоваться только непроверяемыми исключениями и никогда не объявлять никакие исключения ни в одной из сигнатур вашего метода. В некоторых школах программирования на Java это даже рекомендуется. Тем не менее, применяя проверяемые исключения, вы можете использовать компилятор для проверки своего кода, а это весьма соответствует духу статической типизации. Вы будете ориентироваться при работе на свой опыт и вкус.

Фреймворк коллекций Java

Фреймворк коллекций Java — это один из наиболее мощных и удобных инструментов этого языка. Он предоставляет объекты, которые являются коллекциями объектов: списки, наборы и карты. Все интерфейсы и реализации, составляющие эту библиотеку, находятся в пакете `java.util`.

В `java.util` осталось несколько устаревших классов, которые можно считать реликтами. В сущности, они уже не относятся к фреймворку. Лучше их запомнить и стараться не использовать. Речь о классах `Vector`, `Hashtable`, `Enumeration` и `Dictionary`.

Типы интерфейсов, используемых с коллекциями

Все пять основных типов объектов, относящихся к библиотеке коллекций, представлены тем или иным интерфейсом.

- `Collection` (Коллекция) — основной (корневой) тип для всех объектов библиотеки коллекций. `Collection` — это группа объектов, не обязательно упорядоченных и не обязательно поддающихся адресации. Коллекция может содержать дублирующие объекты. Можно удалять из коллекции объекты, добавлять

в нее новые объекты, узнавать ее размер и итерировать объекты (об итерации мы подробно поговорим чуть позже).

- **List (Список)** — упорядоченная коллекция. Существует механизм ассоциирования (mapping) целых чисел 0 и length-1 с **объектами списка**. **Список может содержать дублирующиеся объекты**. К списку применимы все те операции, которые применимы к коллекции. Кроме того, вы можете ассоциировать индекс с элементом и, наоборот, элемент с индексом, пользуясь методами `get` и `indexOf`. Можно также изменить элемент с заданным индексом, применив метод `add(index, e)`. Итератор списка возвращает элементы в упорядоченном виде.
- **Set (Набор)** — неупорядоченная коллекция, в которой отсутствуют дублирующиеся элементы. К набору применимы все те же операции, что и к коллекции. Если вы попытаетесь добавить в набор элемент, копия которого там уже содержится, это не изменит размера `Set`.
- **Map (Ассоциативный контейнер)** — напоминает список, за исключением того, что она отображает не числа на объекты, а объекты-ключи на коллекцию объектов-значений. В карту можно добавлять пары «ключ — значение», а также удалять из нее имеющиеся пары, узнавать размер карты и проводить в ней итерацию, так же как и при работе с любой другой коллекцией. Карты также могут использоваться для отображения слов на их определения, отображения дат на события либо URL (уникальных идентификаторов ресурсов) на кэшированный контент.
- **Iterator (Итератор)** — возвращает элементы коллекции, от которой он является производным. Каждый элемент возвращается один, и только один раз в ответ на вызовы к методу `next` этого итератора. Это предпочтительное средство для обработки всех элементов коллекции. Вместо:

```
for (int i = 0; i < list.size(); i++) {  
    String s = list.get(i)  
    // ...  
}
```

следует делать так:

```
for (Iterator<String> i = list.iterator(); i.hasNext();) {  
    String s = i.next();  
    // ...  
}
```

На самом деле второй вариант можно сократить до следующей формы:

```
for (String s: list) {  
    // ...  
}
```

Типы реализации коллекций

Перечисленные выше интерфейсы допускают несколько вариантов реализации, каждый из которых применим в специфических случаях. Среди наиболее распространенных реализаций интерфейсов можно назвать следующие.

- **ArrayList** — список на основе массива. В нем быстро происходит индексация, но медленно изменяется размер.

- `LinkedList` — список, который может быстро менять размер, но медленнее индексируется.
- `HashSet` — набор, который реализуется в виде хеша. `add`, `remove`, `contains` и `size` выполняются за постоянное время, то есть для их работы нужен хорошо организованный хеш. `HashSet` может содержать `null` (не более одного).
- `HashMap` — реализация интерфейса `Map`, использующего в качестве индекса хеш-таблицу. `add`, `remove`, `contains` и `size` выполняются за постоянное время, то есть для их работы нужен хорошо организованный хеш. `HashMap` может содержать один, и только один ключ `null`, в то время как сколько угодно значений могут быть `null`.
- `TreeMap` — упорядоченная карта. Объекты в карте отсортированы в соответствии с их естественным порядком, если они реализуют интерфейс `Comparable`, либо в соответствии с `Comparator`, переданным конструктору `TreeMap`, если интерфейс `Comparable` в них не реализуется.

Пользователи Java, уже выработавшие собственный стиль, предпочитают объявлять типы интерфейсов, а не типы реализации, если это только возможно. Это общее правило, но его будет проще понять в контексте фреймворка коллекций.

Рассмотрим метод, возвращающий новый список строк, который почти не отличается от списка строк, переданного в качестве второго параметра, но в котором каждому элементу предшествует строка, переданная в качестве первого параметра. Код может выглядеть так:

```
public ArrayList<String> prefixList(
    String prefix,
    ArrayList<String> strs)
{
    ArrayList<String> ret
        = new ArrayList<String>(strs.size());
    for (String s: strs) { ret.add(prefix + s); }
    return ret;
}
```

Но такой вариант реализации нам не подойдет: он просто будет работать не со всеми видами списков! Он применим только к `ArrayList`. Если в какой-то момент понадобится изменить код, вызывающий этот метод, например перейти от использования `ArrayList` на `LinkedList`, код просто не сможет использовать прежний метод. Согласитесь, это совершенно неоправданно.

Более красивая реализация приведенного выше кода может выглядеть так:

```
public List<String> prefix(
    String prefix,
    List<String> strs)
{
    List<String> ret = new ArrayList<String>(strs.size());
    for (String s: strs) { ret.add(prefix + s); }
    return ret;
}
```

Эта версия более адаптируема, поскольку она не связывает метод с конкретной реализацией списка. Метод зависит только от того факта, что параметр реализует

определенный интерфейс. Неважно, как именно. При использовании типа интерфейса в качестве параметра мы сообщаем именно ту информацию, которая нужна для выполнения задачи, — ни больше ни меньше.

На самом деле код можно было бы оптимизировать и далее, если бы в качестве параметра и типа возврата мы имели коллекцию (Collection).

Дженерики в Java

Дженерики в Java — это обширная и по-настоящему сложная тема. На эту тему написаны целые книги. В данном пункте мы обсудим дженерики в контексте их наиболее распространенного набора — библиотеки Collections Library. Однако мы не ставим своей целью рассмотрение их в деталях.

Пока в языке Java не появились дженерики, было невозможно статически типизировать содержимое контейнера. Часто встречался код следующего вида:

```
public List makeList() {  
    // ...  
}  
  
public void useList(List l) {  
    Thing t = (Thing) l.get(0);  
    // ...  
}  
  
// ...  
useList(makeList());
```

Проблема очевидна: useList не гарантирует, что makeList создаст список элементов Thing. Компилятор не может удостовериться, что приведение в useList будет работать, поэтому код может «рвануть» прямо во время исполнения.

Дженерики позволяют решить эту проблему, правда, за счет существенной сложности. Синтаксис объявления дженериков устоялся довольно давно без сопровождающих код комментариев. Вот версия примера, в котором присутствуют дженерики:

```
public List<Thing> makeList() {  
    // ...  
}  
  
public void useList(List<Thing> l) {  
    Thing t = l.get(0);  
    // ...  
}  
  
// ...  
useList(makeList());
```

Тип объектов, заключенных в контейнере, указывается в угловых скобках (<>), которые являются частью типа контейнера. Обратите внимание на то, что в useList уже не требуется приведение, так как теперь компилятору известно, что параметр l — это список Thing.

Описания типов дженериков могут быть довольно развернутыми. Подобные объявления не редкость:

```
Map<UUID, Map<String, Thing>> cache  
= new HashMap<UUID, Map<String, Thing>>();
```

Сборка мусора

В языке Java организуется сборка мусора. Это означает, что код не управляет памятью. Вместо этого код создает новые объекты, выделяя память, а потом просто прекращает пользоваться объектами, в которых больше не нуждается. При необходимости среда времени исполнения Dalvik удаляет такие ненужные объекты и архивирует память.

Еще не так давно разработчикам доставляли беспокойство длительные и непредсказуемые периоды, в течение которых приложение не отвечало — сборщик мусора приостанавливал всю обработку приложения, чтобы восстановить память. Многие разработчики, и те, кто пользовался ранними версиями Java, и те, кто сравнительно недавно имел дело с J2ME, хорошо помнят трюки, уловки и неписанные правила, которые применялись, чтобы не возникало длительных пауз и фрагментации памяти — характерных неудобств, которые доставляли первые сборщики мусора. С тех пор технология сборки мусора претерпела немало изменений. В Dalvik упомянутые проблемы решительно отсутствуют. Создание новых объектов проходит практически без издержек. Лишь в немногих пользовательских интерфейсах, где делается особый упор на интерактивность — в частности, речь о некоторых играх, — все еще возникают паузы, обусловленные сборкой мусора.

Область видимости

Область видимости (scope) определяет, в каких частях программы видимы переменные, методы и другие символы. Вне области видимости символа этот символ недоступен и не может использоваться. В этом разделе мы поговорим об основных аспектах области видимости, начиная с наивысшего уровня.

Пакеты Java

Пакеты Java предоставляют механизм, позволяющий группировать взаимосвязанные типы в универсально-уникальном пространстве имен. Такое группирование не допускает конфликтов между идентификаторами, существующими в пространстве имен пакета, и идентификаторами, созданными и используемыми другими разработчиками в иных пространствах имен.

Типичная программа Java состоит из кода, взятого из множества деревьев пакетов (множество несмежных деревьев также называется forest — «лес»). Стандартная среда времени исполнения Java поддерживает такие пакеты, как `java.lang` и `java.util`. Кроме того, программа может зависеть от других распространенных библиотек,

например библиотек из дерева `org.apache`. По традиции код приложения (то есть создаваемый вами код) попадает в пакет, чье имя представляет собой запись названия вашего домена наоборот, к которой затем добавляется название программы. Следовательно, если ваш домен называется `androidhero.com`, то корень дерева пакетов будет называться `com.androidhero`, а сам код расположится в пакетах с названиями типа `com.androidhero.awesomeprogram` и `com.androidhero.geohottness.service`. Компоновка типичного пакета приложения для Android может содержать пакет для долговременного хранения информации, пакет для пользовательского интерфейса, а также пакет для логики приложения или код контроллера.

Пакеты не только предоставляют уникальное пространство имен, но и подразумевают определенные условия, касающиеся видимости членов (полей и методов) для других объектов, находящихся в том же пакете. Классы в одном и том же пакете могут видеть внутреннюю структуру друг друга, причем так, как эту структуру не могут просмотреть классы, расположенные вне пакета. Чуть ниже мы подробно обсудим эту тему.

Для объявления класса как входящего в состав пакета в первой строке файла, содержащего определение класса, ставится ключевое слово `package`:

```
package your.qualifieddomainname.functionalgroupping
```

Не пытайтесь сократить имя пакета! Поскольку сделанная наскоро временная реализация порой сохраняется годами, представьте себе, как надолго может стать вашей головной болью пакет, имя которого окажется неуникальным.

В некоторых крупных проектах используются совершенно разные домены первого уровня, позволяющие отделять друг от друга пакеты с общедоступными (публичными) API от пакетов, которые реализуют эти API. Например, интерфейс программирования приложений Android использует пакет верхнего уровня `android`, а классы реализации (implementation classes) обычно находятся в пакете `com.android`. Исходный код языка Java от компании Sun организован по схожему принципу. Общедоступные интерфейсы программирования приложений располагаются в пакете `java`, а код реализации — в пакете `sun`. В любом случае приложение, импортирующее пакет реализации, явно делает что-то ненадежное и попадает в зависимость от кода, не относящегося к общедоступным API.

В то время как, в принципе, существует возможность добавлять код в уже имеющиеся пакеты, это обычно считается порочной практикой. Вообще, пакет является не только пространством имен, но и деревом, идущим от одного источника как минимум до обратного доменного имени. Это всего лишь неписаное правило, но разработчик Java считает, что в источнике пакета `com.brashandroid.coolapp.ui` он найдет код пользовательского интерфейса приложения CoolApp для Android. И разработчик будет неприятно удивлен, если ему придется искать вторую часть источников в другом пакете.



Во фреймворке приложений Android также присутствует концепция пакета (Package). Она отличается от описанной здесь, и о ней мы поговорим в главе 3. Не путайте эту концепцию с пакетами Java.

О пакетах Java подробнее рассказано в руководстве по этому языку: <http://download.oracle.com/javase/tutorial/java/package/packages.html>.

Модификаторы доступа и инкапсуляция

Ранее мы упоминали о том, что к членам класса применимы специальные правила видимости. Определения в большинстве блоков Java имеют статическую (лексическую) область видимости: они видимы только внутри данного блока и вложенных в него блоков. Но определения, сделанные в классе, могут быть видимы и вне блока. Java поддерживает публикацию членов класса, относящихся к верхнему уровню, — то есть его методов и полей — так, чтобы к ним имел доступ код из других классов. Этот доступ осуществляется при помощи модификаторов доступа. Модификаторы доступа — это ключевые слова, влияющие на видимость объявлений, к которым применяются эти модификаторы, для тех или иных элементов.

В языке Java существует три ключевых слова, которые выступают в качестве модификаторов доступа: `public`, `protected` и `private`. Вместе они поддерживают четыре уровня доступа. Понятно, что модификаторы доступа связаны с видимостью объявления из-за пределов класса, в котором это объявление содержится. Внутри же этого класса действуют нормальные правила обзора данных в блоке, независимо от конкретной модификации доступа.

Модификатор доступа `private` накладывает наибольшее количество ограничений. Объявление с модификатором доступа `private` невидимо за пределами блока, в котором оно содержится. Это наиболее безопасный способ объявления, поскольку он гарантирует, что любые ссылки, которые будут делаться на объявление, будут идти только из объемлющего класса (того, в котором находится объявление). Чем больше объявлений `private` в классе, тем более безопасен этот класс.

Следующий по строгости уровень ограничений при доступе называется доступом к пакету или стандартным доступом. Доступ к объявлениям, к которым не применяется какой-либо модификатор доступа, выполняется по умолчанию, то есть они видимы только для других классов того же пакета. Доступ по умолчанию очень удобен для того, чтобы создавать состояние, совместно используемое несколькими объектами. Подобным образом действует объявление `friend` в C++.

Модификатор доступа `protected` предоставляет все те же права, что и доступ по умолчанию, но, кроме того, разрешает и доступ из любого подтипа. Любой класс, который дополняет другой класс с объявлениями `protected`, имеет доступ к этим объявлениям.

Наконец, `public` — это слабейший модификатор, разрешающий доступ откуда угодно.

Ниже приводится пример кода, позволяющий лучше познакомиться с модификаторами. Здесь мы видим четыре класса в двух разных пакетах, все они ссылаются на поля, объявленные в одном из классов — `Accessible`:

```
package over.here;
```

```
public class Accessible {  
    private String localAccess;  
    String packageAccess;  
    protected String subtypeAccess;  
    public String allAccess;  
  
    public void test() {
```

```

        // Все присваивания ниже работают:
        // поля объявляются в вышестоящем
        // блоке и поэтому видимы.
        localAccess = "success!!";
        packageAccess = "success!!";
        subtypeAccess = "success!!";
        allAccess = "success!!";
    }
}

package over.here;
import over.here.Accessible;

// этот класс находится в том же пакете, что и Accessible
public class AccessibleFriend {

    public void test() {
        Accessible target = new Accessible();

        // private-члены невидимы
        // вне того класса, в котором происходит объявление
        target.localAccess = "fail!!"; // ОШИБКА!!

        // доступ по умолчанию — объявление видимо внутри пакета
        target.packageAccess = "success!!";

        // защищенный (protected) доступ — это расширенный
        // вариант стандартного доступа
        target.subtypeAccess = "success!!";

        // видимо везде
        target.allAccess = "success!!";
    }
}

package over.there;
import over.here.Accessible;

// подтип Accessible
// в другом пакете
public class AccessibleChild extends Accessible {

    // видимые поля из Accessible кажутся такими,
    // как будто они объявлены в охватывающем блоке
    public void test() {
        localAccess = "fail!!"; // ОШИБКА!!
        packageAccess = "fail!!"; // ОШИБКА!!

        // защищенные (private) объявления
        // видимы из подтипов
        subtypeAccess = "success!!";
    }
}

```

```

        // видимо везде
        allAccess = "success!!";
    }
}

package over.there;
import over.here.Accessible;

// класс, совершенно не относящийся к Accessible
public class AccessibleStranger {

    public void test() {
        Accessible target = new Accessible();
        target.localAccess = "fail!!"; // ERROR!!
        target.packageAccess = "fail!!"; // ERROR!!
        target.subtypeAccess = "success!!"; // ERROR!!

        // видимо везде
        target.allAccess = "success!!";
    }
}

```

Идиомы программирования в Java

Проблема идиоматического использования языка располагается где-то между правильным пониманием специфики синтаксиса того или иного языка программирования и хорошим паттерн-ориентированным проектированием (которое не зависит от языка). Программист, применяющий идиоматический подход, задействует для выражения схожих идей единообразный код. Поступая так, он создает программы, которые легко понятны, оптимально используют среду времени исполнения, а также позволяют избегать сбоев, от которых не застрахован синтаксис любого языка.

Безопасность типов в Java

Основной целью при проектировании Java было создание языка, который обеспечил бы безопасное программирование. «Пространность» и негибкость Java, которую не клеймил только самый ленивый и которая действительно отсутствует в таких языках, как Python, Ruby и Objective-C, нужна для того, чтобы компилятор мог гарантировать: во время исполнения не могут возникать целые классы ошибок.

Применяемая в Java статическая типизация оказалась очень полезной далеко за пределами собственного компилятора Java. Способность машины к синтаксическому разбору и пониманию семантики кода Java стала основной движущей силой при разработке таких мощных инструментов, как FindBugs и средства рефакторинга интегрированной среды разработки.

Многие разработчики отстаивают точку зрения, что, особенно при работе с современными инструментами программирования, эти ограничения — не слишком

дорогая цена за возможность немедленного обнаружения проблем, которые в иных случаях могут проявиться лишь уже на этапе развертывания кода. Разумеется, не менее велика и когорта программистов, которые доказывают, что они экономят такую массу времени, работая с динамическими языками, что вполне успевают писать, кроме кода, и всеобъемлющие тесты компонентов, и интеграционные тесты, что тоже позволяет избежать массы проблем.

Независимо от того, какая из этих точек зрения вам ближе, совершенно очевидно, что доступные инструменты нужно использовать с максимальной эффективностью. Статическое связывание, применяемое в Java, **вне всякого сомнения, ограничивает** возможности разработчика. С другой стороны, Java — это очень хороший язык со статическим связыванием, хотя с динамикой у Java и правда неважно. На самом деле в Java можно выполнять и по-настоящему динамические операции, используя содержащиеся в этом языке API рефлексии и интроспекции, а также активно прибегая к приведению типов. Но если работать таким образом, то за исключением редчайших случаев язык и среда времени исполнения будут использоваться для достижения противоречащих друг другу целей. Ваша программа, вероятно, будет работать очень медленно, и тулчейн Android (набор инструментов для компиляции) может совершенно не разобраться в вашем коде. Самое важное замечание заключается, пожалуй, в том, что, если в этой редко используемой части платформы найдутся ошибки (баги), вы узнаете о них в числе первых. Советуем примириться со статической природой Java (по крайней мере, пока не появится хорошей динамической альтернативы) и в полной мере воспользоваться преимуществами этого языка.

Инкапсуляция

Разработчики часто ограничивают видимость членов объекта с целью создания инкапсуляции. Смысл инкапсуляции заключается в том, что объект ни в коем случае не должен сообщать о себе такие детали, поддержка которых в нем не предусмотрена. Возвращаясь к нашему примеру с мохито, напоминаем, что вас совершенно не волнует, как коллега достал нужную вам мяту. Но, предположим, вы ему говорите: «А не мог бы ты нарвать в саду еще мяты? И, раз уж ты там будешь, полей заодно розовый куст». Итак, мы уже не можем сказать, что вас не волнует, откуда коллега берет мяту. Теперь вас интересует, как именно он ее добывает.

Аналогично интерфейс объекта (коротко называемый API) состоит из методов и типов, доступных для вызывающего кода. Аккуратно применяя инкапсуляцию, разработчик скрывает детали реализации объекта от того кода, который использует этот объект. Такой контроль и защита позволяют создавать программы, которые отличаются значительной гибкостью и позволяют разработчику объекта со временем изменять реализацию объекта, не вызывая при этом «эффекта ряби» (как от камешка, брошенного в воду) в вызывающем коде.

Получатели и установщики

Простая и при этом распространенная форма инкапсуляции в Java связана с использованием методов-получателей (**getter**) и методов-установщиков (**setter**). Рассмотрим упрощенное определение класса `Contact`:

```
public class Contact {  
    public String name;  
    public int age;  
    public String email;  
}
```

При таком определении внешние объекты должны получать доступ к полям класса напрямую. Например:

```
Contact c = new Contact();  
c.name = "Alice";  
c.age = 13;  
c.email = "alice@mymail.com";
```

Попользовавшись таким кодом на практике самую малость, вы обнаружите, что у контактов, оказывается, бывает по несколько адресов электронной почты. К сожалению, для реализации многоадресности в упрощенной реализации кода потребуется обновить каждую отдельную ссылку так, чтобы она приобрела вид `Contact.email`, и так во всей программе.

Для сравнения рассмотрим следующий класс:

```
class Contact {  
    private int age;  
    private String name;  
    private String email;  
  
    Contact(int age, String name, String email) {  
        this.age = age;  
        this.name = name;  
        this.email = email;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getEmail() {  
        return address;  
    }  
}
```

Применяя модификатор доступа `private`, мы закрываем непосредственный доступ к полям данной версии класса `Contact`. Применяя общедоступные (`public`) методы-получатели, разработчик может изменять способы, которыми объект `Contact` возвращает имя, возраст или адрес электронной почты, относящийся к `Contact`. Например, адрес электронной почты может сохраняться отдельно (как в предыдущем коде) или составляться из соединенных имени пользователя и хост-имени, если в конкретном приложении такой вариант окажется более удобным. Внутри

системы возраст может сохраняться в виде `int` или `Integer`. Класс можно дополнить для поддержки множественных адресов электронной почты, совершенно не изменяя какой-либо клиент.

Java позволяет использовать прямые ссылки на поля и не допускает, подобно некоторым другим языкам, автоматического обертывания ссылок на поля в методах-получателях и методах-установщиках. Для сохранения инкапсуляции необходимо самостоятельно определять каждый метод доступа. В большинстве интегрированных сред разработки предоставляются функции генерирования кода, решающие такие задачи быстро и точно.

Методы-получатели и методы-установщики, используемые в качестве оберток, обеспечивают гибкость кода на дальнейших этапах работы. В свою очередь, прямой доступ к полям означает, что весь код, использующий поле, придется менять, если изменится тип этого поля или если поле исчезнет. Методы-получатели и методы-установщики — это простые инструменты, обеспечивающие инкапсуляцию объекта. Есть отличное практическое правило, рекомендующее делать все поля либо `private`, либо `final`. В хорошо написанных программах Java применяется не только такая форма, но и другие более изощренные формы инкапсуляции, помогающие сохранять способность к адаптации даже в сравнительно сложных программах.

Использование анонимных классов

Специалисты, имеющие опыт разработки пользовательских интерфейсов, знакомы с концепцией обратного вызова (**callback**): код должен получить уведомление, когда в пользовательском интерфейсе произойдет какое-либо изменение. Примеры таких изменений — нажатие кнопки или переход модели в какое-то новое состояние. Может быть, из сети пришли новые данные и их нужно отобразить. Вам понадобится способ добавления во фреймворк блока кода, который впоследствии будет выполняться по вашему требованию.

Хотя в языке Java и присутствует идиома для передачи блоков кода, она немного неудобна, так как ни блоки кода, ни методы, не являются в языке объектами первого класса. В языке не предусмотрен способ получения ссылки ни на первые, ни на вторые.

Ссылка может указывать на экземпляр класса. Например, в Java можно передавать не блоки или функции, а целый класс, определяющий нужный вам код в качестве одного из своих методов. Сервис, предоставляющий API обратного вызова, определит его протокол при помощи интерфейса. Клиент сервиса определяет реализацию этого интерфейса и передает информацию во фреймворк.

Рассмотрим, например, как в Android реализуется механизм отклика на нажатие клавиши пользователем. Класс `Android View` определяет интерфейс `OnKeyListener`, который, в свою очередь, определяет метод `onKey`. Если код передает `View` реализацию `OnKeyListener`, то метод `onKey` будет вызываться всякий раз, когда класс `View` будет обрабатывать новое событие, связанное с нажатием клавиш.

Код может иметь следующий вид:

```
public class MyDataModel {  
    // класс обратного вызова  
    private class KeyHandler implements View.OnKeyListener {
```

```

        public boolean onKey(View v, int keyCode, KeyEvent event) {
            handleKey(v, keyCode, event)
        }
    }

    /** @param view для вида, который мы моделируем */
    public MyDataModel(View view) {view.setOnKeyListener(new KeyHandler())}

    /** обработка события, связанного с нажатием клавиши */
    void handleKey(View v, int keyCode, KeyEvent event) {
        // здесь идет код, занятый обработкой нажатия клавиши...
    }
}

```

При создании нового экземпляра класса `MyDataModel` при помощи аргумента `view` конструктору сообщается, к какому виду прикрепляется этот класс. Конструктор создает новый экземпляр тривиального класса обратного вызова, `KeyHandler`, и устанавливает его в вид. Все последующие события, связанные с клавишами, будут передаваться методу `handleKey`, который относится к экземпляру модели.

Поставленная задача, таким образом, безусловно, решается, однако решение может получиться довольно некрасивым, особенно если классу придется обрабатывать разнохарактерные события, поступающие от разнообразных видов! Через какое-то время все эти определения типов совершенно засорят программу. Определение может быть расположено довольно далеко от использования элемента, а если задуматься — зачем эти определения вообще нужны?

В Java такой код можно несколько упростить, воспользовавшись анонимным классом. Вот фрагмент кода, подобный показанному выше. Но он реализуется с применением анонимного класса:

```

public class MyDataModel {
    /** @param view для вида, который мы моделируем */
    public MyDataModel(View view) {
        view.setOnKeyListener(
            // Это анонимный класс!!
            new View.OnKeyListener() {
                public boolean onKey(View v, int keyCode, KeyEvent event) {
                    handleKey(v, keyCode, event)
                } } );
    }

    /** обработка события, связанного с нажатием клавиши */
    void handleKey(View v, int keyCode, KeyEvent event) {
        // здесь идет код, занятый обработкой нажатия клавиши...
    }
}

```

Конечно, на синтаксический разбор такого кода может понадобиться некоторое время, но он практически идентичен коду из предыдущего примера. Он передает новоиспеченный экземпляр подтипа `View.OnKeyListener` в качестве аргумента, это делается при вызове `view.setOnKeyListener`. Но в данном случае аргумент к вызову `view.setOnKeyListener` использует особый синтаксис, определяющий новый подкласс

интерфейса `View.OnKeyListener` и инстанцирующий его за одну операцию. Новый экземпляр — это экземпляр класса без имени, то есть анонимного класса. Его определение существует только в инстанцирующем его утверждении.

Анонимные классы очень удобны и используются в Java в качестве идиомы для выражения многих разновидностей блоков кода. Объекты, создаваемые с применением анонимных классов, являются в языке объектами «первого класса» и допустимы везде, где мог бы использоваться объект того же типа. Например, возможно такое присваивание:

```
public class MyDataModel {
    /** @param view для вида, который мы моделируем */
    public MyDataModel(View view1, View view2) {
        // получение ссылки на анонимный класс
        View.OnKeyListener keyHdlr = new View.OnKeyListener() {
            public boolean onKey(View v, int keyCode, KeyEvent event) {
                handleKey(v, keyCode, event)
            } };

        // использование класса для обмена информацией с двумя видами
        view1.setOnKeyListener(keyHdlr);
        view2.setOnKeyListener(keyHdlr);
    }

    /** обработка события, связанного с нажатием клавиши */
    void handleKey(View v, int keyCode, KeyEvent event) {
        // здесь идет код, занятый обработкой нажатия клавиши...
    }
}
```

Может возникнуть вопрос: почему анонимный класс в этом примере делегирует свою фактическую реализацию (метод `handleKey`) объемлющему классу? На самом деле не существует правила, которое бы как-то регламентировало содержимое анонимного класса: он вполне может содержать и полную реализацию. С другой стороны, хорошее понимание идиом подсказывает, что код, изменяющий состояние объекта, следует поместить в класс объекта. Если реализация содержится в объемлющем классе, ее могут использовать другие методы и обратные вызовы. Анонимный класс — просто посредник, в этом и заключается его функция.

В Java действительно присутствуют по-настоящему строгие ограничения, касающиеся использования в анонимном классе переменных, которые находятся в области видимости (то есть всего, что определено в вышестоящем блоке). В частности, анонимный класс может ссылаться на переменную, унаследованную из окружающей области видимости, лишь в том случае, если эта переменная объявлена как `final`. Например, скомпилировать следующий фрагмент кода не удастся:

```
/** создание обработчика нажатий клавиш, подставляющего информацию о нажатой клавише */
public View.OnKeyListener curry(int keyToMatch) {
    return new View.OnKeyListener() {
        public boolean onKey(View v, int keyCode, KeyEvent event) {
            if (keyToMatch == keyCode) { foundMatch(); } // ОШИБКА!!
        } };
}
```

Проблема снимается, если объявить аргумент к `curry` как `final`. Когда мы объявляем его как `final`, это, конечно же, означает, что его нельзя изменить в анонимном классе. Но для преодоления этой проблемы есть простой идиоматический способ:

```
/** создание обработчика нажатий клавиш, выполняющего приращение и подставляющего
информацию о нажатой клавише */
public View.OnKeyListener curry(final int keyToMatch) {
    return new View.OnKeyListener() {
        private int matchTarget = keyToMatch;
        public boolean onKey(View v, int keyCode, KeyEvent event) {
            matchTarget++;
            if (matchTarget == keyCode) { foundMatch(); }
        }
    };
}
```

Модульное программирование в Java

Несомненно, возможность расширения классов в Java обеспечивает разработчикам существенную гибкость при переопределении различных аспектов объектов, в зависимости от того, в каком контексте объект используется. Однако требуется немалый практический опыт, чтобы научиться рационально использовать классы и интерфейсы. В идеале разработчик стремится создавать такие фрагменты кода, в которые будет несложно вносить изменения с течением времени и которые можно будет многократно использовать в максимально разнообразных контекстах, в многочисленных приложениях и, возможно, даже в качестве библиотек. При применении такого подхода к программированию удастся значительно снизить количество ошибок и время, необходимое для вывода приложения на рынок. Модульное программирование, инкапсуляция и разделение функций — основные стратегии, обеспечивающие стабильность кода и пригодность его к многократному использованию.

В объектно-ориентированном программировании существует фундаментальный принцип проектирования, который заключается в том, что уже существующий код используется при написании нового кода путем делегирования или наследования. В следующих примерах представлены различные иерархии объектов, в которых перечислены транспортные средства — этот транспорт может присутствовать, например, в симуляторе автогонок. В каждом примере избран собственный вариант модульности.

Разработчик начинает с создания класса транспортных средств, в котором содержится вся транспортная логика и вся логика для конкретного типа двигателя, вот так:

```
// Упрощенный код!
public class MonolithicVehicle {
    private int vehicleType;

    // поля для электродвигателя
    // поля для бензинового двигателя
    // поля для гибридного двигателя
```

```
// поля для парового двигателя

public MonolithicVehicle(int vehicleType) {
    vehicleType = vehicleType;
}

// другие методы для реализации транспортных средств
// и разнотипных двигателей

void start() {
    // код для электродвигателя
    // код для бензинового двигателя
    // код для гибридного двигателя
    // код для парового двигателя
}
}
```

Это упрощенный код. Он, возможно, и будет работать, но в нем перемешаны не связанные друг с другом типы реализации (например, все типы двигателей). Поэтому код будет сложно расширять. Рассмотрим, например, случай, в котором требуется изменить реализацию и добавить в нее новый тип двигателя (ядерный). Код для каждой разновидности автомобильного двигателя имеет неограниченный доступ к коду любого другого двигателя. Ошибка в реализации двигателя может спровоцировать ошибку в совсем другом двигателе. Изменение одного двигателя может привести к неожиданным изменениям в другом. И, конечно же, машине, в которой нет электрического двигателя, придется тем не менее тащить с собой представления всех существующих в системе типов двигателей. Разработчики, которым в перспективе придется реализовать для этой программы монолитный двигатель, должны понимать все сложные взаимодействия, которые могут возникнуть при изменении кода. Некоторые категории просто несравнимы.

Как можно было бы улучшить эту реализацию? Очевидная идея — использовать подклассы. Например, иерархию классов, представленную в следующем коде, можно применить для реализации различных типов транспортных средств, каждое из которых будет жестко связано с типом двигателя:

```
public abstract class TightlyBoundVehicle {
    // поле двигателя отсутствует

    // Каждый подкласс должен переопределять этот метод, чтобы реализовать
    // собственный способ запуска двигателя той или иной машины.
    protected abstract void startEngine();

    public final void start() { startEngine(); }
}

public class ElectricVehicle extends TightlyBoundVehicle {
    protected void startEngine() {
        // реализация электрического двигателя
    }
}

public class GasVehicle extends TightlyBoundVehicle {
```

```

        protected void startEngine() {
            // реализация бензинового двигателя
        }
    }

    public void anInstantiatingMethod() {
        TightlyBoundVehicle vehicle = new ElectricVehicle();
        TightlyBoundVehicle vehicle = new GasVehicle();
        TightlyBoundVehicle vehicle = new HybridVehicle();
        TightlyBoundVehicle vehicle = new SteamVehicle();
    }

```

Очевидно, код стал лучше. Теперь код для двигателя каждого типа инкапсулирован в собственном классе и не может влиять на код, связанный с другими двигателями. Можно расширять код для отдельных типов транспортных средств, не затрагивая какие-либо другие типы. Во многих случаях такая реализация является идеальной.

С другой стороны, что произойдет, если вы захотите перевести вашу машину, жестко связанную с бензиновым двигателем, на биотопливо? В этой реализации машина и двигатель — это один и тот же объект. Разделить их невозможно. В реалистичной ситуации, которую мы моделируем, эти объекты нужно рассматривать в отдельности и в архитектуре приложения должно быть более слабое связывание:

```

interface Engine {
    void start();
}

class GasEngine implements Engine {
    void start() {
        // свеча зажигает бензин
    }
}

class ElectricEngine implements Engine {
    void start() {
        // ток подается в батарею
    }
}

class DelegatingVehicle {
    // есть поле двигателя
    private Engine mEngine;

    public DelegatingVehicle(Engine engine) {
        mEngine = engine;
    }

    public void start() {
        // делегирующее транспортное средство может использовать
        // бензиновый или электрический двигатель
    }
}

```

```

        mEngine.start();
    }
}

void anInstantiatingMethod() {
    // Новые типы транспортных средств создаются с легкостью.
    // Для этого нужно просто подключать новые двигатели.
    DelegatingVehicle electricVehicle =
        new DelegatingVehicle(new ElectricEngine());
    DelegatingVehicle gasVehicle = new DelegatingVehicle(new GasEngine());
    // DelegatingVehicle hybridVehicle =
    // new DelegatingVehicle(new HybridEngine());
    // DelegatingVehicle steamVehicle =
    // new DelegatingVehicle(new SteamEngine());
}

```

В приведенной архитектуре класс транспортных средств делегирует все поведения, связанные с двигателем, тому объекту-двигателю, которым он обладает. Такой подход иногда называется «иметь», в противоположность предыдущему примеру с подклассами, который называется «быть». Код может проявлять и еще более значительную гибкость, так как он отделяет знания о том, как именно работает двигатель, от информации о машине, в которой этот двигатель установлен. Каждый двигатель делегирует информацию к слабо связанному типу двигателя и не знает, как этот двигатель реализует свое поведение. В предыдущем примере мы применили многоразовый класс `DelegatingVehicle`, который совершенно не изменяется, когда мы переходим к работе с новым типом двигателя. Транспортное средство может использовать любую реализацию интерфейса `Engine`. Кроме того, появляется возможность создавать новые типы транспортных средств (например, спортивный, компактный или роскошный автомобиль) и для каждого из них использовать свой тип `Engine`.

Применяя делегирование, мы минимизируем взаимозависимость между двумя объектами и обеспечиваем максимальную гибкость при их изменении, если такое изменение потребуется. Предпочитая делегирование наследованию, разработчик облегчает расширение и оптимизацию кода. При использовании интерфейсов для определения контракта между объектом и его делегатами разработчик гарантирует, что делегаты будут вести себя именно так, как от них ожидается.

Основы многопоточного параллельного программирования в Java

В языке Java поддерживаются параллельные (конкурентные) потоки выполнения задач. Утверждения, содержащиеся в различных потоках, выполняются в запрограммированном порядке, но порядковые отношения отсутствуют между утверждениями, находящимися в различных потоках. Базовый элемент параллельного исполнения в Java заключен в классе `java.lang.Thread`. При рекомендуемом методе

порождения потоков используется реализация интерфейса `java.lang.Runnable`, как показано в следующем примере:

```
// программа, поочередно обрабатывающая сообщения из двух потоков
public class ConcurrentTask implements Runnable {
    public void run() {
        while (true) {
            System.out.println("Message from spawned thread");
        }
    }
}

public void spawnThread() {
    (new Thread(new ConcurrentTask())).start();

    while (true) {
        System.out.println("Message from main thread");
    }
}
```

В предыдущем примере метод `spawnThread` создает новый поток, передавая новый экземпляр `ConcurrentTask` конструктору потока. Затем метод вызывает `start` к новому потоку. После вызова метода `start`, относящегося к потоку, базовая виртуальная машина (VM) создает новый параллельный поток исполнения, который, в свою очередь, вызовет метод `run` переданного `Runnable`, выполняя эту задачу одновременно с порождением потока. В этот момент в виртуальной машине происходит два независимых процесса: порядок исполнения и хронометраж в одном потоке никак не связаны с порядком и хронометражем другого.

Класс `Thread` не является `final`. Можно определить новую параллельную задачу, сделав подкласс `Thread` и переопределив его метод `run`. Но этот подход не отличается особыми преимуществами. На самом деле `Runnable` больше пригоден к адаптации. Поскольку `Runnable` — это интерфейс, тот `Runnable`, который вы передаете конструктору `Thread`, может дополнить какой-нибудь другой полезный класс.

Синхронизация и потоковая безопасность

Когда два или более работающих потока имеют доступ к одному и тому же набору переменных, потоки могут изменять эти переменные, причем так, что возможно повреждение данных или нарушение логики одного или нескольких из этих потоков. Подобные ненамеренные ошибки конкурентного доступа называются *нарушениями безопасности потоков* (**thread safety violations**). Их сложно воспроизводить, сложно находить и сложно тестировать.

В Java прямо не прописаны ограничения на доступ нескольких потоков к переменным. Вместо этого в качестве основного механизма обеспечения безопасности потоков в Java используется ключевое слово `synchronized`. Это ключевое слово сериализует доступ к тому блоку, который контролирует и, что еще важнее, синхронизирует видимое состояние между двумя потоками. Рассуждая о параллельности

в Java, легко упустить из виду, что синхронизация одновременно управляет и доступом, и видимостью. Рассмотрим следующую программу:

```
// Этот код серьезно поврежден!!!
public class BrokenVisibility {
    public static boolean shouldStop;

    public static void main(String[] args) {
        new Thread(
            new Runnable() {
                @Override public void run() {
                    // этот код выполняется в порожденном потоке
                    final long stopTime
                        = System.currentTimeMillis() + 1000;
                    for (;;) {
                        shouldStop
                            = System.currentTimeMillis() > stopTime;
                    }
                }
            }
        ).start();

        // этот код выполняется в основном потоке
        for (;;) {
            if (shouldStop) { System.exit(0); }
        }
    }
}
```

Можно подумать: «Нет никакой необходимости синхронизировать переменную `shouldStop`. Действительно, при доступе к ней между основным и порожденным потоком может возникнуть конфликт. И что? Через секунду порожденный поток обязательно установит значение `true`. Булевы выражения являются атомарными. Если основной поток на данный момент не увидит, что порожденный поток имеет значение `true`, то, несомненно, он увидит порожденный поток в значении `true` в следующий раз». Эта логика не только ложная, но и опасная. Она не учитывает действия оптимизирующих компиляторов и кэширующих процессоров! Программа может никогда не закончиться. Каждый из двух потоков вполне может использовать собственную копию `shouldStop`, существующую только в аппаратном кэше какого-нибудь локального процессора. Поскольку синхронизация между двумя потоками отсутствует, копия кэша, возможно, так и не будет предоставлена в общий доступ и основной поток никогда не увидит значения порожденного потока.

В Java действует простое правило, позволяющее избежать нарушения безопасности потоков: когда два различных потока получают доступ к одному и тому же изменяемому состоянию (переменной), *весь* доступ к этому состоянию должен осуществляться синхронно, с общей блокировкой.

Некоторые разработчики нарушают это правило, поразмыслив о поведении совместно используемого состояния в своей программе, пытаясь оптимизировать код. Поскольку многие устройства, на которых сегодня используется платформа Android, не допускают параллельного исполнения (вместо этого один процессор поочередно

используется всеми потоками), возможно, создастся впечатление, что программы, в которых нарушается описанное правило, работают правильно. Тем не менее, когда мобильные устройства рано или поздно будут оснащены несколькими многоядерными процессорами и большими многоуровневыми кэшами процессоров, неправильно написанные программы, скорее всего, будут страдать от ошибок — серьезных, периодических, которые к тому же будет очень сложно находить.

При реализации параллельных процессов в Java лучше всего пользоваться мощными библиотеками `java.util.concurrent`. Здесь найдутся практически любые конкурентные структуры, которые могут потребоваться, — оптимально реализованные и протестированные. В Java разработчикам уже почти не приходится прибегать к низкоуровневым конкурентным конструкциям, так как это не более целесообразно, чем изобретать собственную реализацию двунаправленного списка.

Ключевое слово `synchronized` может применяться в трех контекстах: для создания блока, для работы со статическим методом или для работы с динамическим методом. При определении блока это ключевое слово принимает в качестве аргумента ссылку на объект, который будет использоваться в качестве семафора. Прimitives типы не могут применяться в качестве семафоров, а любой объект — может.

При использовании этого ключевого слова в качестве модификатора динамического метода оно ведет себя так, как если бы содержимое метода было обернуто в синхронизированный блок, который применял бы сам экземпляр в целях блокировки. Это показано на следующем примере:

```
class SynchronizationExample {

    public synchronized void aSynchronizedMethod() {
        // Поток, в котором выполняется этот метод, блокирует
        // "this". Любой другой поток, пытающийся использовать
        // this, или любой другой метод, синхронизированный
        // с "this", будут поставлены в очередь, до тех пор
        // пока этот поток не отменит блокировку.
    }

    public void equivalentSynchronization() {
        synchronized (this) {
            // Это равноценно использованию
            // ключевого слова synchronized в методе def.
        }
    }

    private Object lock = new Object();

    public void containsSynchronizedBlock() {
        synchronized (lock) {
            // Поток, в котором выполняется этот метод,
            // блокирует "lock", а не "this".
            // Потоки, пытающиеся захватить "this",
            // могут выполняться успешно. Только те, которые пытаются
            // захватить "lock", будут заблокированы.
        }
    }
}
```

Этот способ очень удобен, но пользоваться им нужно с осторожностью. Сложный класс, содержащий множество активно применяемых методов и синхронизирующий их таким образом, может спровоцировать конфликт при блокировках. Если несколько внешних потоков попытаются одновременно обратиться к фрагментам данных, которые не связаны друг с другом, то лучше всего защитить эти информационные фрагменты отдельными блокировками.

Если ключевое слово `synchronized` применяется к статическому методу, мы сталкиваемся с ситуацией, когда содержимое метода словно обернуто в блок, синхронизированный с классом объекта. Все статические синхронизированные методы для всех экземпляров данного класса будут конфликтовать из-за общей блокировки, используемой для самого объекта класса.

Наконец, следует отметить, что объектные блокировки в Java являются много-разовыми (*reentrant*). Следующий код совершенно безопасен и не вызывает взаимоблокировки (клинча):

```
class SafeSeizure {
    private Object lock = new Object();

    public void method1() {
        synchronized (lock) {
            // делаем то, что надо
            method2();
        }
    }

    public void method2() {
        synchronized (lock) {
            // делаем то, что надо
        }
    }
}
```

Управление потоками с помощью методов `wait()` и `notify()`

Класс `java.lang.Object` определяет методы `wait()` и `notify()` в рамках протокола блокировки, входящего в состав каждого объекта. Поскольку все классы в Java дополняют `Object`, все экземпляры объектов поддерживают эти методы управления блокировкой, которая связана с экземпляром.

Подробное обсуждение инструментов многопоточного исполнения в Java не является темой этой книги. Рассмотрим следующий пример, который демонстрирует важнейший элемент, необходимый для совместной работы двух потоков. Один поток приостанавливается, пока второй поток выполняет задачу, нужную для продолжения работы:

```
/**
 * Задача, медленно заполняющая список и уведомляющая
 * блокировку, стоящую на "this", по окончании работы.
```

```

* Заполнение списка безопасно для потоков.
*/
public class FillListTask implements Runnable {
    private final int size;
    private List<String> strings;

    public FillListTask(int size) {
        this.size = size;
    }

    public synchronized boolean isFinished() {
        return null != strings;
    }

    public synchronized List<String> getList() {
        return strings;
    }

    @Override
    public void run() {
        List<String> str = new ArrayList<String>(size);
        try {
            for (int i = 0; i < size; i++ ) {
                Thread.sleep(2000);
                str.add("element " + String.valueOf(i));
            }

            synchronized (this) {
                strings = str;
                this.notifyAll();
            }
        }
        catch (InterruptedException e) {
            // Перехват прерванного исключения вне цикла,
            // поскольку прерванное исключение является сигналом того,
            // что поток должен завершить работу.
        }
    }

    /**
    * Ожидает, пока задача заполнения списка будет завершена.
    */
    public static void main(String[] args)
        throws InterruptedException
    {
        FillListTask task = new FillListTask(7);

        new Thread(task).start();

        // Вызов wait() снимает блокировку

```

```

// задачи и приостанавливает работу потока до тех пор.
// пока он не получит уведомления.
synchronized (task) {
    while (!task.isFinished()) {
        task.wait();
    }
}

System.out.println("Array full: " + task.getList());
}
}

```

Большинство разработчиков обычно не пользуются такими низкоуровневыми инструментами, как `wait` и `notify`, а обращаются к пакету `java.util.concurrent`, содержащему более высокоуровневые инструменты.

Синхронизация и структуры данных

Android поддерживает функционально насыщенную библиотеку коллекций Java (**Java Collection Library**) для стандартной версии Java (**Standard Edition Java**). Внимательно ознакомившись с этой библиотекой, вы заметите, что большинство коллекций существует в двух вариантах: `List` и `Vector`, `HashMap` и `Hashtable` и т. д. В версии Java 1.3 появляется совершенно новый фреймворк коллекций, который полностью замещает старые коллекции. Однако для обеспечения обратной совместимости старые версии не выходят из употребления.

Тем не менее новым коллекциям следует отдавать предпочтение по сравнению с их устаревшими аналогами. Они обладают более унифицированным интерфейсом программирования приложений, для их поддержки существуют более удобные инструменты и т. д. Но важнее всего, пожалуй, то, что все устаревшие коллекции синхронизированы. Может показаться, что идея синхронизации очень стоящая, но, как показано в следующем примере, ее может быть недостаточно:

```

public class SharedListTask implements Runnable {
    private final Vector<String> list;

    public SharedListTask(Vector<String> l) {
        this.list = l;
    }

    @Override
    public void run() {
        // размер списка получен слишком рано
        int s = list.size();

        while (true) {
            for (int i = 0; i < s; i++ ) {
                // Выбрасывает IndexOutOfBoundsException!!
                // Когда размер списка равен 3, а s равно 4.
            }
        }
    }
}

```

```
        System.out.println(list.get(i));
    }
}

public static void main(String[] args) {
    Vector<String> list = new Vector<String>();
    list.add("one");
    list.add("two");
    list.add("three");
    list.add("four");

    new Thread(new SharedListTask(list)).start();

    try { Thread.sleep(2000); }
    catch (InterruptedException e) { /* игнорировать */ }

    // Структура данных полностью синхронизирована,
    // но это защищает только методы по отдельности!
    list.remove("three");
}
```

Хотя все случаи использования `Vector` полностью синхронизированы и каждый вызов к одному из его методов гарантированно будет атомарным, программа не выполнится. Полная синхронизация `Vector`, разумеется, является недостаточной, поскольку код делает копию его размера и использует ее даже после того, как другой поток может изменить этот размер.

Поскольку сама по себе синхронизация методов объекта коллекции так часто является недостаточной, коллекции в новом фреймворке вообще не синхронизированы. Если код, обрабатывающий коллекцию, так или иначе должен синхронизироваться, синхронизация самой коллекции является избыточной мерой, связанной к тому же с растратой ресурсов.

3 Составные части приложения Android

Опираясь на базовые концепции, изложенные в предыдущих главах и позволяющие писать надежный код на языке Java, перейдем к этой главе, в которой делается введение в основные и самые важные темы, связанные с программированием на платформе Android.

Сравнение Android и традиционных моделей программирования

Операционные системы традиционно используют единую точку входа, которая часто называется `main`.

`main` может производить синтаксический разбор некоторых аргументов командной строки, а потом переходить к исполнению цикла, который будет считывать пользовательский ввод и выдавать вывод. Операционная система загружает код программы в процесс, а потом приступает к его исполнению. В принципе, процессы такого рода выглядят так, как показано на рис. 3.1.

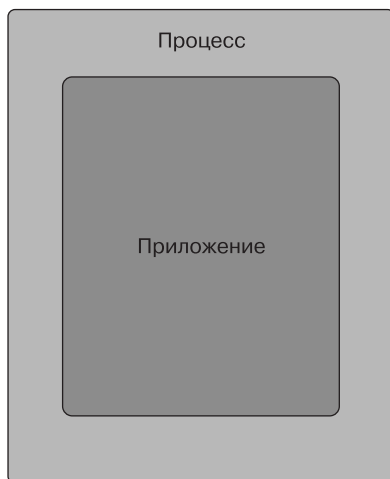


Рис. 3.1. Простое приложение в процессе

Когда программы написаны на Java, схема получается несколько сложнее: виртуальная машина (VM) Java в процессе работы загружает байт-код для инстанцирования классов Java, а программа их использует (рис. 3.2). Если вы работаете с системой насыщенного пользовательского интерфейса, например Swing, то сначала можно запустить систему пользовательского интерфейса, а потом написать обратные вызовы к вашему коду, который будет обрабатывать события.



Рис. 3.2. Приложение Java, использующее виртуальную машину Java, в процессе работы

В Android применяется более насыщенный и сложный вариант данного подхода, так как в приложении имеется несколько точек входа. Программы Android должны быть рассчитаны на то, что система будет запускать их в различных местах, в зависимости от того, откуда приходит пользователь и в какое место он направляется. Ваша программа представляет собой не иерархию мест, а взаимодействующую группу компонентов, запуск которых может выходить за рамки нормальной функциональности приложения. Например, компонент для сканирования штрихкодов предоставляет самостоятельную функцию, которую многие приложения могут интегрировать в свои рабочие циклы, связанные с пользовательским интерфейсом. Чтобы избавить пользователя от необходимости самостоятельно запускать каждое приложение, компоненты сами активизируют друг друга, для осуществления взаимодействия по команде пользователя.

Активности, намерения и задачи

Активность (activity) в Android — это и элемент для взаимодействия с пользователем (обычно активность целиком занимает экран мобильного устройства с Android), и элемент исполнения. Когда вы создаете интерактивную программу

Android, то начинаете с создания подклассов на основе класса `Activity`. Активности предоставляют многократно, взаимозаменяемые элементы рабочего цикла компонентов пользовательского интерфейса во всем приложении Android.

Как в таком случае одна активность активизирует другую и передает информацию о том, что намерен делать пользователь? Коммуникационным блоком является класс `Intent` (в переводе с английского — «намерение»). `Intent` представляет абстрактное описание операции, которую одна активность должна выполнить по требованию другой, например сделать фотоснимок. Намерения образуют основу системы слабого связывания, которая позволяет активностям запускать друг друга. Когда приложение назначает намерение, может возникнуть ситуация, в которой несколько различных активностей могут зарегистрироваться на выполнение желаемой операции.

На одном уровне абстрагирования приложения Android во многом напоминают веб-приложения. Активности аналогичны сервлетам веб-приложения. Хорошо написанная активность отвечает за управление одной страницей пользовательского интерфейса. Каждая активность имеет собственное уникальное имя. Пользователи переходят в веб-приложении от страницы к странице, следуя по ссылкам. А в приложении Android пользовательские взаимодействия активизируются через намерения. Новые страницы могут использовать старые страницы, просто связываясь с ними через ссылки. Если в мире веб-приложений некоторые сервлеты предоставляют пользовательские интерфейсы, а другие — API для служб, то в мире приложений Android активности предоставляют пользовательские интерфейсы, а классы `Service` и `ContentProvider`, о которых мы поговорим чуть ниже, обеспечивают программный доступ к сервисам. Понимая это архитектурное сходство, вы сможете писать приложения Android, эффективно использующие фреймворк Android.

Вы уже создали код для активности в тестовом приложении, которое мы выполняли для проверки того, правильно ли установлен комплект Android SDK. Давайте вновь обратимся к этому коду:

```
public class TestActivity extends Activity {  
    /** Вызывается при первом создании активности. */  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
    }  
}
```

Когда система запускает эту активность, вызывается конструктор для `TestActivity` подкласса `Activity`, после чего вызывается его метод `onCreate`. Таким образом, загружается и отображается иерархия видов, описанная в файле `main.xml`. Метод `onCreate` запускает жизненный цикл `Activity`, который будет подробно рассмотрен в главе 10.

Класс `Activity` — это один из важнейших классов в системе Android. Он обеспечивает модульность приложений и позволяет совместно использовать функции. Активность (`Activity`) взаимодействует со средой времени исполнения Android для реализации ключевых аспектов жизненного цикла приложения. Каждая активность также может независимо конфигурироваться при помощи класса `Context`.



Мы используем термин «активность» в качестве названия экземпляров класса Activity аналогично тому, как термин «объект» означает экземпляры различных классов.

Каждая активность в приложении Android в целом не зависит от других активностей. Код, реализующий одну активность, не занимается непосредственным вызовом методов в коде, который реализует другую активность. Другие элементы фреймворка Android — в частности, уже упоминавшиеся намерения (Intent) — используются для управления коммуникацией. Поэтому категорически не рекомендуется сохранять ссылки на объекты Activity. Среда времени исполнения Android (Android Runtime Activity), создающая активности, а также управляющая активностями и другими компонентами приложения, часто возвращает используемую ими память в общий пул, чтобы ограничивать отдельные задачи и выделять на каждую из них относительно небольшой объем памяти.



Часто начинающие программисты, работающие с Android, пытаются не позволить жизненному циклу компонента Android выполнять операции над экземплярами Activity этой программы. Можете быть уверены, что спонтанные попытки управлять памятью активностей будут в основном контрпродуктивными.

Чтобы не было необходимости основывать управление рабочими циклами в пользовательском интерфейсе на вызовах методов, приложения описывают намерение (Intent), которое они хотят выполнить, и запрашивают у системы найти подходящий способ для этого. Приложение Android Home Screen (Домашняя страница Android) запускает вашу программу, пользуясь этими описаниями, и каждое приложение затем может действовать так же, применяя подобранные таким образом намерения. Получающийся в результате рабочий цикл называется задачей (task). Задача — это цепь активностей, которая зачастую может захватывать более одного приложения и, конечно же, гораздо больше одного процесса. На рис. 3.3 показана задача, которая распространяется на три приложения и множество активностей (в табл. 3.1 приводится пример). Цепь активностей, которые составляют эту задачу, распространяется на три отдельных процесса и на три кучи и может существовать независимо от других задач, которые, возможно, будут запускать другие экземпляры подклассов той же активности (класса Activity).

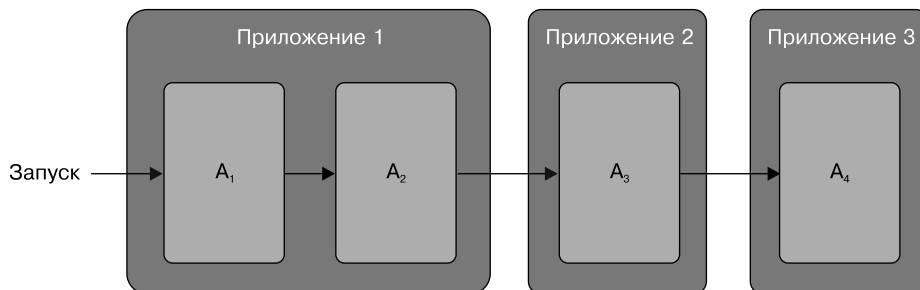


Рис. 3.3. Активности в единой задаче, которая распространяется на несколько приложений

Таблица 3.1. Примеры единой задачи, состоящей из активностей, которые относятся к разным приложениям

Приложение	Активность	Следующее действие пользователя
Сообщения	Просмотр списка сообщений	Пользователь выбирает сообщение из списка
Сообщения	Просмотр сообщения	Пользователь переходит в Menu ► View Contact (Меню ► Просмотреть контакт)
Контакты	Просмотр контакта	Пользователь выбирает Call Mobile (Звонок по мобильному телефону)
Телефон	Звонок на мобильный номер контакта	

Другие компоненты Android

В создании приложений для Android участвуют еще три других компонента: сервисы (services), поставщики содержимого (content providers) и широкополосные приемники (broadcast receivers). Класс Service поддерживает фоновые функции. Класс ContentProvider обеспечивает множественным приложениям доступ к хранилищу данных, а Broadcast Receiver позволяет нескольким участникам слушать намерения, которые транслируются приложениями в системе.

Еще есть класс Application, но вы увидите, что приложение, по сравнению с его компонентами, является в Android относительно малозначительной сущностью. Хорошо написанные приложения «растворяются» в среде Android, где они могут запускать активности в других приложениях, чтобы воспользоваться их функциями, либо могут дополнять собственную функциональность, пользуясь другими компонентами Android, поддерживающими их работу.

Информация о константах, используемых для ассоциирования намерений со стандартными сборками приложений Android, приводится по адресам <http://developer.android.com/guide/topics/intents/intents-filters.html> и <http://developer.android.com/guide/appendix/g-app-intents.html>. Можно считать поставщики содержимого и намерения Android вторичными API, которые следует научиться применять, чтобы иметь возможность пользоваться сильнейшими качествами Android и плавно интегрировать свое приложение в платформу Android.

Сервис

Класс Service в Android предназначен для решения фоновых задач, которые могут быть в активном состоянии (выполняться), но эта работа никак не будет отражаться на экране. Приложение-плеер также, вероятно, будет реализовано в форме сервиса, чтобы воспроизведение музыки не прекращалось, пока пользователь, к примеру, будет просматривать веб-страницы. Сервисы также позволяют приложениям совместно использовать функции в ходе долговременных соединений. Такая практика напоминает интернет-сервисы (веб-службы), такие как FTP и HTTP, которые ожидают, пока их не активирует запрос от клиента. На платформе Android не принята практика возвращения сервисных ресурсов в общее пользование, поэтому,

когда сервис запускается, он, скорее всего, будет доступен, если только нет серьезного дефицита памяти.

Подобно классу `Activity`, класс `Service` предлагает методы, управляющие его жизненным циклом, которые, в частности, отвечают за остановку и перезапуск сервиса.

Поставщики содержимого

Поставщики содержимого — это компоненты, которые в целом аналогичны веб-сервису с передачей состояния представления (RESTful). Поставщики содержимого находятся по URI (универсальному идентификатору ресурса), а операции подкласса `ContentProvider` можно сопоставить с веб-операциями RESTful-сервисов в том, как, например, происходит при работе с ними ввод и получение данных. Особый уникальный идентификатор ресурса, начинающийся с `content://` (этот идентификатор распознается в пределах всего локального устройства), предоставляет вам доступ к данным, содержащимся в поставщике содержимого. Для использования `ContentProvider` вы указываете уникальный идентификатор ресурса и сообщаете, как нужно поступить с данными, на которые поставлена ссылка. Ниже приведен список операций, которые способны выполнять поставщики содержимого. Это знаменитый «квартет» базовых активностей, связанных с обработкой данных: создание (вставка), считывание (запрос), обновление и удаление.

- Вставка (Insert) — метод `insert` класса `ContentProvider` аналогичен операции `POST`, которая применяется с REST-сервисами. Он вставляет новые записи в базу данных.
- Запрос (Query) — метод `query` класса `ContentProvider` аналогичен операции `GET`, которая применяется с REST-сервисами. Он возвращает набор записей в специализированном классе коллекций, называемом `Cursor`.
- Обновление (Update) — метод `update` класса `ContentProvider` аналогичен операции `UPDATE`, которая применяется с REST-сервисами. Он заменяет старые записи в базе данных новыми.
- Удаление (Delete) — метод `delete` класса `ContentProvider` аналогичен операции `DELETE`, которая применяется с REST-сервисами. Он удаляет из базы данных методы, соответствующие заданным условиям.



REST означает «передача состояния представления» (Representational State Transfer). Это не официальный протокол в отличие, например, от HTTP. REST — это скорее концептуальный фреймворк, в котором протокол HTTP используется в качестве основы для простого доступа к данным. Реализации REST могут отличаться, однако все они стремятся к простоте. API поставщиков содержимого Android формализует REST-подобные операции в отдельный API и вообще разработан в духе простоты, присущей REST. REST посвящена следующая статья «Википедии»: <http://en.wikipedia.org/wiki/REST>¹.

¹ Существует соответствующая статья на русском языке: <http://ru.wikipedia.org/wiki/REST>. — *Примеч. пер.*

Компоненты поставщика содержимого Android — это ядро модели содержимого в этой операционной системе. Предоставляя `ContentProvider`, ваше приложение может использовать данные совместно с другими приложениями и управлять моделью данных приложения. Класс-партнер, `ContentResolver`, позволяет другим компонентам системы Android находить поставщики содержимого. Поставщики содержимого встретятся вам во всех частях платформы. Вы увидите, что они используются как в самой операционной системе, так и в приложениях, написанных другими разработчиками. Необходимо отметить, что основные приложения Android используют поставщики содержимого, которые могут предоставлять быстрые и отлаженные функции для новых приложений Android. В том числе поставщики предоставляются браузером (`Browser`), календарем (`Calendar`), списком контактов (`Contacts`), историей звонков (`Call Log`), медиа (`Media`) и настройками (`Settings`).

Поставщики содержимого — это уникальное явление в системах межпроцессной связи (IPC), встречающихся на других платформах, в частности CORBA, RMI и DCOM¹, в которых значительную роль играют вызовы удаленных процедур. Поставщики содержимого действуют и как механизм длительного хранения данных, и как форма межпроцессной коммуникации. Вместо того чтобы просто допустить возможность межпроцессных вызовов методов, поставщики содержимого позволяют разработчикам эффективно использовать целые базы данных SQL — совместно для нескольких процессов. Поставщики содержимого дают возможность совместно применять не объекты, а целые таблицы SQL.

Использование поставщика содержимого

При написании серверного веб-приложения разработчик обычно пользуется доступом к двум разнородным API. API первого вида — это код, библиотеки и модель данных, из которых состоит создаваемая программа. Но, кроме работы с этими компонентами, разработчику, вероятно, потребуется делать вызовы к другим веб-приложениям, используя их тщательно определенные API для получения данных и сервисов, предоставляемых этими приложениями. Аналогично два уровня API предоставляются и в Android: во-первых, это библиотеки и сервисы, к которым ваш код обращается напрямую, а во-вторых, что не менее важно, — ряд сервисов, доступных благодаря поставщикам содержимого. Поскольку поставщики содержимого очень важны в Android, мы сделаем здесь краткое введение в проблему и покажем, как написать клиент, использующий поставщик содержимого.

В данном примере применяется один из важнейших поставщиков содержимого — база данных `Contacts` (Контакты). Этот пример должен помочь вам более основательно понять, как поставщик содержимого вписывается в ваше приложение. Класс `ContentProvider` предоставляет центральный API для поставщиков содержимого, от которого можно производить подтипы для управления конкретными типами данных. Активности обращаются к экземплярам конкретного поставщика содержимого, используя класс `ContentResolver` и связанные с ним URL следующим образом:

¹ CORBA — общая архитектура брокера объектных запросов, RMI — вызов удаленных методов в языке Java, DCOM — распределенная модель компонентных объектов, технология Microsoft. — Примеч. пер.

```
Cursor c = getResolver().query(
    android.provider.ContactsContract.Data.CONTENT_URI,
    new String[] {
        android.provider.ContactsContract.Data._ID,
        android.provider.ContactsContract.Phone.NUMBER},
    null,
    null,
    null);;
```

База данных Contacts (Контакты) — это самостоятельное приложение, работающее в отдельном процессе. При использовании поставщика содержимого его операции над данными должны вызываться при помощи REST-подобных URI. URL поставщика содержимого всегда имеет следующую форму:

```
content://authority/path/id
```

Здесь *authority* (источник) — **Java-пакет пространства имен поставщика содержимого** (зачастую это относящееся к **Java пространство имен реализации поставщика содержимого**). Вот два примера уникальных идентификаторов ресурса, используемых с поставщиками содержимого:

```
// ссылка на отдельный контакт
content://contacts/people/25
```

```
// этот URI указывает на телефонные номера человека,
// чей ID в списке контактов равен "25"
content://contacts/people/25/phones
```

Когда разработчик вызывает метод `query` применительно к поставщику содержимого, вызов возвращает объект `Cursor`, реализующий интерфейс `android.database.Cursor`. Этот интерфейс позволяет получить один результат (например, строку из базы данных) за раз при помощи индекса, который автоматически обновляется при каждом получении нового результата. Разработчики, знакомые с технологией JDBC (взаимодействие Java и баз данных), могут сравнить эту ситуацию с использованием `java.sql.ResultSet`. В большинстве случаев объекты `Cursor` представляют результаты запросов к таблицам базы данных SQLite. Разработчик может получить доступ к полям курсора, пользуясь индексами из базовой таблицы SQLite. Вот пример итерирования курсора в Android и доступа к его полям:

```
// код из метода активности
Cursor contactsCursor = getResolver().query(
    ContactsContract.Contacts.CONTENT_URI,
    null,
    null,
    null,
    null);

if (contactsCursor.moveToFirst()) {
    int idx = contactsCursor.getColumnIndex(Contacts.People.DISPLAY_NAME);

    do { name = contactsCursor.getString(idx); }
    while (contactsCursor.moveToNext());
}
```

Здесь следует отметить, что всякий раз, когда клиент использует курсор от поставщика содержимого, курсор обязательно необходимо закрывать после окончания работы клиента с ним. Если этого не сделать, может получиться серьезная утечка памяти, которая в итоге может привести к краху приложения. В Android применяются два метода, которые позволяют гарантировать, что курсоры поставщика будут непременно закрыты, если не используются:

- активность напрямую вызывает `Cursor.close`;
- активность вызывает или `managedQuery` для запрашивания поставщиков содержимого, или `startManagingCursor(Cursor c)`. В обоих случаях работа основана на системе, наблюдающей за ссылками на курсоры и позволяющей узнать, когда у конкретной ссылки не остается активных клиентов. Когда счетчики ссылок показывают, что все клиенты закончили работу, система сама вызывает `Cursor.close`.

Подробнее о данных и поставщиках содержимого мы поговорим в главах 12 и 13.

Поставщики содержимого и Интернет

Вместе с компонентом `Activity` приложения Android поставщики содержимого предоставляют детали, необходимые для построения архитектуры MVC («Модель-вид-контроллер»¹). Кроме поддержки REST-подобных операций, поставщики содержимого поддерживают паттерн «Наблюдатель» (`observer`), который, в свою очередь, поддерживает MVC. Класс `ContentResolver` предоставляет метод `notifyChange`, вызываемый клиентским кодом, всякий раз, когда в основополагающей базе данных происходит изменение. Благодаря этому вызову широковещательное уведомление направляется всем объектам `Cursor`, зарегистрированным в качестве наблюдателей. Это делается при помощи метода `registerContentObserver`.

Можно подумать: «Допустим, так, но нужные мне данные находятся не на устройстве, а в Интернете». Оказывается, что Android располагает множеством инструментов, которые относительно упрощают доступ к сетевым данным. Возможно, вам приходилось пользоваться приложениями, работающими с данными, которые расположены в Интернете, и применяющими для этого сетевые классы Android. К сожалению, характерной чертой таких приложений является то, что им требуется довольно много времени, чтобы получить доступ к данным и доставить их вам с конкретного сервера, подключенного к Интернету. По вашему желанию такие программы даже могут отображать индикатор загрузки.

Не лучше ли будет проверить поставщики содержимого на прочность, кэшируя данные на локальном устройстве, а заодно испытать силу архитектуры Android, использующей паттерн MVC и сосредоточенной вокруг базы данных? Так свежие данные будут отображаться на экране пользовательского устройства сразу же, по мере поступления. Эти вопросы рассматриваются в главе 13. Из этой главы вы узнаете, как в Android комбинируются пользовательские интерфейсы, поставщики содержимого и связанные классы, сетевые API Android и поддержка MVC, что дает в сумме клиент REST. Этот клиент реализует преимущества, проистекающие из сходства архитектуры поставщиков содержимого и служб с передачей состояния

¹ Встречается также перевод «Модель-представление-контроллер». — *Примеч. пер.*

представления, и пользователь может не тратить время на разглядывание индикатора загрузки, пока приложение занимается выборкой данных.

Широковещательный приемник

Класс `BroadcastReceiver` реализует иной вариант высокоуровневого механизма межпроцессной коммуникации, действующего в Android. При этом используются объекты класса `Intent`. У `BroadcastReceiver` более простой жизненный цикл, чем у других рассмотренных нами компонентов. Широковещательный приемник получает действия, совершаемые объектами `Intent`, подобно тому как это делает `Activity`. Но в отличие от активности широковещательный приемник не имеет собственного пользовательского интерфейса. Типичный случай использования широковещательного приемника — получение сигнала, который активирует приложение в определенный момент времени.

Жизненные циклы компонентов

Как было указано выше, приложения Android не похожи на приложения большинства других операционных систем. Приложение Android — это не просто фрагмент кода, который запускается и после этого делает, что хочет. Приложения Android — это управляемые компоненты с полными жизненными циклами. Например, относящийся к активности метод `onCreate` вызывается при запуске приложения, а метод `onDestroy` — при завершении его работы. Жизненные циклы компонентов обеспечивают эффективное использование памяти приложений (области динамической памяти). Они позволяют сохранять состояние целых процессов и впоследствии восстанавливать это состояние. Таким образом, в системе Android одновременно может работать больше приложений, чем умещается в памяти.

Жизненный цикл активности

Из всех компонентов самым сложным жизненным циклом обладает активность. Здесь этот цикл изображен схематически, далее мы рассмотрим, как указанные переходы между состояниями осуществляются в коде. На рис. 3.4 показаны состояния и переходы между ними, происходящие в течение жизненного цикла активности. При обработке жизненного цикла важнее всего выбрать, какие обратные вызовы необходимо реализовать, и знать, когда эти обратные вызовы должны выполняться.

В главе 10 эта тема будет рассмотрена подробно. Пока остановимся на двух методах класса `Activity`: `onSaveInstanceState` и `onCreate`. Среда времени исполнения вызывает первый метод, чтобы предупредить приложение о необходимости сохранить текущее состояние. Второй метод вызывается для того, чтобы новый экземпляр `Activity` мог восстановить приложение в том состоянии, в котором оно было сохранено. Следующие фрагменты кода с реализациями методов взяты из главы 10, где приведен полный листинг программы, в том числе переменные членов (`member variables`), к которым относится код:

```

@Override
protected void onSaveInstanceState(Bundle outState) {
    // сохранение состояния, специфичного для экземпляра
    outState.putString("answer", state);
    super.onSaveInstanceState(outState);
    Log.i(TAG, "onSaveInstanceState");
}

```

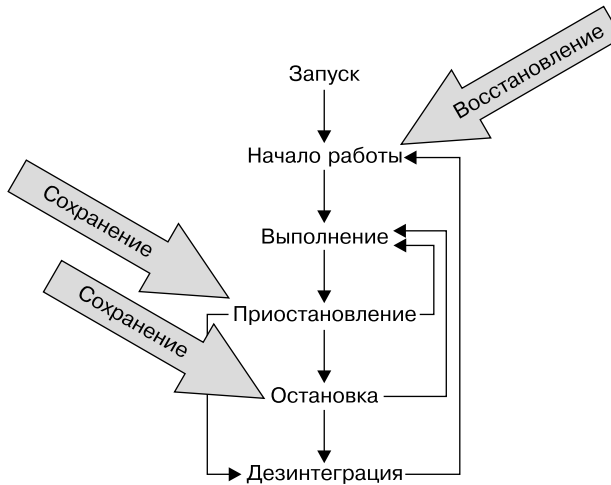


Рис. 3.4. Состояния жизненного цикла активности

Среда времени исполнения вызывает метод `onSaveInstanceState` после того, как определит, что активность, вероятно, придется уничтожить, но желательно иметь возможность восстановить ее позже. Это важное отличие от других методов, связанных с жизненным циклом, которые вызываются при изменении состояния. Если, например, происходит явное завершение активности, нет необходимости восстанавливать ее состояние, даже притом, что активность на пути к завершению пройдет через состояние приостановления (`paused`) и будет вызван ее метод `onPause`. Как показано в предыдущем фрагменте кода, работа, которую необходимо выполнить в методе `onSaveInstanceState`, заключается в сохранении любого состояния, которое позволит пользователю возобновить работу с приложением позднее. Возможно, пользователь даже не осознает, что активность со времени последнего ее использования могла быть дезинтегрирована и только потом восстановлена.

```

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    // восстановить состояние: мы знаем, что savedInstanceState не равно null
    String answer = savedInstanceState.getString("answer");
    // ...
    Log.i(TAG, "onRestoreInstanceState"
        + (null == savedInstanceState ? "" : RESTORE) + " " + answer);
}

```

Метод `onRestoreInstanceState` вызывается, когда восстанавливается активность, которая ранее была дезинтегрирована. С этого момента работает новый экземпляр `Activity` вашего приложения. Данные, которые вы сохранили в прошлом экземпляре этой активности при помощи `onSaveInstanceState`, передаются новой активности методом `onRestoreInstanceState`.

Можно подумать, что такой сложный жизненный цикл и жесткие требования, которых приходится придерживаться (эти требования обусловлены оптимальным использованием динамической памяти), приводят к тому, что жизненный цикл активности в коде Android будет сразу заметен и что придется тратить массу времени на обслуживание жизненного цикла активности и выполнение всех этих требований. Но это не так.

В массе примеров кода на Android, в частности в небольших примерах, реализуется очень мало обратных вызовов, связанных с жизненным циклом. Это объясняется тем, что родительский класс `Activity` обрабатывает обратные вызовы жизненного цикла, класс `View` и дочерние классы класса `View`, а также сохраняет и их состояния (рис. 3.5). Поэтому в очень многих случаях классы `View` в Android обеспечивают весь функционал, необходимый для работы пользовательского интерфейса, и приложениям Android не приходится явно обрабатывать большинство обратных вызовов, связанных с жизненным циклом.

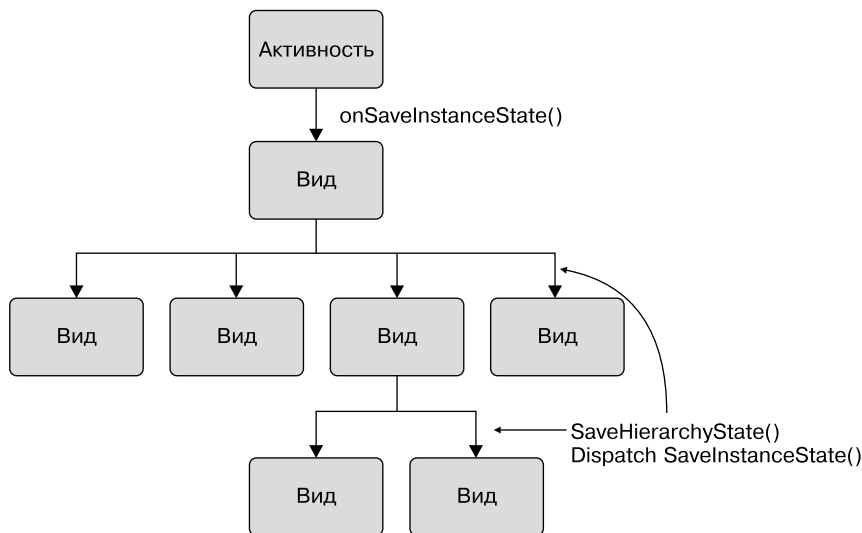


Рис. 3.5. Сохранение состояния пользовательского интерфейса

В принципе, этот механизм хорош, поскольку с ним программировать для Android гораздо удобнее. Все, что показано на рис. 3.5, не требует написания никакого специального кода. Но здесь есть и отрицательная сторона: программист может настолько привыкнуть игнорировать жизненный цикл активности, что его код в конце концов превратится в неразбериху, кишашую ошибками. Вот почему мы отдельно подчеркиваем, как важно понимать жизненные циклы и почему в главе 10 будет рассказано, как обрабатывать все обратные вызовы, связанные с жизненным

циклом, а также регистрировать их. Начинать работу, полностью понимая жизненный цикл активности, — это, пожалуй, самая важная предпосылка, которая поможет избежать сложно диагностируемых ошибок.

Перенос программ на платформу Android

Мы уже изучили, что архитектура приложения Android **коренным образом** отличается от архитектуры приложений, работающих в типичных операционных системах для ПК. Более того, эта архитектура значительно отличается и от архитектуры приложений для многих мобильных устройств, в частности приложений для iPhone, iPod Touch и iPad, работающих в операционной системе iOS. Если **пытаться** переносить (портировать) приложения на платформу Android, подстраивая архитектуру этой платформы под свое приложение, насильно втискивая на платформу Android приложения с традиционной архитектурой — словом, делая «подстрочный перевод» с языков Objective-C, C++ или C# методом за методом, — скорее всего, результат будет плачевным.

Если вы хотите перенести готовую программу на платформу Android, сначала разберите ее на части: модель данных, пользовательский интерфейс, крупнейшие неинтерактивные модули и библиотеки. Все эти элементы должны портироваться (заново реализовываться) в модели приложения Android **так, чтобы они максимально** вписывались в эту модель. Среда времени исполнения Android в чем-то схожа с другими современными управляемыми средами времени исполнения и с языковыми системами. Подробнее разобравшись с Android, вы будете достаточно подкованы, чтобы различить архитектурные аналогии с другими платформами, и при портировании будете выбирать более рациональные варианты реализации.

Статические ресурсы приложения и его контекст

Итак, мы рассмотрели некоторые базовые конструкции, образующие архитектуру приложения Android. Давайте поговорим о физических артефактах.

В главе 1 была изложена базовая информация о комплекте для разработки ПО под Android (Android SDK). В главе 5 мы сузим фокус проблемы, подробно рассмотрев один из наиболее популярных инструментов для разработки под Android — интегрированную среду разработки Eclipse. Сделаем еще один шаг и изучим, как в проекте организуется код.

Еще раз подчеркнем, что *проект* — это рабочее пространство, выделенное для создания целостного развертываемого артефакта. В широком смысле в мире Java артефакт может представлять собой всего лишь библиотеку (файл JAR, который не может запускаться сам по себе, но реализует определенный специфический функционал). С другой стороны, артефакт может быть развертываемым веб-приложением или приложением для ПК, которое вызывается двойным щелчком на ярлыке на Рабочем столе. В контексте Android артефакт — это, как правило,

одиночная служба, которую можно запустить: `ContentProvider`, `Service` или `Activity`. Поставщик содержимого, используемый отдельной активностью, определенно может начинать свой жизненный цикл как часть проекта активности. Если возникнет ситуация, в которой им должна будет воспользоваться другая активность, то, возможно, потребуется произвести рефакторинг и выделить этот поставщик содержимого в собственный проект.

Исходный код приложений Android практически всегда использует следующую иерархию каталогов:

```
AndroidManifest.xml
bin/
    ... компилируемые классы ...
gen/
    ... код, автоматически генерируемый системой сборки android ...
res/
    layout/
        ...содержит файлы компоновки приложения...
    drawable/
        ...содержит изображения, патчи, XML для изобразительных ресурсов...
    raw/
        ...содержит файлы с данными, загружаемыми в виде потоков...
    values/
        ...содержит XML-файлы, в которых записаны строковые
            и числовые значения, используемые в коде...
src/
    java/package/directories/
```

Как правило, компилятор Java ожидает, что деревья каталогов будут содержать файлы с исходным кодом Java (JAVA), к которым компилятор применяет синтаксический разбор, и двоичные файлы (CLASS), выдаваемые в качестве вывода. Хотя это и не необходимо, но управлять проектом становится гораздо проще, если у деревьев, содержащих файлы первого и второго вида, будут разные корни, в качестве которых обычно выступают каталоги, называемые соответственно `src` и `bin`.

В проекте Android присутствуют еще два очень важных дерева каталогов — `res` и `gen`. В `res` содержатся определения статических ресурсов: цветов, константных строк, компоновки и т. д. Инструменты Android выполняют предварительную обработку этих определений, преобразуя их в значительно оптимизированные представления и исходный код Java, через который код приложения ссылается на эти представления. Автоматически сгенерированный код, а также код, созданный для объектов AIDL (см. врезку «AIDL и вызовы удаленных процедур» далее) помещается в каталоге `gen`. Система компилирует содержимое обоих каталогов и полученный в результате материал помещает в `bin`. Чуть ниже будет объяснено, почему каталог `res` так важен для обеспечения доступа к данным приложения через объект `Context`.



Если вы добавляете в проект систему контроля за изменениями, например Git, Subversion или Perforce, можете смело исключать из проекта каталоги `bin` и `gen`!

Организация исходного кода Java

Исходный код вашего приложения находится в каталоге `src`. Как было указано в главе 2, нужно поместить весь код в пакет, имя которого является производным от доменного имени владельца кода. Предположим, например, что вы — разработчик в огромной фирме, сайт которой называется `awesome-android.net`. И вот вам поручено написать для `voracious-carrier.com` программу, которая выдавала бы прогноз погоды. Вероятно, вы поместите весь ваш код в пакет `com.voraciouscarrier.weatherprediction` или, например, `com.voracious_carrier.weather_prediction`. Хотя символ — вполне можно использовать в доменных именах службы DNS, он не допускается в названиях пакетов Java. Пользовательский интерфейс для этого грандиозного приложения может находиться в пакете `com.voraciouscarrier.weatherprediction.ui`, а модель приложения — в `com.voraciouscarrier.weatherprediction.futureweather`.

Заглянув в каталог `src` проекта, вы обнаружите в нем подкаталог, который называется `com`. Он, в свою очередь, содержит каталог `voraciouscarrier` и т. д. Дерево исходных каталогов структурно отражает дерево пакетов. Компилятор Java ожидает, что каталоги будут выстроены именно таким образом, и, возможно, не сможет скомпилировать ваш код, если он имеет иную структуру.

Наконец, когда ваш поставщик содержимого `FutureWeather` оказывается ценен сам по себе, вы решаете извлечь его и использовать в новом проекте, с таким пространством имен пакета, которое не ограничивается названием приложения, где изначально создавался этот поставщик. Выполнение такой операции вручную — это тихий ужас. Нужно будет создать новую структуру каталогов, правильно разместить файлы в данной структуре, исправить названия пакетов, находящиеся в начале имени каждого файла с исходным кодом, и, наконец, исправить ссылки на все те элементы, которые вы переместили.

Здесь вам очень пригодятся инструменты рефакторинга, имеющиеся в Eclipse. Всего несколько щелчков кнопкой мыши — и можно создать новый проект для поддерева, которое уже является автономным, вырезать код поставщика содержимого и вставить его в новое дерево, а потом переименовать пакеты так, как требуется. Eclipse подправит большую часть моментов, в том числе изменившиеся ссылки. Подробнее мы обсуждаем этот вопрос в главе 5, где говорим о среде Eclipse.

Необходимо оговориться, что не следует пользоваться сокращенными названиями пакетов, например писать просто `weatherprediction`. Даже если вы совершенно уверены, что создаваемый вами код никогда не будет использоваться вне его текущего контекста, вы, возможно, захотите задействовать в этом контексте код, взятый извне. Не допускайте потенциальной возможности конфликта имен.

Ресурсы

Кроме кода, программам может понадобиться хранить значительные объемы данных, чтобы управлять своим поведением во время исполнения. Эти данные могут состоять из изображений, которые нужно вывести, либо из простых текстовых строк, указывать цвет фона или название шрифта, который следует использовать.

Такие данные называются ресурсами. Вместе вся эта информация образует *контекст* приложения. Android предоставляет доступ к контексту через класс `Context`. Класс `Context` может дополняться как `Activity`, так и `Service`. Это означает, что все активности и сервисы имеют доступ к данным `Context` через указатель `this`. В следующих разделах будет рассказано, как использовать объект `Context` для доступа к ресурсам приложения во время исполнения.

В приложении Android все изображения, ярлыки и файлы компоновки пользовательского интерфейса располагаются в каталоге (директории) `res`. Обычно в нем содержится не менее четырех подкаталогов:

- `layout` — содержит XML-файлы компоновки пользовательского интерфейса Android, которые будут описаны в главе 6;
- `drawable` — включает в себя отрисовываемые объекты, в частности ярлык приложения, упоминавшийся в предыдущем разделе;
- `raw` — содержит файлы, которые могут считываться в потоковом режиме во время исполнения приложения. Такие необработанные файлы отлично подходят для того, чтобы сообщать приложению информацию для отладки, так как снимают необходимость выходить в сеть и получать оттуда данные;
- `values` — включает в себя значения, которые приложение будет считывать во время исполнения, или статические данные, которые приложение будет использовать для таких целей, как, например, интернационализация строк пользовательского интерфейса.

Приложения получают доступ к ресурсам этих каталогов, пользуясь методом `Context.getResources()` и классом `R`.

Для доступа к данным каталога `res` разработчик Java, предпочитающий работать в традиционном стиле, может писать такой код, в котором будут строиться относительные пути к файлам, а потом для открытия ресурсов будет использоваться файловый API. После загрузки нескольких байтов ресурсов разработчик, вероятно, решит произвести синтаксический разбор формата, специфичного для данного приложения, чтобы наконец получить доступ к элементам, без которых не обойдется ни одно приложение: к изображениям, строкам и файлам с данными. Учитывая, что все приложения нуждаются в загрузке схожей информации, в Android включена утилита для прекомпиляции, которая преобразует ресурсы в объекты данных, к которым можно получить доступ программными методами. Этот инструмент интегрируется с IDE, обеспечивая легкость создания и поддержки данных-ресурсов.

Утилита создает каталог под названием `gen`. Этот каталог содержит класс, который всегда называется `R`. Данный класс находится в пакете приложения Java, упомянутом в файле описания Android. Файл класса `R` содержит поля, которые уникально идентифицируют все ресурсы в структуре пакета приложения. Разработчик вызывает метод `Context.getResources` для получения экземпляра `android.content.res.Resources`, в котором и располагаются ресурсы приложения. (Методы в классе `Context` можно вызывать напрямую, так как `Activity` — как и `Service` — дополняют `Context`.) Затем разработчики вызывают методы объекта `Resources` для получения ресурсов желаемого типа, как показано ниже:

```
// код в методе Activity
String helloWorld = this.getResources().getString(R.string.hello_world);
int anInt = this.getResources().getInteger(R.integer.an_int);
```

Как вы впоследствии увидите, класс `R` в Android является вездесущим, обеспечивая легкий доступ к таким ресурсам, как, например, компоненты файлов пользовательского интерфейса.



Правила видимости, действующие в Java, требуют, чтобы в рамках одного пакета классы были видимы друг для друга, даже если они явно не импортировались (при помощи оператора `import`). Это правило соблюдается и в том случае, когда классы не находятся в одном каталоге файловой системы. Инструментарий Android автоматически размещает класс `R` в том пакете Java, который помечен в манифесте атрибутом `package`. Однако стоит сделать и так, чтобы именно в этом пакете находилась большая часть вашего кода. Атрибут `package` в манифесте — это уникальное пространство имен для вашего приложения. Нет необходимости, чтобы это пространство соответствовало уникальному пространству имен, которое вы используете для вашего кода в качестве корневого пакета, но такой подход, несомненно, хорош.

Описания приложений

Наряду с данными, используемыми работающим приложением, приложению также требуется способ для описания требуемого рабочего окружения. Нужные данные — имя этого приложения, регистрируемые в нем намерения, необходимые права доступа и другая информация, которая описывает приложение для системы Android. Эти данные хранятся в файле, называемом файлом описания (манифестом). Android требует, чтобы все перечисленные параметры явно описывались в XML-файле `AndroidManifest.xml`. Здесь приложение объявляет о наличии поставщиков содержимого, сервисов, необходимых прав доступа и других элементов. Информация из файла описания также доступна работающему приложению через его контекст. Файл описания организует приложение Android в стройную структуру, которая совместно применяется другими приложениями и позволяет операционной системе загружать и исполнять приложения в управляемой среде. Структура включает в себя обычное расположение директорий и файлы таких типов, которые обычно содержатся в этих директориях.

Итак, четыре компонента приложения Android — `Activity`, `Service`, `ContentProvider` и `BroadcastReceiver` — формируют основу для разработки приложений в Android (рис. 3.6). Чтобы пользоваться ими всеми, приложение должно содержать соответствующие объявления в своем файле `AndroidManifest.xml`.

Класс `Application`

Однако среди компонентов Android есть и «пятый мушкетер»: класс `Application`. Но многие приложения Android не образуют подклассов от `Application`. Поскольку в большинстве случаев без подклассов `Application` можно обойтись, мастер установки проекта Android не создает этот класс автоматически.

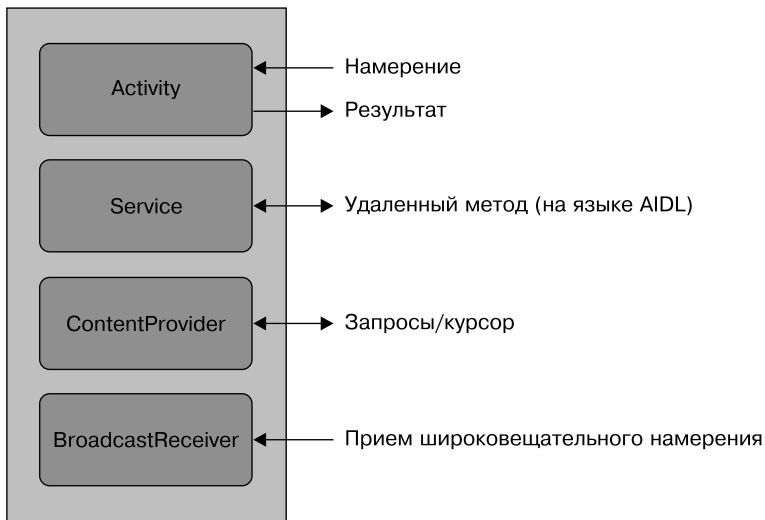


Рис. 3.6. Четыре разновидности компонентов в Android

Параметры инициализации в файле AndroidManifest.xml

Ниже приведен код файла описания Android из нашего тестового приложения, которое мы создали в главе 1. Тестовое приложение предназначено исключительно для демонстрации базовой компоновки приложения Android. В этом файле описания Android содержатся базовые элементы, о которых мы говорили выше:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.oreilly.demo.pa.ch01.testapp"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.ACCESS_MOCK_LOCATION" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application android:icon="@drawable/icon"
        android:label="@string/app_name"
        android:debuggable="true">

        <activity android:name=".TestActivity"
            android:label="Test Activity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

```

        </intent-filter>
    </activity>

    <provider android:name=".TestProvider"
        android:authorities=
            "com.oreilly.demo.pa.ch11.video.FinchVideo"
        />

    <service android:name=".TestService"
        android:label="Test Service"/>

    <receiver
        android:name=".TestBroadcastReceiver"
        android:label="Test Broadcast Receiver"/>
</application>
<uses-sdk android:minSdkVersion="7" />
</manifest>

```

Как и в любом грамотном XML-файле, в строке 1 дается стандартное объявление версии XML и сообщается используемая кодировка. Далее определяется несколько параметров и объявляются необходимые права доступа для всего приложения. Все элементы подробно рассмотрены ниже.

○ manifest:

- `package="com.oreilly.demo.pa.ch01.testapp"` — пакет, в котором по умолчанию находятся модули приложения;
- `android:versionCode` — произвольное число, являющееся номером версии приложения Android. Каждое приложение должно содержать код версии, и этот код должен иметь больший порядковый номер в каждой новой версии, выходящей на рынок. Таким образом, другие программы (например, Android Market, установщики и средства запуска) с легкостью обнаруживают, какая версия программы старше. Этот же номер версии должен быть указан в имени вашего файла APK, чтобы было очевидно, какая версия в нем содержится;
- `android:versionName` — это строка, больше похожая на номера версий, которые обычно используются при нумерации приложений, например 1.0.3. Это идентификатор версии, который будет отображаться пользователю (либо вашим приложением, либо другими приложениями). Принцип наименования вы выбираете сами, но вообще принято использовать схему вида `m.n.o` для стольких чисел, сколько вам требуется. Так обозначаются последовательные этапы изменения приложения;
- `uses-permission android:name=...` — четыре объявления в файле описания TestApp показывают, что приложение должно будет использовать такие функции Android, которые требуют явного разрешения на их применение со стороны пользователя конкретного локального устройства, на котором работает приложение. Право доступа запрашивается на этапе установки приложения. С этого момента Android напоминает, что пользователь разрешил (или не разрешил) запускать это приложение и предоставлять ему доступ

к защищенным функциям. Многие права доступа уже определены в Android и описаны в документации Android (ищите `android.Manifest.permission`). Но, кроме того, вы можете задавать и собственные права доступа и использовать их для ограничения доступа других приложений к функциям вашего приложения — такое разрешение должен будет специально давать пользователь устройства, на котором установлена ваша программа. В качестве примера мы запросили следующие стандартные права доступа: `ACCESS_FINE_LOCATION` — требуется для получения информации о местоположении с датчика GPS; `CALL_PHONE` — позволяет пользователю совершать телефонные звонки; `ACCESS_MOCK_LOCATION` — дает возможность получать фиктивную информацию о местоположении, когда мы работаем с эмулятором; `INTERNET` — позволяет выходить в Интернет и получать оттуда данные.

○ `application`:

- `label` — предоставляет для приложения метку, которую может прочитать пользователь;
- `icon="@drawable/icon"` — имя для PNG-файла, содержащего изображение, которое вы будете использовать в качестве ярлыка для вашей программы. В данном случае мы указываем комплекту SDK Android искать файл ярлыка в подкаталоге `drawable` директории `res` (ресурсы) в приложении `TestApp`. Android будет пользоваться этим ярлыком для обозначения вашего приложения на рабочем столе (главном экране).

○ `activity` — переходя к обсуждению `TestActivity`, сначала определим несколько ее атрибутов. К числу важнейших атрибутов относятся следующие:

- `android:name` — имя класса для `Activity`. В полное название активности входит имя пакета (в данном приложении — `com.oreilly.demo.pa.ch01.testapp`), но, поскольку этот файл всегда используется в контексте пространства имен пакета, нет необходимости дополнительно указывать предшествующие части имени пакета, можно ограничиться `.TestActivity`. В данном случае даже точка в начале названия необязательна;
- `android:label` — это название, которое мы хотим отображать в верхней части экрана Android, когда на нем будет выполняться конкретная активность. Это имя определяется в файле `strings.xml` и сопоставляется с нашим приложением;
- `intent-filter` — здесь объявляется фильтр намерений, сообщающий Android, когда следует запускать данную активность (`Activity`). Когда приложение дает Android запрос на выполнение намерения (`Intent`), среда времени исполнения просматривает доступные активности и сервисы в поисках тех из них, которые могли бы выполнить этот запрос. Мы задаем два атрибута:
 - `action` — этот атрибут сообщает Android, как запускать приложение, после того, как среда времени исполнения решает, что запустить нужно именно его. Android ищет активность, объявляющую о своей готовности совершить действие `MAIN`. Любое приложение, которое система собирается запустить, должно иметь одну, и только одну активность (или сервис), объявляющую о готовности выполнить запуск;

- `category` — преобразователь намерений `Intent` в Android использует этот атрибут для дальнейшей квалификации искомого намерения (`Intent`). В этом случае квалификация заключается в том, что мы хотели бы отобразить данную активность (`Activity`) в пользовательском меню, чтобы пользователь мог выбрать активность для запуска этого приложения. Для решения данной задачи указывается категория `LAUNCHER`. Приложение может обойтись и без этого атрибута — просто оно не будет запускаться с рабочего стола Android. Как правило, на каждое приложение должен приходиться ровно один `LAUNCHER` и он будет появляться в том же фильтре намерений, что и начальная активность (`Activity`) вашего приложения.
- `provider` — обеспечивает объявление поставщика содержимого, `name` указывает имя класса поставщика, а `authorities` указывает источники (`authority` в составе `URI`), которые должен обрабатывать поставщик содержимого. Источник `URI` представляет информацию о домене `URI` поставщика содержимого и позволяет системе преобразования содержимого в Android находить поставщик, который должен обрабатывать `URI` конкретного типа. Чуть ниже в этой главе мы подробнее опишем, как клиенты используют поставщики содержимого. Мы объявили поставщик с именем `TestProvider`.
- `service` — позволяет приложению объявлять о поддержке указанной службы, где `name` указывает класс службы, а `label` предоставляет удобочитаемую метку для службы. Мы объявили службу с именем `.TestService`.
- `receiver` — позволяет объявлять поддержку приложения для широкополосного приемника, `name`, в свою очередь, указывает класс-приемник, а `label` предоставляет для этого приемника удобочитаемую метку. Мы объявили приемник с именем `TestBroadcastReceiver`.

Упаковка приложения Android: файл APK

Последним статическим компонентом приложения является, конечно, само приложение в упакованном виде. В Android есть приложение `apkbuilder`, создающее инсталлируемые файлы приложений Android. Эти файлы имеют расширение `APK`. `APK` — это формат-архив, как и многие другие форматы приложений, ориентированные на Java. Он содержит файл описания приложения, скомпилированные классы приложения и все ресурсы. Еще в Android SDK есть утилита `aapt` для преобразования содержимого каталога `res` в упакованный файл ресурса с сопутствующим классом `R`. Но разработчики, как правило, предпочитают перепоручить работу с этой утилитой своей среде разработки, которая поддерживает актуальность ресурсов и собирает файл `APK`.

Как только разработчик создаст файл `APK`, этот файл нужно сделать доступным для установки. Для этого существует несколько способов:

- воспользоваться каталогом интерфейсов `adb` или, что бывает чаще, интегрированной средой разработки (IDE);
- воспользоваться картой памяти;

- предоставить файл для скачивания на веб-сервере;
- загрузить файл на рынок Android Market, а потом выбрать команду Install (Установить).

Среда времени исполнения приложения Android

Уникальная архитектура компонентов приложения Android обязана своим появлением отчасти тому, как в Android реализуется среда многопроцессного исполнения. Чтобы эта среда подходила для большого количества приложений, получаемых от многих производителей, и минимальным требованием для работы было доверие каждому из производителей, Android задействует несколько экземпляров виртуальной машины Dalvik, по одному для каждой задачи. В следующих главах мы исследуем, как жизненные циклы компонентов позволяют Android оптимизировать способ сборки мусора в кучах (неупорядоченных массивах), используемых приложениями, и как обеспечивается выполнение стратегии восстановления памяти сразу в нескольких кучах.

В результате применения простого и надежного подхода к многопроцессной обработке в Android приходится эффективно делить память на несколько куч. Каждая куча должна быть относительно небольшой, чтобы в памяти одновременно могли уместиться несколько приложений. В каждой куче жизненный цикл компонентов позволяет компонентам, которые в данный момент не применяются — особенно неактивным компонентам пользовательского интерфейса, — попадать в сборку мусора, когда в куче не хватает места, а позже вновь восстанавливаться при необходимости. Это, в свою очередь, стимулирует создание моделей данных, при котором центральная роль достается базе данных. В таких моделях большинство данных по определению остается в неизменном состоянии в любое время. К этой проблеме мы еще вернемся на страницах нашей книги, в частности в главе 10.

Виртуальная машина Dalvik

Применяемый в Android подход к многопроцессной обработке, при котором используется несколько процессов и несколько экземпляров виртуальной машины Dalvik, требует, чтобы каждый экземпляр виртуальной машины эффективно использовал дисковое пространство. Такая эффективность достигается частично с помощью жизненного цикла компонентов, который позволяет объектам сначала попадать в сборку мусора, а потом восстанавливаться, а частично с помощью самой виртуальной машины. Android использует виртуальную машину Dalvik для эксплуатации байт-кодовой системы dex, разработанной специально для Android. Байт-коды dex расходуют дисковое пространство приблизительно вдвое экономичнее, чем байт-коды Java, что приводит практически к двукратному снижению издержек памяти, затрачиваемых на работу классов Java в каждом процессе. В системах Android также используется память, копирующая данные при записи, позволяющая совместно использовать динамическую память (heap memory) между несколькими экземплярами одного и того же исполняемого файла Dalvik.

Зигота: ветвление нового процесса

Кроме того, было бы неэффективно, если бы каждый новый процесс загружал все необходимые базовые классы каждый раз, когда начинает работу новый экземпляр виртуальной машины. Поскольку Android помещает каждое приложение в отдельный процесс, он может пользоваться преимуществами базовых операций ветвления, поддерживаемых в операционной системе Linux, на которой основан Android. Таким образом, операционная система может порождать новые процессы из процесса-шаблона, который находится в оптимальном состоянии для запуска нового экземпляра виртуальной машины. Такой процесс-шаблон называется зиготой (Zygote). Это экземпляр виртуальной машины Dalvik. В свою очередь, зигота Dalvik содержит набор предварительно загруженных классов, которые вместе с оставшимися свойствами состояния процесса-зиготы копируются в копии зиготы, возникающие после ветвления. Благодаря ветвлению, происходящему в зиготе, приложения Android запускаются быстро. Возможность предзагрузки классов работает вместе с системой «копирования при записи», применяемой для управления памятью в куче Dalvik. Таким образом, удастся радикально снизить объем потребляемой памяти.

Работа в безопасном режиме: процессы и пользователи

Действующие в Android меры безопасности в значительной степени основаны на предохранительных ограничениях, действующих на уровне операционной системы Linux, в частности на границах процессного и пользовательского уровней. Поскольку Android — это система для персонального использования, то есть устройством владеет один человек и этот же человек обычно с ним работает, — в Android интересным образом применяется присущий Linux механизм многопользовательской поддержки. Android создает нового пользователя и группу для приложений каждого отдельного поставщика. В результате каждому приложению предоставляется свой набор пользовательских привилегий (привилегии одинаковы только у тех приложений, которые подписаны одним и тем же поставщиком). Файлы, принадлежащие одному приложению, по умолчанию недоступны для применения приложениями других поставщиков.

Чтобы привести аналогию такой ситуации в Windows, представим себе, что на вашем компьютере одновременно работают текстовый редактор и браузер, причем текстовый редактор запущен от вашего имени, а браузер — от имени другого пользователя. Но вы можете видеть и использовать графические интерфейсы обоих пользователей. В полезном остатке получаем повышение безопасности, ведь приложение каждого поставщика работает в собственной «ячейке».

В операционных системах для персональных компьютеров работа в безопасном режиме обычно не доводится до такой степени — как только приложение установлено в системе, оно получает доступ ко всем пользовательским данным. Проектировщики Android предвидели появление многочисленных маленьких приложений от многих поставщиков и осознавали, что не всем приложениям можно

доверять. Поэтому приложения и не имеют прямого доступа к данным других приложений.

Полное описание системы безопасности в Android приводится в документации Android по адресу <http://developer.android.com/guide/topics/security/security.html>.

Библиотеки Android

Android добавляет для работы некоторое количество новых пакетов, которые вместе с несколькими деревьями пакетов из состава традиционного множества пакетов Java (J2SE) образуют **интерфейс программирования приложений для среды времени исполнения Android**. Рассмотрим, какие компоненты входят в состав этого комбинированного API.

android и dalvik

В этих деревьях пакетов содержится вся **Android-специфичная часть среды времени исполнения Android**. Данные библиотеки рассматриваются на протяжении большей части нашей книги, поскольку в их числе — библиотеки для работы с графическим пользовательским интерфейсом Android и работы с текстом. Эти библиотеки называются `android.graphics`, `android.view`, `android.widget` и `android.text`. К данной категории относятся также библиотеки фреймворка приложений — `android.app`, `android.content` и `android.database`. В них, в свою очередь, содержатся другие ключевые фреймворки, ориентированные на мобильное программирование, в частности `android.telephony` и `android.webkit`. Опытный программист, работающий с Android, вероятно, знаком как минимум с несколькими первыми пакетами из перечисленных здесь. Чтобы ознакомиться с документацией **Android в контексте дерева пакетов**, можете начать с «верхушки» документации по разработке для Android — <http://developer.android.com/reference/packages.html>.

java

В этом пакете содержится реализация основных библиотек среды времени исполнения Java. В пакете `java.lang` находится определение класса `Object`, который является базовым для всех объектов Java. В `java` также имеется пакет `util`, в котором находится фреймворк коллекций Java: `Array`, `Lists`, `Map`, `Set` и `Iterator`, а также их реализации. В библиотеке коллекций **Java предоставляется хорошо сконструированный набор структур данных для языка Java** — благодаря этим структурам вы избавляетесь от необходимости писать собственные списки со ссылками.

Как было указано в главе 2, в пакете `util` содержатся коллекции из двух различных поколений. Некоторые происходят из Java 1.1, а другие основаны на более новой, пересмотренной парадигме коллекций. Коллекции из 1.1 (например, `Vector` и `Hashtable`) полностью синхронизированы, их интерфейсы отличаются меньшим единообразием. Более новые версии (например, `HashMap` и `ArrayList`) не синхронизированы, но имеют лучшую взаимную совместимость и их следует использовать в первую очередь.

Для поддержания совместимости в рамках языка Java библиотеки Android также содержат реализации некоторых устаревших классов, пользоваться которыми

вообще не следует. Например, во фреймворке `Collections` имеется класс `Dictionary`, который совершенно устарел и не должен применяться. Интерфейс `Enumeration` заменен на `Iterator`, а `TimerTask` — на `ScheduledThreadPoolExecutor` из фреймворка многопоточных приложений (`Concurrency`). В справочной документации по Android эти устаревшие типы описаны и идентифицируются очень хорошо.

В java также содержатся базовые типы для некоторых других широко применяемых объектов, в частности `Currency`, `Date`, `TimeZone` и `UUID`, а также базовые фреймворки ввода-вывода, работы в сети, многопоточного исполнения и обеспечения безопасности.

Пакеты `awt` и `rmi` отсутствуют в иерархической структуре той версии java, которая используется с Android. Пакет `awt` заменяется библиотеками графического пользовательского интерфейса Android. Отсутствует единый аналог для системы удаленных сообщений (`remote messaging`), но внутри системы действуют `ServiceProviders`, использующие `Parcelable`. Этот механизм описан в разделе «Сериализация» данной главы. Он обеспечивает функционал, схожий со службой удаленных сообщений.

javax

Этот пакет очень напоминает пакет `java`. В нем содержатся элементы языка Java, которые официально считаются опциональными. Здесь вы найдете библиотеки, поведения из которых полностью описаны, но они необязательны для полноценной реализации языка Java. Поскольку в состав среды времени исполнения Android не входят некоторые компоненты, которые являются обязательными (поэтому Android не является Java, да и никогда на это не претендовал), в реализации `javax` для Android существуют характерные черты, необходимые для того, чтобы пакеты Android максимально напоминали пакеты Java. В обоих деревьях пакетов содержатся реализации библиотек, описанных как компоненты языка Java.

Самый важный элемент в `javax` — это фреймворк XML. Там находятся инструменты синтаксического разбора SAX и DOM, реализация XPath и XSLT.

Кроме того, в пакете `javax` содержатся важные расширения системы безопасности, а также API для OpenGL. Опытный программист, работающий с Java, заметит, что в реализации пакетов `javax`, применяемой в среде времени исполнения Android, не хватает нескольких важных разделов. В частности, тех, которые заняты работой с пользовательским интерфейсом и медийными компонентами. Отсутствуют, например, разделы `javax.swing`, `javax.sound` и прочие подобные им. Их место занимают другие пакеты, специфичные для Android.

org.apache.http

В этом дереве пакетов содержится стандартная реализация Apache для клиента и сервера HTTP — `HttpCore`. В данном пакете находится все необходимое для обмена информацией по протоколу HTTP, в том числе классы, представляющие сообщения, заголовки, соединения, запросы и отклики.

Проект Apache `HttpCore` расположен в Интернете по адресу <http://hc.apache.org/httpcomponents-core/index.html>.

org.w3c.dom, org.xml.sax, org.xmlpull, and org.json

Эти пакеты являются определениями общедоступных API для некоторых широко используемых форматов данных: XML (<http://www.w3.org/standards/xml>), XML Pull (<http://www.xmlpull.org/index.shtml>) и JSON (<http://www.json.org>).

Расширение Android

Итак, теперь у вас есть базовая путевая карта для перемещения по фреймворку Android, и возникает естественный вопрос: «Как воспользоваться всем этим, чтобы написать собственное приложение?» Как расширить тот фреймворк, который мы только что описали как очень сложный, но «неживой» сам по себе, и сделать на его основе какое-нибудь полезное приложение?

Логично, что на этот вопрос есть несколько ответов. Библиотеки Android *опранизованы* так, что они позволяют процессам получать доступ к фреймворку на различных его уровнях. Тем не менее описываемые здесь концепции глубоко укоренены в Android. Тщательно их изучив, вы сможете интуитивно догадываться, как лучше взаимодействовать с API Android.

Шаблон приложения Android

Двадцать лет назад типичное компьютерное приложение запускалось из командной строки, а весь объем его кода строился на уникальной программной логике. Но в наши дни приложения требуют очень сложной поддержки для работы с интерактивными пользовательскими интерфейсами, управления сетью, обработки вызовов и т. д. Логика поддержки должна быть одинаковой для всех приложений. Как мы уже видели, во фреймворке Android для решения этих проблем предлагается подход, ставший практически общепринятым, особенно в условиях, когда окружения приложений становятся все сложнее. Итак, говорят о *скелетном приложении* (skeleton application), или *шаблоне приложения* (application template).

Когда мы создавали простое скелетное приложение, при помощи которого проверяли установку нашего комплекта ПО для программирования под Android, приложение получилось полноценным и его вполне можно запускать. Эта программа способна выполнять сетевые запросы, отображать вывод на экране и обрабатывать экранный ввод. Данное приложение также могло обрабатывать входящие вызовы, и хотя пользоваться такой функцией мы не собирались, программа также могла проверять ваше местоположение. В программу еще не передана никакая полезная функциональность (логика), с которой программа могла бы работать. Это и есть скелетное приложение.

При работе с фреймворком Android задача разработчика заключается не столько в том, чтобы сделать полнофункциональную программу, сколько в том, чтобы реализовать нужные поведения, а потом встроить их в скелетное приложение в правильно подобранных точках расширения. Девиз MacApp, одного из первых фреймворков, где применялись скелетные приложения, таков: «Не вызывай нас,

мы сами вызовем тебя». Итак, если создание приложений для Android принципиально сводится к пониманию того, как нужно расширять фреймворк, целесообразно рассмотреть некоторые универсальные проверенные методы, помогающие делать такие расширения.

Переопределения и обратные вызовы

Простейшая в реализации сущность — которую разработчик обычно выбирает для внедрения во фреймворке нового поведения — это, как правило, обратный вызов. Базовая идея обратного вызова (данный паттерн довольно часто встречается в библиотеках Android) уже была проиллюстрирована в главе 2.

Чтобы создать точку расширения для добавления обратного вызова, в классе определяются две вещи. Во-первых, определяется интерфейс **Java** (обычно его название заканчивается на `Handler`, `Callback` или `Listener`¹). Этот интерфейс описывает, но не реализует действие обратного вызова. Кроме того, класс определяет метод-установщик, принимающий в качестве аргумента тот объект, который реализует интерфейс.

Рассмотрим приложение, в котором должен обрабатываться текстовый ввод, получаемый от пользователя. Ввод, редактирование и отображение текста, разумеется, требуют большого и сложного набора классов пользовательского интерфейса. Но приложение в то же время не должно решать большинство связанных с ними задач. Вместо этого в компоновку приложения добавляется библиотечный виджет — скажем, `EditText`. (Компоновка и виджеты описаны в разделе «Сборка графического интерфейса» главы 6.) Фреймворк обеспечивает инстанцирование виджета, отображение его на экране, обновление содержимого по мере того, как пользователь вводит текст, и т. д. Здесь происходит все, за исключением задач, для решения которых и предназначено ваше приложение: выполнение определенных действий при изменении текста. Здесь и применяется обратный вызов.

В документации по системе Android указано, что объект `EditText` определяет метод `addTextChangedListener`, принимающий в качестве аргумента объект `TextWatcher`. Данный объект определяет методы, активируемые, если изменяется текст, который содержится в виджете. Образец кода приложения может выглядеть так:

```
public class MyModel {
    public MyModel(TextView textBox) {
        textBox.addTextChangedListener(
            new TextWatcher() {
                public void afterTextChanged(Editable s) {
                    handleTextChange(s);
                }
                public void beforeTextChanged(
                    CharSequence s,
                    int start,
```

¹ Эти названия переводятся соответственно как «обработчик», «обратный вызов» и «слушатель». — *Примеч. пер.*

```

        int count,
        int after)
    { }
    public void onTextChanged(
        CharSequence s,
        int start,
        int count,
        int after)
    { }
    });
}

void handleTextChange(Editable s) {
    // определенные операции над s, измененным текстом
}
}

```

`MyModel` — это центральная часть вашего приложения. Здесь программа принимает текст, вводимый пользователем, и совершает над этим текстом определенные полезные операции. При создании модель `MyModel` получает `TextBox`, то есть получает поле, в которое пользователь будет вводить текст. На данный момент вы, наверное, уже успели набить руку в синтаксическом разборе такого кода. В своем конструкторе `MyModel` создает новую анонимную реализацию интерфейса `TextWatcher`. Здесь же реализуются три метода, необходимые для этого интерфейса. Два из них, `onTextChanged` и `beforeTextChanged`, ничего не делают. А вот третий метод, `afterTextChanged`, вызывает метод `handleTextChange`, относящийся к `MyModel`.

Вся эта система отлично работает. Возможно, два метода, `beforeTextChanged` и `onTextChanged`, которые не используются в данном конкретном приложении, немного захламляют код. Однако за исключением этого момента в коде очень красиво реализуется разделение функций. Модель `MyModel` не представляет, как `TextView` отображает текст, где он выводится на экране и как в поле попадает тот или иной текст. Маленький промежуточный класс (relay class), анонимный экземпляр `TextWatcher`, просто передает измененный текст между видом и `MyModel`. Реализация модели `MyModel` занята лишь теми событиями, которые происходят при изменении текста.

Этот процесс, в ходе которого скрепляются пользовательский интерфейс и его поведения, часто называется *подключением* (wiring up). Хотя данный процесс достаточно мощный, с ним также связано множество ограничений. Клиентский код — то есть код, который регистрируется на получение обратного вызова, — не может влиять на поведение вызывающего элемента. К тому же клиент не получает никакой информации о состоянии, кроме параметров, передаваемых в вызове. Тип интерфейса (в данном случае `TextWatcher`) представляет собой явный контракт между отправителем обратного вызова и клиентом.

Существует действие, при помощи которого клиент обратного вызова может влиять на вызывающий элемент: клиент может отказать в отклике. Клиентский код должен воспринимать обратный вызов как обычное уведомление, а не как попытку запустить какую-либо длительную встроенную обработку (inline processing).

Если требуется выполнить какой-либо значительный кусок работы (то есть более нескольких сотен команд или любые вызовы, которые могут замедлить функционирование служб, в частности служб файловой системы или сетевых служб), то эти команды нужно поставить в очередь и выполнить позже, возможно, в другом потоке. Мы подробно поговорим о том, как это делается, в подразделе «AsyncTask и поток пользовательского интерфейса» данной главы.

Аналогично служба, которая пытается поддерживать несколько клиентов обратных вызовов, может испытывать нехватку ресурсов процессора, даже если все клиенты работают относительно хорошо. В то время как `addTextChangedListener` поддерживает возможность подписки для нескольких клиентов, многие обратные вызовы, входящие в состав библиотеки Android, поддерживают только один обратный вызов. При работе с такими обратными вызовами (например, `setOnKeyListener`) назначение нового клиента для определенного обратного вызова, направляемого к конкретному объекту, приводит к замене предыдущего клиента, стоявшего на этом месте. Зарегистрированный ранее клиент больше не будет получать уведомлений об обратных вызовах. На самом деле он не получит уведомления даже о том, что больше не является клиентом. С этого момента все уведомления будет получать клиент, зарегистрированный последним. Такое ограничение, существующее в коде, помогает решить очень насущную проблему — таким образом исключается ситуация, в которой обратный вызов поддерживал бы неограниченное количество клиентов. Если в вашем коде приходится распределять уведомления сразу между множеством получателей, вам придется придумать для этого такой способ, который будет безопасен в контексте вашего приложения.

Паттерн обратного вызова повсеместно встречается в библиотеках Android. Поскольку эта идиома известна всем разработчикам Android, вам тоже следует писать свой код по такому принципу. Когда какому-либо классу требуются уведомления об изменениях, происходящих в других классах, — особенно при динамическом изменении ассоциаций во время исполнения, — попробуйте реализовать такое отношение в виде обратного вызова. Если отношение не является динамическим, воспользуйтесь внедрением зависимости (*dependency injection*), то есть примените параметр конструктора и финальное поле, чтобы сделать требуемое отношение постоянным.

Полиморфизм и композиция

При разработке для Android, как и в других объектно-ориентированных средах, применяются полиморфизм и композиция — превосходные инструменты, дополняющие среду разработки. В предыдущем примере были продемонстрированы как полиморфизм, так и композиция. Давайте ненадолго остановимся, заострим внимание на этих концепциях и заново рассмотрим их ценность — теперь в контексте целей разработки.

Анонимный экземпляр `TextWatcher`, передаваемый `addTextChangedListener` в качестве объекта обратного вызова, использует композицию для реализации такого поведения. Экземпляр сам по себе не реализует никаких поведений. Вместо этого он делегирует задачу реализации методу `handleTextChanged`, относящемуся

к модели `MyModel`, предпочитая реализацию отношения has-a реализации самого требуемого поведения. Таким образом, функции ясно и четко разделяются. Если `MyModel` потребуются расширить еще сильнее, например, для применения текста, получаемого из иного источника, то новый источник также будет использовать `handleTextChanged`. Не придется отслеживать код сразу в нескольких отдельных анонимных классах.

На данном примере также демонстрируется использование полиморфизма. Экземпляр, передаваемый методу `addTextChangedListener`, обладает сильной статической типизацией. Это анонимный подтип `TextWatcher`. Данная конкретная реализация — в нашем случае делегирование задачи `handleTextChanged` в `MyModel` — практически наверняка не будет походить ни на одну другую реализацию данного интерфейса. Тем не менее, поскольку это реализация интерфейса `TextWatcher`, он будет иметь статическую типизацию, независимо от того, как именно он выполняет свою задачу. Компилятор может гарантировать, что методу `addTextChangedListener` в `EditText` передаются только такие объекты, которые как минимум предназначены для выполнения поставленной задачи. Реализация не застрахована от ошибок, но как минимум `addTextChangedListener` никогда не получит объект, предназначенный для обработки сетевых событий. В этом и есть суть полиморфизма.

В данном контексте следует упомянуть об одном антипаттерне, так как он встречается очень часто. Многие разработчики считают анонимные классы слишком пространным и неуклюжим способом передачи указателя к функции. Чтобы обойтись без анонимных классов, такие разработчики вообще опускают объект-мессенджер вот так:

```
// !!! Антипаттерн – так поступать не следует.
public class MyModel implements TextWatcher {
    public MyModel(TextView textBox) {
        textBox.addTextChangedListener(this);
    }

    public void afterTextChanged(Editable s) {
        handleTextChanged(s);
    }

    public void beforeTextChanged(
        CharSequence s,
        int start,
        int count,
        int after)
    { }

    public void onTextChanged(
        CharSequence s,
        int start,
        int count,
        int after)
    { }
```

```
void handleTextChanged(Editable s) {
    // Определенные операции над s, измененным текстом.
}
}
```

В некоторых ситуациях такой подход оправдан. Если клиент обратного вызова (в данном случае модель `MyModel`) мал, прост и используется всего в двух контекстах, код получится ясным и уместным.

С другой стороны (об этом подсказывает название `MyModel1`), класс будет широко использоваться при довольно разных обстоятельствах, а в этом случае попытка обойтись без классов-мессенджеров нарушает инкапсуляцию и ограничивает возможности расширения кода. Очевидно, если такую реализацию придется расширить для обработки ввода, получаемого из второго `TextBox`, и задействовать при этом иное поведение, код получится путаным.

Но практически так же порочна и ситуация, иногда называемая *загрязнением интерфейса* (**interface pollution**). Она возникает, когда описанная выше идея выражается чрезмерно сильно. Получается нечто подобное:

```
// !!! Антипаттерн, ВНИМАНИЕ!
public class MyModel
    implements TextWatcher, OnKeyListener, View.OnTouchListener,
        OnFocusChangeListener, Button.OnClickListener
{
    // ....
}
```

Подобный код в определенном смысле кажется заманчиво-элегантным, такой код встречается довольно часто. Но, к сожалению, теперь `MyModel` слишком тесно связана со всеми обрабатываемыми ею событиями.

В большинстве случаев не удастся сформулировать «железных» правил борьбы с загрязнением интерфейса. Как уже было отмечено, существует масса действующего кода, который выглядит именно так. Но все же чем меньше интерфейс, тем меньше в нем возникает сбоев и тем проще его изменять. Когда интерфейс объекта расширяется настолько, что уже явно кажется вам некрасивым, попробуйте разбить его на мелкие фрагменты, которыми будет легко управлять. Для этого применяется композиция.

Расширение классов Android

В то время как обратные вызовы обеспечивают ясные, хорошо продуманные средства для расширения поведений классов, есть и такие ситуации, в которых обратные вызовы не дают нужной степени гибкости. Очевидная проблема, возникающая с паттерном обратных вызовов, заключается в том, что иногда коду приходится взять на себя управление в той или иной ситуации, не предусмотренной разработчиками библиотек. Если служба не определяет обратного вызова, вам придется искать другой способ введения кода в поток управления. Одно из возможных решений — создание подкласса.

¹ Переводится как «моя модель». — *Примеч. ред.*

Некоторые классы в библиотеках Android разрабатывались специально для создания дочерних классов (например, класс `BaseAdapter` из `android.widget` и `AsyncTask`, который мы опишем чуть ниже). Но вообще создание подклассов — не та задача, к которой можно относиться легкомысленно.

Подкласс может полностью заменить поведение любого нефинального метода в своем суперклассе и тем самым совершенно нарушить архитектурный контракт класса. В системе типов Java ничто не мешает, например, подклассу `TextBox` переопределить метод `addTextChangedListener` так, что он будет игнорировать собственный аргумент и не будет уведомлять клиентов обратного вызова об изменениях, происходящих в содержимом текстового поля. (В данном случае можно представить себе, например, реализацию «безопасного» текстового поля, содержимое которого скрыто.)

Такое нарушение контракта — а распознать детали контракта бывает непросто — может порождать ошибки двух разновидностей, и обе эти разновидности довольно нелегко находить. Первая и наиболее очевидная проблема возникает, когда разработчик использует нестандартный подкласс, создающий, например, описанное выше поле для безопасного ввода текста. Предположим, разработчик создает вид, в котором содержится несколько виджетов, и применяет его метод `addTextChangedListener` к каждому виджету, регистрируя их на получение обратных вызовов. Но в ходе тестирования обнаруживается, что некоторые виджеты не работают так, как следует. Программист тратит несколько часов на изучение кода, пока наконец не обнаруживает, что метод просто *ничего не делает*! И вдруг программиста осеняет, он заглядывает в исходный код виджета, чтобы убедиться, что в этом виджете на самом деле нарушен семантический контракт класса. Бррр!

Но еще коварнее оказывается то обстоятельство, что и сам фреймворк Android может изменяться между следующими друг за другом версиями **SDK**. Может измениться реализация метода `addTextChangedListener`. Возможно, код в другой части фреймворка Android попытается вызвать `addTextChangedListener`, рассчитывая на его нормальное поведение. И вдруг, поскольку подкласс переопределяет метод, мы наблюдаем блистательный крах всего приложения!

Можно минимизировать вероятность возникновения подобной проблемы, вызывая сверхреализацию (*superimplementation*) для переопределенного метода таким образом:

```
public void addTextChangedListener(TextWatcher watcher) {  
    // ваш код...  
    super.addTextChangedListener(watcher)  
    // здесь тоже ваш код...  
}
```

Таким образом, вы гарантируете, что ваша реализация дополняет, но не заменяет собой существующее поведение, даже если со временем реализация суперкласса изменится. Существует правило написания кода, продвигаемое в некоторых сообществах разработчиков, которое формулируется как: «Разрабатывай в расчете на расширение». Правило постулирует, что все методы должны быть либо абстрактными, либо финальными. Это правило может показаться драконовским,

но подумайте сами: ведь переопределяющий метод обязательно нарушит семантический контракт объекта и, чтобы не допустить этого, нужно как минимум вызвать сверхреализацию.

Параллелизм в Android

В главе 2 мы уже упоминали о том, что правильно писать программы с параллельным исполнением задач может быть очень непросто. В библиотеках Android предлагаются удобные инструменты, позволяющие сделать параллелизм и проще, и безопаснее.

Обсуждая многопоточные (конкурентные) программы, разработчики часто говорят о том, что написание кода в виде нескольких потоков якобы подразумевает, что эти потоки будут выполняться одновременно, а также о том, что многопоточность в целом ускоряет выполнение программы. Разумеется, все не так просто. Если в распоряжении нет нескольких процессоров, которым можно поручить выполнение потоков, то программа, от которой требуется выполнять множественные, несвязанные, ограниченные по скорости вычислений задачи, будет решать эти задачи с одинаковой скоростью, независимо от того, реализованы они в разных потоках или в одном и том же потоке. В системе с одним процессором многопоточная версия программы может работать даже медленнее, чем однопоточная, из-за того, что дополнительные ресурсы будут тратиться на переключение контекста.

Многопоточные приложения Java появились задолго до того, как средний пользователь смог позволить себе машины с несколькими процессорами, на которых было бы удобно работать с такими приложениями. В мире Android многопоточность приобретает большое значение, хотя на протяжении ближайшего года или около того на многих устройствах будет стоять по одному процессору. Итак, в чем же польза от многопоточности, если не в ускорении работы программы?

Если вам сколько-нибудь долго доводилось заниматься программированием, то, полагаем, вы понимаете абсолютную важность того, что все команды вашего кода должны выполняться в жестком последовательном порядке. Выполнение любой отдельно взятой команды должно так или иначе происходить до выполнения следующей команды. Потоки — это просто явный способ смягчения этого правила. Потоки — это абстракции, которыми разработчики пользуются для того, чтобы можно было создавать и писать код, который будет упорядоченным, логичным и удобочитаемым, даже если воплощенные в нем задачи в реальности не являются упорядоченными.

Параллельное выполнение независимых потоков не приводит к неизбежному усложнению программы, если потоки действительно полностью независимы (например, если один из них работает на вашем компьютере, а другой — на моем). Но когда двум параллельным процессам приходится выполнять одну и ту же задачу, им требуется рандеву (точка для встречи и взаимодействия). Например, возможна ситуация, в которой данные из сети будет нужно отобразить на экране или пользовательский ввод потребуется записать в хранилище данных. Организация рандеву (особенно при рассмотрении этой проблемы в контексте оптимизато-

ров кода, процессоров с конвейерной архитектурой и многоуровневой кэш-памяти) может быть довольно сложной. Такая сложность становится до боли очевидной, когда программа какое-то время работала на одном процессоре, на первый взгляд, без проблем, но как только ее перенесли в многопроцессорную среду — она терпит крах в каких-то странных обстоятельствах, очень осложняющих отладку.

Процесс рандеву, делающий данные или состояние одного потока видимыми для другого потока, обычно называется *публикацией* ссылки. Когда один поток сохраняет состояние так, что это видно из другого потока, мы говорим, что он публикует ссылку на это состояние. Как было указано в главе 2, единственный вариант надежной публикации ссылки требует, чтобы все потоки, ссылающиеся на данные, синхронизировались при использовании этих данных на одном объекте. Любые другие варианты неправильны и ненадежны.

AsyncTask и поток пользовательского интерфейса

Если вам приходилось работать с современными фреймворками пользовательских интерфейсов, то фреймворк пользовательского интерфейса Android покажется вам очень знакомым. Этот интерфейс событийно-управляемый, основан на библиотеке вкладываемых друг в друга (*nestable*) **компонентов**. И что особенно важно в данном случае, этот фреймворк **однопоточный**. Уже много лет назад разработчики обнаружили, что, поскольку графический пользовательский интерфейс должен реагировать на асинхронные события, поступающие из нескольких источников, в многопоточном пользовательском интерфейсе практически невозможно избежать взаимоблокировки. Напротив, один и тот же поток должен обслуживать как ввод (сенсорный экран, клавиатура и т. д.), так и вывод (например, дисплей). Он выполняет запросы, поступающие из этих источников, и делает это последовательно, в том порядке, как получает запросы.

В то время как пользовательский интерфейс работает на одном потоке, практически любое нетривиальное приложение Android будет многопоточным. Например, пользовательский интерфейс должен отвечать на действия пользователя и анимировать дисплей, независимо от того, занят ли в данный момент обработкой входящих данных тот код, который получает данные из сети. Пользовательский интерфейс должен работать быстро, в частности быстро реагировать, и принципиально не может функционировать «с оглядкой» на другие, долговременные процессы. Долговременные процессы должны работать асинхронно.

В Android есть удобный инструмент для реализации асинхронных задач, он называется `AsyncTask`. Он полностью скрывает многие детали потоков, используемых для выполнения задачи.

Рассмотрим предельно упрощенное приложение, инициализирующее игровой движок. Это приложение будет отображать какую-нибудь вводную графику, пока загружается контент. На рис. 3.7 показан простейший пример такого приложения. Когда вы нажимаете кнопку, оно инициализирует игровой уровень, а потом отображает в текстовом поле приветствие.

Это шаблонный (так называемый *boilerplate*) код приложения. В нем не хватает только той части, которая, собственно, инициализирует игру и обновляет текстовое поле:

```

/** AsyncTaskDemo */
public class AsyncTaskDemo extends Activity {

    int mInFlight;

    /** @see android.app.Activity#onCreate(android.os.Bundle) */

```



Рис. 3.7. Простое приложение для инициализации игры

```

@Override
public void onCreate(Bundle state) {
    super.onCreate(state);

    setContentView(R.layout.asyncdemo);

    final View root = findViewById(R.id.root);
    final Drawable bg = root.getBackground();

    final TextView msg = ((TextView) findViewById(R.id.msg));

    final Game game = Game.newGame();

    ((Button) findViewById(R.id.start)).setOnClickListener(
        new View.OnClickListener() {
            @Override public void onClick(View v) {
                // !!! Здесь инициализируется игра!
            }
        });
}

```

А теперь предположим, что в этом примере мы хотим отобразить просто анимированный фон (ползущие точки с рис. 3.7), пока пользователь дожидается инициализации игры. Вот набросок кода, который для этого понадобится:

```

/**
 * Синхронный запрос к удаленной службе.

```

```

* НЕ ИСПОЛЬЗОВАТЬ!!
*/
void initGame(
    View root,
    Drawable bg,
    Game game,
    TextView resp,
    String level)
{
    // Если анимация еще не началась,
    // то нужно сделать так.
    if (0 >= mInFlight++ ) {
        root.setBackgroundResource(R.anim.dots);
        ((AnimationDrawable) root.getBackground()).start();
    }

    // инициализация игры и получение сообщения-приветствия
    String msg = game.initialize(level);

    // Если это последняя работающая инициализация,
    // удалить и очистить анимацию.
    if (0 >= --mInFlight) {
        ((AnimationDrawable) root.getBackground()).stop();
        root.setBackgroundDrawable(bg);
    }

    resp.setText(msg);
}

```

Тут все очень просто. Пользователь может несколько раз нажать кнопку запуска игры, поэтому одновременно может начаться несколько инициализаций. Если фон-заставка еще не отображается, покажите его, помня при этом, что сейчас запускается еще одна игра. Потом сделайте медленный вызов к инициализатору движка игры. Когда инициализация игры завершится, выполните очистку. Если закончилась инициализация последней запущенной игры, уберите вводную заставку. Наконец, отобразите в текстовом поле приветствие.

Этот код уже очень близок к тому варианту, который нужен, чтобы приложение-пример работало правильно. Но он содержит один очень пагубный недостаток: код блокирует поток пользовательского интерфейса на все то время, пока длится вызов к `game.initialize`. Такая ситуация чревата всяческими неприятными эффектами.

Наиболее явный подобный эффект заключается в том, что не будет работать фоновая анимация. Даже хотя логика настройки и запуска анимации почти правильна, из кода вполне ясно следует, что в пользовательском интерфейсе не должно происходить ровно ничего — до тех пор пока не завершится вызов удаленной процедуры.

И это еще полбеды. К тому же фреймворк **Android** отслеживает потоки пользовательского интерфейса приложения, чтобы помешать неисправным или вредоносным программам вызвать зависание устройства. Если приложение слишком

долго не отвечает на ввод, фреймворк приостанавливает его исполнение, уведомляет пользователя о возникшей проблеме и предлагает возможность принудительно закрыть это приложение. Если построить и запустить это приложение, как показано выше, в частности реализовать `initGame` именно так, как в примере (попробуйте, пример поучительный), то, как только вы нажмете кнопку **Send Request** (Отправить запрос), интерфейс зависнет. Если нажмете еще пару раз, то увидите предупреждение, примерно как на рис. 3.8.

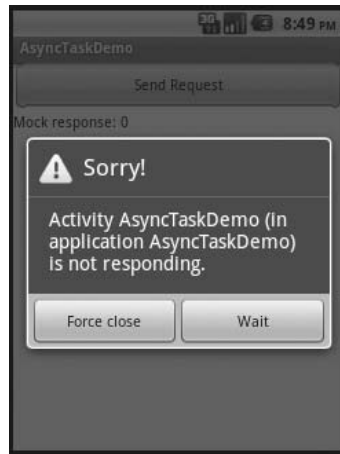


Рис. 3.8. Приложение, которое не отвечает

Вот здесь нам и пригодится `AsyncTask`! Этот класс в **Android** является относительно безопасным, мощным и простым способом выполнения фоновых задач. Вот новая реализация `initGame` в виде `AsyncTask`:

```
private final class AsyncInitGame
    extends AsyncTask<String, Void, String>
{
    private final View root;
    private final Game game;
    private final TextView message;
    private final Drawable bg;

    public AsyncInitGame(
        View root,
        Drawable bg,
        Game game,
        TextView msg)
    {
        this.root = root;
        this.bg = bg;
        this.game = game;
        this.message = msg;
    }
}
```

```

// запуск потока пользовательского интерфейса
@Override protected void onPreExecute() {
    if (0 >= mInFlight++) {
        root.setBackgroundResource(R.anim.dots);
        ((AnimationDrawable) root.getBackground()).start();
    }
}

// запуск потока пользовательского интерфейса
@Override protected void onPostExecute(String msg) {
    if (0 >= --mInFlight) {
        ((AnimationDrawable) root.getBackground()).stop();
        root.setBackgroundDrawable(bg);
    }

    message.setText(msg);
}

// запуск потока пользовательского интерфейса
@Override protected String doInBackground(String... args) {
    return ((1 != args.length) || (null == args[0]))
        ? null
        : game.initialize(args[0]);
}
}

```

Данный код практически идентичен коду из первого примера. Он разделен на три метода, исполняющих примерно тот же код, что и выше, в том же порядке, что и в `initGame`.

Этот `AsyncTask` создан в потоке пользовательского интерфейса. Когда поток пользовательского интерфейса активирует метод `execute`, относящийся к задаче, то сначала к потоку пользовательского интерфейса применяется метод `onPreExecute`. Таким образом, задача может инициализировать себя и свое окружение — в данном случае установить фоновую анимацию. Потом `AsyncTask` ставит себя в очередь для параллельного выполнения метода `doInBackground`. Когда наконец выполнение `doInBackground` завершается, фоновый поток удаляется, а метод `onPostExecute` активируется, опять же в потоке пользовательского интерфейса.

Если допустить, что данная реализация `AsyncTask` является правильной, то слушателю щелчков (`click listener`) требуется просто создать экземпляр и активировать его, вот так:

```

((Button) findViewById(R.id.start)).setOnClickListener(
    new View.OnClickListener() {
        @Override public void onClick(View v) {
            new AsyncInitGame(
                root,
                bg,
                game,
                msg)
                .execute("basic");
        }
    });

```

Действительно, код `AsyncInitGame` теперь является полным, правильным и надежным. Рассмотрим его подробнее.

Для начала отметим, что базовый класс `AsyncTask` является абстрактным. Единственный способ воспользоваться им — это создать подкласс, специально предназначенный для выполнения конкретной задачи (в виде отношения `is-a`, а не `has-a`). Как правило, подкласс будет простым, анонимным и будет определять всего несколько методов. С учетом хорошего стиля и разделения функций проблем, упоминавшихся в главе 2, нужно, чтобы подкласс был небольшим и делегировал реализацию тем классам, которые отвечают соответственно за пользовательский интерфейс и за асинхронную задачу. В данном примере, в частности, `doInBackground` — это просто посредник класса `Game`.

Вообще, `AsyncTask` принимает набор параметров и возвращает результат. Поскольку параметры должны передаваться между потоками, а результат — возвращаться между потоками, необходимо квитирование установления связи (так называемое рукопожатие), обеспечивающее безопасность потоков. `AsyncTask` активируется при вызове его метода `execute` с некоторыми параметрами. Эти параметры в конечном счете передаются методу `doInBackground`, работающему в фоновом потоке. Делается это посредством механизма `AsyncTask`. В свою очередь, `doInBackground` выдает результат. Механизм `AsyncTask` возвращает этот результат, передавая его в качестве аргумента `doPostExecute`, работающему в том же потоке, что и исходный `execute`. На рис. 3.9 показан поток данных.

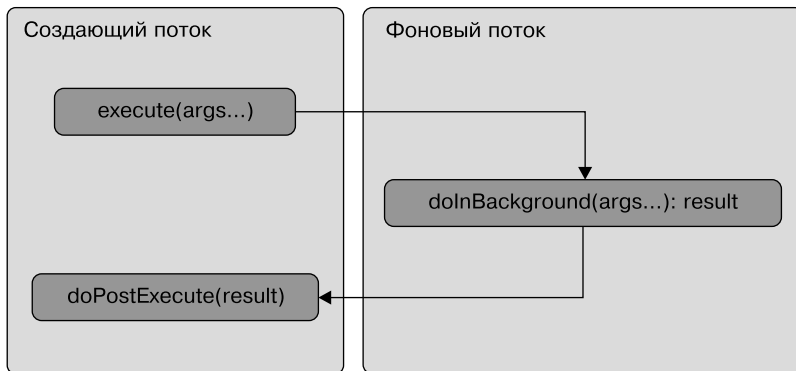


Рис. 3.9. Поток данных в `AsyncTask`

`AsyncTask` обеспечивает в этом фрагменте данных не только безопасность потоков, но и безопасность типов. `AsyncTask` — это классический пример шаблона с безопасностью типов. Абстрактный базовый класс (`AsyncTask`) использует дженерики Java, позволяющие реализациям указывать типы для параметров и результата задачи.

При определении конкретного подкласса `AsyncTask` вы указываете точные типы `Params`, `Progress` и `Result` — это переменные типов в определении `AsyncTask`. Первый и последний из типов этих переменных (`Params` и `Result`) — это соответственно типы параметров и результата задачи. Средний тип переменных мы также вскоре обсудим.

Конкретный тип, связываемый с Params, — это тип параметров, передаваемых в execute, и, следовательно, тип параметров, которые передаются в doInBackground. Аналогично конкретный тип, связанный с Result, — это тип значения, возвращаемого из doInBackground, а значит, тип параметра, который передается в onPostExecuteExecute.

Синтаксический разбор всего этого несколько сложен, и первый пример, AsyncInitGame, не особенно полезен, так как в нем и параметр, и результат относятся к одному и тому же типу — String. Ниже приведена пара примеров, в которых типы параметров и результатов отличаются. Они лучше иллюстрируют использование переменных универсальных типов:

```
public class AsyncDBReq
    extends AsyncTask<PreparedStatement, Void, ResultSet>
{
    @Override
    protected ResultSet doInBackground(PreparedStatement... q) {
        // реализация...
    }

    @Override
    protected void onPostExecute(ResultSet result) {
        // реализация...
    }
}

public class AsyncHttpReq
    extends AsyncTask<HttpRequest, Void, HttpResponse>
{
    @Override
    protected HttpResponse doInBackground(HttpRequest... req) {
        // реализация...
    }

    @Override
    protected void onPostExecute(HttpResponse result) {
        // реализация...
    }
}
```

В первом примере аргументом метода execute экземпляра AsyncDBReq будет одна или несколько переменных PreparedStatement. Реализация метода doInBackground для экземпляра AsyncDBReq примет параметры этой переменной PreparedStatement в качестве своих аргументов и возвратит ResultSet. Экземпляр метода onPostExecute примет этот ResultSet в качестве параметра и использует его по назначению.

Аналогично во втором примере вызываемый метод execute экземпляра AsyncHttpReq примет одну или несколько переменных HttpRequest. Метод doInBackground примет эти запросы в качестве параметров и возвратит HttpResponse. В свою очередь, onPostExecute обработает HttpResponse.



Обратите внимание на то, что один экземпляр `AsyncTask` можно запустить всего один раз. При повторном вызове метода `execute` применительно к задаче будет выдано исключение `IllegalStateException`. Для каждой активации задачи требуется новый экземпляр.

При том, насколько `AsyncTask` упрощает параллельную обработку, его контракт накладывает на программу серьезные ограничения, не поддающиеся автоматической проверке. Необходимо прилагать все усилия, чтобы соблюдать эти ограничения! Нарушения ограничений вызывают ошибки именно такого рода, которые упоминались выше: перемежающиеся отказы системы, причины которых очень сложно найти.

Наиболее очевидное из упомянутых ограничений заключается в том, что, поскольку метод `doInBackground` работает в отдельном потоке, он должен создавать только такие ссылки на переменные, наследуемые в его область видимости, какие обеспечивают безопасность типов.

Вот пример распространенной ошибки:

```
// ...какой-то класс

int mCount;

public void initButton1( Button button) {
    mCount = 0;
    button.setOnClickListener(
        new View.OnClickListener() {
            @SuppressWarnings("unchecked")
            @Override public void onClick(View v) {
                new AsyncTask<Void, Void, Void>() {
                    @Override
                    protected Void doInBackground(Void... args) {
                        mCount++; // !!! БЕЗОПАСНОСТЬ ПОТОКОВ НЕ СОБЛЮДЕНА!
                        return null;
                    }
                }.execute();
            }
        });
}
```

Хотя здесь совершенно не из-за чего выводить предупреждения о проблеме — нет ошибок компиляции, предупреждений времени исполнения (*runtime warning*) и, кроме того, при возникновении ошибки не происходит немедленного краха системы, — этот код совершенно неправильный. К переменной `mCount` получают доступ два различных потока, между которыми не происходит синхронизации.

С учетом сказанного вы можете удивиться тому, что доступ к `mInFlight` не синхронизирован с доступом к `AsyncTaskDemo`. Вот здесь как раз все нормально. Контракт `AsyncTask` гарантирует, что методы `onPreExecute` и `onPostExecute` будут запускаться в том же потоке, из которого был вызван `execute`. В отличие от `mCount`, доступ к `mInFlight` осуществляется из единственного потока — следовательно, синхронизация при этом не требуется.

Вероятно, наиболее пагубный вариант возникновения таких проблем с конкуренцией, о которых мы только что рассказали, — это удержание ссылки на параметр.

Например, следующий код является неправильным (вы видите почему?):

```
public void initButton(
    Button button,
    final Map<String, String> vals)
{
    button.setOnClickListener(
        new View.OnClickListener() {
            @Override public void onClick(View v) {
                new AsyncTask<Map<String, String>, Void, Void>() {
                    @Override
                    protected Void doInBackground(
                        Map<String, String>... params)
                    {
                        // реализация использует карту params
                    }
                }.execute(vals);
                vals.clear(); // !!! БЕЗОПАСНОСТЬ ПОТОКОВ НЕ СОБЛЮДЕНА!!!
            }
        });
}
```

Проблема довольно тонкая. Вы, возможно, заметили, что на `vals` (аргумент для `initButton`) ставятся параллельные (конкурентные) ссылки. И это происходит без синхронизации — вот в чем дело! `Vals` передается к `AsyncTask` как аргумент `execute`. Это происходит при активации задачи. Фреймворк `AsyncTask` может гарантировать, что эта ссылка будет правильно опубликована в фоновом потоке, когда вызывается `doInBackground`. При этом ничего не удастся сделать со ссылкой на `vals`, которая удерживается, а позднее применяется в методе `initButton`. Вызов к `vals.clear` изменяет состояние, которое используется в другом потоке без синхронизации. Следовательно, безопасность потоков нарушается.

Чтобы решить такую проблему, лучше всего убедиться, что аргументы для `AsyncTask` являются постоянными. Если их нельзя изменить (как `String` или `Integer`) или они являются POJO (Plain Old Java Objects, «старые добрые объекты Java»), содержащими только финальные поля, то безопасность потоков соблюдается и дальнейшей обработки не требуется. Если к `AsyncTask` передается объект, который может быть изменен, то единственный способ гарантировать безопасность потоков — убедиться, что ссылку на него удерживает только `AsyncTask`.

Поскольку в предыдущем примере (см. рис. 6.1) параметр `vals` передается методу (`initButton`), мы совершенно не можем гарантировать, что на этот параметр не будет «висячих» ссылок. Даже после удаления вызова `vals.clear` нельзя быть уверенными, что код станет правильным, так как сторона, вызывающая `initButton`, может сохранить ссылку на карту, которая в конечном итоге будет передана как параметр `vals`. Единственный способ добиться правильности этого кода — сделать полную (глубокую) копию карты и всех содержащихся в ней объектов!

Разработчики, знакомые с пакетом Java Collections, могут возразить, что можно обойтись и без полной глубокой копии параметров карты, а применить в качестве обертки `unmodifiableMap`:

```
public void initButton(
    Button button,
    final Map<String, String> vals)
{
    button.setOnClickListener(
        new View.OnClickListener() {
            @Override public void onClick(View v) {
                new AsyncTask<Map<String, String>, Void, Void>() {
                    @Override
                    protected Void doInBackground(
                        Map<String, String>... params)
                    {
                        // реализация использует карту params
                    }
                }.execute(Collections.unmodifiableMap(vals));
                vals.clear(); // !!! БЕЗОПАСНОСТЬ ПОТОКОВ ПО-ПРЕЖНЕМУ
                           // НЕ СОБЛЮДЕНА!!!
            }
        });
}
```

К сожалению, код *остался* неправильным. `Collections.unmodifiableMap` обеспечивает постоянство вида той карты, которая в ней обернута. Однако такая обертка не закрывает доступ к объекту таким процессам, которые удерживают ссылку на оригинальный, изменяемый вариант такого объекта — и процессы, обладающие такими ссылками, действительно могут изменить его когда угодно. В предыдущем примере, хотя `AsyncTask` и не может изменить значение карты, переданной ему в методе `execute`, метод `onClick` все равно изменяет карту со ссылкой на `vals`. В то же время фоновый поток использует эту карту без синхронизации. Опа!

При работе с `AsyncTask` вы рискуете наступить еще на одни грабли. Не забывайте, что активность, порождающая задачу, имеет жизненный цикл. Если задача сохраняет указатель на создавшую ее активность, а пользователь во время выполнения задачи звонит по телефону, то задача может остаться с указателем на уничтоженную активность. Такая ситуация особенно легко возникает с анонимными классами, сохраняющими неявный указатель на создающий их класс. `AsyncTask` лучше всего подходит для выполнения очень кратковременных задач, длящихся не более нескольких секунд. Но такой механизм не следует использовать с процессами, которые могут длиться минуту или дольше.

Завершая этот подраздел, отметим, что в примере есть еще один метод, используемый `AsyncTask`: `onProgressUpdate`. Этот метод нужен для того, чтобы долгоиграющие задачи могли периодически отправлять в пользовательский интерфейс статусные сообщения и эти операции были безопасными. Здесь, возможно, будет целесообразно реализовать индикатор загрузки, показывающий, когда завершится инициализация игры:

```

public class AsyncTaskDemoWithProgress extends Activity {

    private final class AsyncInit
        extends AsyncTask<String, Integer, String>
        implements Game.InitProgressListener
    {
        private final View root;
        private final Game game;
        private final TextView message;
        private final Drawable bg;

        public AsyncInit(
            View root,
            Drawable bg,
            Game game,
            TextView msg)
        {
            this.root = root;
            this.bg = bg;
            this.game = game;
            this.message = msg;
        }

        // работает в потоке пользовательского интерфейса
        @Override protected void onPreExecute() {
            if (0 >= mInFlight++) {
                root.setBackgroundResource(R.anim.dots);
                ((AnimationDrawable) root.getBackground()).start();
            }
        }

        // работает в потоке пользовательского интерфейса
        @Override protected void onPostExecute(String msg) {
            if (0 >= --mInFlight) {
                ((AnimationDrawable) root.getBackground()).stop();
                root.setBackgroundDrawable(bg);
            }

            message.setText(msg);
        }

        // работает в собственном потоке
        @Override protected String doInBackground(String... args) {
            return ((1 != args.length) || (null == args[0]))
                ? null
                : game.initialize(args[0], this);
        }

        // работает в потоке пользовательского интерфейса

```

```

@Override protected void onProgressUpdate(Integer... vals) {
    updateProgressBar(vals[0].intValue());
}

// работает в потоке пользовательского интерфейса
@Override public void onInitProgress(int pctComplete) {
    publishProgress(Integer.valueOf(pctComplete));
}
}

int mInFlight;
int mComplete;

/** @see android.app.Activity#onCreate(android.os.Bundle) */
@Override
public void onCreate(Bundle state) {
    super.onCreate(state);

    setContentView(R.layout.asyncdemoprogress);

    final View root = findViewById(R.id.root);
    final Drawable bg = root.getBackground();

    final TextView msg = ((TextView) findViewById(R.id.msg));

    final Game game = Game.newGame();

    ((Button) findViewById(R.id.start)).setOnClickListener(
        new View.OnClickListener() {
            @Override public void onClick(View v) {
                mComplete = 0;
                new AsyncInit(
                    root,
                    bg,
                    game,
                    msg)
                    .execute("basic");
            }
        });
}

void updateProgressBar(int progress) {
    int p = progress;
    if (mComplete < p) {
        mComplete = p;
        ((ProgressBar) findViewById(R.id.progress))
            .setProgress(p);
    }
}
}

```

В данном примере предполагается, что при инициализации игры в качестве аргумента принимается `Game.OnProgressListener`. В процессе инициализации периодически вызывается метод `onInitProgress`, который должен уведомить, какая часть работы выполнена. Затем в этом примере `onInitProgress` будет вызван «из-под» `doInBackground`, если смотреть по дереву вызовов, то есть вызов будет происходить в фоновом потоке. Если бы `onInitProgress` вызывал `AsyncTaskDemoWithProgress.updateProgressBar` напрямую, то последующий вызов `bar.setStatus` также происходил бы в фоновом потоке, нарушая правило, в соответствии с которым только поток пользовательского интерфейса может изменять объекты `View`. Получилось бы подобное исключение:

```
11-30 02:42:37.471: ERROR/AndroidRuntime(162):  
    android.view.ViewRoot$CalledFromWrongThreadException:  
    Only the original thread that created a view hierarchy can touch its views1.
```

Чтобы правильно опубликовать информацию о ходе загрузки и передать ее обратно в поток пользовательского интерфейса, `onInitProgress` вызывает метод `publishProgress`, относящийся к `AsyncTask`. В свою очередь, `AsyncTask` обрабатывает детали диспетчирования `publishProgress` в потоке пользовательского интерфейса, так что `onProgressUpdate` может свободно использовать методы `View`.

Завершим детальное рассмотрение `AsyncTask` суммированием основных его аспектов.

- Пользовательский интерфейс Android является однопоточным. Чтобы правильно обращаться с ним, разработчик должен хорошо ориентироваться в работе с идиомой постановки задач в очередь.
- Для сохранения жизнеспособности пользовательского интерфейса в его потоке не следует выполнять задачи, требующие на обработку более пары миллисекунд и состоящие более чем из нескольких сотен инструкций.
- Конкурентное программирование — действительно сложная вещь. Оно очень легко может пойти в неправильном направлении, а проверять программу на наличие ошибок довольно сложно.
- `AsyncTask` — удобный инструмент для выполнения небольших асинхронных задач. Просто не забывайте, что метод `doInBackground` работает в другом потоке! Он не должен записывать никакое состояние, видимое из другого потока, или считывать состояние, доступное для изменений из другого потока. Это же касается его параметров.
- Постоянные (immutable) объекты — важный инструмент для передачи информации между параллельными потоками.

Потоки в процессе Android

`AsyncTask` и `ContentProvider` вместе образуют очень мощную идиому, которая может быть адаптирована к разнообразным видам распространенных архитектур

¹ Только тот поток, который первоначально создал иерархию видов, может иметь доступ к этим видам. — *Примеч. пер.*

приложения. Почти любой паттерн «Модель-вид-контроллер» (MVC), в котором вид опрашивает модель, может (и, пожалуй, должен) реализовываться таким образом. Если архитектура приложения требует, чтобы модель отправляла изменения в вид, или в данной архитектуре модель является долгоживущей или работает непрерывно, одного только `AsyncTask` может быть недостаточно.

Вспомним основополагающее правило обмена данными между потоками, которое мы сформулировали еще в подразделе «Синхронизация и потоковая безопасность» раздела «Идиомы программирования в Java» главы 2. В самом общем виде это правило очень обременительное. Но исследование `AsyncTask`, сделанное нами в предыдущем подразделе, демонстрирует идиому, которая упрощает правильную координацию конкурентных задач в Android: сложная работа, заключающаяся в публикации состояния одного потока и предоставлении этой информации другому потоку, полностью скрыта в реализации класса-шаблона. В то же время в предыдущем подразделе мы дополнительно заострили внимание на некоторых ловушках, связанных с конкурентным программированием, в которые часто попадают невнимательные разработчики. Существуют и другие безопасные идиомы, способные упростить решение некоторых проблем, связанных с конкурентным программированием. Одна из этих идиом — широко применяемая в Java вообще — внедрена во фреймворке Android. Иногда ее называют *привязкой к потоку* (thread confinement).

Предположим, что поток `DBMinder` создает объект и изменяет его в течение некоторого времени. Завершив работу, он должен передать объект другому потоку, `DBViewer`, для дальнейшей обработки. Для этого `DBMinder` и `DBViewer`, использующие привязку к потоку, должны совместно использовать точку сброса объекта (drop point) и связанную с ней блокировку. Процесс строится таким образом.

1. `DBMinder` захватывает блокировку и сохраняет ссылку на объект, находящийся в точке сброса.
2. `DBMinder` *уничтожает все ссылки на объект!*
3. `DBMinder` снимает блокировку.
4. `DBViewer` захватывает блокировку и обнаруживает в точке сброса ссылку на объект.
5. `DBViewer` восстанавливает ссылку из точки сброса, а потом очищает эту точку.
6. `DBViewer` снимает блокировку.

Такой процесс работает с любым объектом, независимо от того, гарантируется ли в самом этом объекте безопасность потоков. Это обусловлено тем, что поле сброса является единственным состоянием, которое когда-либо в ходе работы программы совместно используется несколькими потоками. В нашем случае оба потока корректно захватывают единственную блокировку, прежде чем получить доступ к точке сброса. Когда `DBMinder` завершает работу с объектом, он передает объект к `DBViewer` и не сохраняет никаких ссылок на этот объект: состояние переданного объекта никогда не используется совместно несколькими потоками.

Привязка к потоку — это удивительно мощный инструмент. Обычно реализации используют в качестве общей точки сброса упорядоченную очередь задач. Несколь-

ко потоков могут одновременно требовать захвата блокировки, но удержание блокировки одним потоком длится не дольше, чем необходимо для постановки задачи в очередь. Один или несколько потоков-исполнителей захватывают блокировку очереди, чтобы взять из нее задачи для выполнения. Подобный паттерн иногда называют «модель “поставщик/потребитель”» (producer/consumer model). Пока фрагмент работы может выполняться полностью в контексте потока-исполнителя, запросившего этот фрагмент, дальнейшей синхронизации не требуется. Внимательно изучив реализацию `AsyncTask`, вы поймете, как именно все это работает.

Привязка к потоку настолько полезна, что в Android она внедрена во фреймворк системы в виде класса под названием `Looper`. При инициализации `Looper` поток Java превращается в очередь задач. Весь свой жизненный цикл он тратит на взятие задач из локальной очереди и их выполнение. Другие потоки ставят задачи в очередь для обработки со стороны инициализированного процесса, как показано выше. Поскольку поток, ставящий задачи в очередь, удаляет все ссылки на объекты, которые в эту очередь ставит, оба потока можно программировать, не волнуясь о проблемах, связанных с конкурентной обработкой. Кроме того что такой подход радикально упрощает корректное написание программ, он еще и устраняет неэффективное использование ресурсов, которое может быть обусловлено чрезмерными затратами на синхронизацию.

Возможно, это описание очереди задач напоминает вам о той концепции, которую мы затрагивали в начале данной главы? Однопоточный событийно-управляемый пользовательский интерфейс Android — это, в сущности, `Looper`. При запуске `Context` система выполняет некоторую регистрацию, а потом инициализирует поток запуска как `Looper`. Этот поток становится основным потоком службы и потоком пользовательского интерфейса для активности. В активности фреймворк пользовательского интерфейса сохраняет ссылку на этот поток, и его очередь задач становится очередью событий пользовательского интерфейса: все внешние драйверы, экран, клавиатура, обработчик вызовов и т. д. направляют задачи в эту очередь.

Вторая половина `Looper` называется `Handler`. `Handler`, создаваемый в потоке `Looper`, предоставляет портал для очереди `Looper`. Когда поток `Looper` хочет позволить какому-нибудь другому потоку, стоящему в очереди, получить доступ к очереди задач, он создает новый `Handler` и передает его другому потоку. Чтобы упростить работу с `Handler`, используется несколько кодовых комбинаций: `View.post(Runnable)`, `View.postDelayed(Runnable, long)` и `Activity.runOnUiThread(Runnable)`.

В инструментарии Android имеется еще одна удобная и мощная парадигма, предназначенная для межпроцессной коммуникации и разделения работы: `ContentProvider`, об этом мы поговорим в главе 12. Прежде чем строить собственную архитектуру, основанную на низкоуровневых компонентах, рассматриваемых в этом разделе, подумайте, не достаточно ли для решения стоящей перед вами задачи обычного поставщика содержимого. Поставщики содержимого — гибкие и расширяемые компоненты, при этом параллельное исполнение с их применением протекает достаточно быстро во всех приложениях, кроме большинства программ, чувствительных к задержкам (time-sensitive).

Сериализация

Сериализация — это процесс преобразования данных из быстрых, эффективных, внутрисистемных представлений в такую форму, которая обеспечивает долговременное хранение данных или передачу их по сети. Преобразование данных в сериализованную форму часто называется *маршалингом*. Обратное преобразование данных в живое представление, существующее в оперативной памяти, называется *десериализацией* или *демаршалингом*.

Точный принцип сериализации данных зависит от той причины, по которой проводится сериализация. Например, данные, сериализуемые для передачи по сети, бывает невозможно считывать, пока они в пути. С другой стороны, информация, сериализуемая для хранения в базе данных, будет гораздо полезнее, если ее представление допускает SQL-запросы, которые легко составлять и понимать. В первом случае формат сериализации может быть двоичным. Во втором случае это, вероятно, будет аннотированный текст (labeled text).

В среде Android существует четыре распространенных варианта использования сериализации.

- Управление жизненным циклом — в отличие от более крупных устройств, например ноутбуков и настольных компьютеров, устройство Android, как правило, не позволяет оперативно переместить приложение во вспомогательный своп-файл памяти, когда это приложение становится неактивным. В такой ситуации фреймворк предоставляет объект, называемый Bundle. Когда работа приложения приостанавливается, оно записывает свое состояние в Bundle. При восстановлении приложения фреймворк Android гарантирует предоставление копии того же Bundle в ходе инициализации. Приложение должно быть способно сериализовать все данные, которые ему необходимо сохранить на время приостановления работы, и уметь сохранить сериализованную версию данных в Bundle.
- Долговременное хранение — кроме непосредственного состояния приложения, хранимого в Bundle, большинство программ также управляет тем или иным долговременным хранилищем данных. Обычно в качестве такого хранилища выступает база данных SQLite, обернутая в ContentProvider. Приложения должны преобразовывать состояния внутрисистемного представления данных объекта и представления этих же объектов в базе данных. В крупных системах такой процесс называется *объектно-реляционным отображением* (ORM, object-relational mapping). Эта технология поддерживается такими фреймворками, как Hibernate и iBATIS. Локальное хранилище данных Android построено проще, к тому же оно легчевеснее. Оно описано в главе 9.
- Локальная межпроцессная коммуникация — во фреймворке Android пропагандируется архитектура, при которой крупные монолитные приложения подразделяются на более мелкие компоненты: пользовательские интерфейсы, поставщики содержимого и сервисы (службы). Эти компоненты не имеют доступа к пространству друг друга, занимаемому в памяти, и, передавая информацию друг другу, должны посылать ее через границы процессов в виде сериализованных сообщений. Для этого в Android есть отлично приспособленный инструмент — язык AIDL.

- Сетевая коммуникация — это одна из изюминок работы с мобильными устройствами. Возможность подключаться к Интернету и пользоваться невероятным множеством сервисов, которые там имеются, — это суть Android. Приложения должны быть приспособлены к взаимодействию с протоколами, требуемыми для работы с внешними сервисами. При этом необходим механизм для преобразования внутренней информации в запросы к таким сервисам и последующего обратного преобразования полученного ответа.

В следующих подразделах описываются различные классы, предоставляемые для достижения этих целей.

Сериализация в Java

Java определяет фреймворк для сериализации через интерфейс-маркер (marker interface) `Serializable` и два типа сериализации — `ObjectOutputStream` и `ObjectInputStream`. Поскольку сериализация в Java обычно «работает и все», даже опытные программисты могут не осознавать, насколько сложен этот механизм.

Определенно обсуждение сериализации выходит за рамки этой книги. Если вам интересен этот вопрос, вы можете изучить различные источники, посвященные ему.

Сериализация Java поддерживается в Android. Например, тип `Bundle` обладает двумя методами — `putSerializable` и `getSerializable`, которые соответственно добавляют объект `Serializable` в `Bundle` и получают его оттуда. Рассмотрим пример:

```
public class JSerialize extends Activity {
    public static final String APP_STATE
        = "com.oreilly.android.app.state";
    private static class AppState implements Serializable {
        // определения, методы-получатели, методы-установщики
        // для параметров состояния приложения
        // ...
    }

    private AppState applicationState;

    /** Вызывается при первом создании активности */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        applicationState = (null == savedInstanceState)
            ? new AppState(/* ... */)
            : (AppState) savedInstanceState.getSerializable(APP_STATE);

        setContentView(R.layout.main);

        // ...
    }
}
```

```

/**
 * @see android.app.Activity#onSaveInstanceState(android.os.Bundle)
 */
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putSerializable(APP_STATE, applicationState);
}
}

```

В данном примере приложение сохраняет определенную глобальную информацию о состоянии (например, список недавно использовавшихся элементов) в виде объекта `Serializable`. Когда исполнение активности `JSerialize` приостанавливается и другая активность может заменить ее в памяти, фреймворк Android активирует относящийся к `JSerialize` метод обратного вызова `onSaveInstanceState`, передавая объект `Bundle`. Метод обратного вызова использует `Bundle.putSerializable` для сохранения состояния объекта в `Bundle`. При возобновлении работы `JSerialize` метод `onCreate` получает состояние из `Bundle`, пользуясь `getSerializable`.

Объект `Parcelable` для передачи данных

Хотя фреймворк Android поддерживает сериализацию Java, это обычно не лучший способ маршалинга состояния программы. Собственный внутренний протокол Android, предназначенный для сериализации, называется `Parcelable`. Он легковесен, отлично оптимизирован, а работать с ним лишь немногим сложнее, чем с сериализацией. Это наилучший способ организации локальной межпроцессной коммуникации. Существуют причины (они будут совершенно очевидны, когда мы вернемся к рассмотрению объектов `Parcelable` в подразделе «Классы, поддерживающие сериализацию» далее), по которым эти объекты нельзя хранить дольше, чем длится жизненный цикл приложения. Объекты `Parcelable` — не лучший вариант для того, чтобы выполнять маршалинг состояния, например, в базу данных или файл.

Ниже показан очень простой объект, содержащий состояние. Рассмотрим, как сделать его `parcelable`:

```

public class SimpleParcelable {
    public enum State { BEGIN, MIDDLE, END; }

    private State state;
    private Date date;

    State getState() { return state; }
    void setState(State state) { this.state = state; }

    Date getDate() { return date; }
    void setDate(Date date) { this.date = date; }
}

```

Для того чтобы быть `parcelable`, объект должен выполнять три следующих требования:

- он должен реализовывать интерфейс `Parcelable`;
- у него должен быть инструмент маршалинга, реализация метода интерфейса `writeToParcel`;
- у него должен быть инструмент демаршалинга, переменная `public static final` с именем `CREATOR`, содержащая ссылку на реализацию `Parcelable.Creator`.

Метод интерфейса `writeToParcel` — это инструмент маршалинга. Он вызывается применительно к объекту, когда этот объект необходимо сериализовать в `Parcel`. Задача инструмента маршалинга — записать всю информацию, необходимую для восстановления состояния объекта в получившемся `Parcel`. Как правило, под этим понимается выражение состояния объекта при помощи шести примитивных типов данных: `byte`, `double`, `int`, `float`, `long` и `String`.

Ниже показан уже рассмотренный простой объект, но с добавлением инструмента маршалинга:

```
public class SimpleParcelable implements Parcelable {
    public enum State { BEGIN, MIDDLE, END; }

    private static final Map<State, String> marshalState;
    static {
        Map<State, String> m = new HashMap<State, String>();
        m.put(State.BEGIN, "begin");
        m.put(State.MIDDLE, "middle");
        m.put(State.END, "end");
        marshalState = Collections.unmodifiableMap(m);
    }

    private State state;
    private Date date;

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        // преобразование даты в тип long
        dest.writeLong(
            (null == date)
            ? -1
            : date.getTime());

        dest.writeString(
            (null == state)
            ? ""
            : marshalState.get(state));
    }

    State getState() { return state; }
    void setState(State state) { this.state = state; }

    Date getDate() { return date; }
    void setDate(Date date) { this.date = date; }
}
```

Разумеется, точная реализация `writeToParcel` будет зависеть от содержимого сериализуемого объекта. В данном случае объект `SimpleParcelable` имеет два компонента состояния, их оба он записывает в получаемый `Parcel`.

Выбор вариантов представления простейших типов данных обычно требует лишь немного изобретательности. Например, `Date` в приведенном выше примере легко представить в виде времени, истекшего с момента окончания прошлого тысячелетия.

При выборе сериализованного представления обязательно следует учитывать изменения, которые могут произойти в данных позднее. Разумеется, в данном примере будет гораздо проще представить `state` как `int`, для получения значения которого вызывается `state.ordinal`. Однако при таком подходе будет достаточно сложно обеспечивать последующую совместимость объекта с более новыми версиями (**forward compatibility**). **Предположим, нам требуется добавить в определенной точке новое состояние, `State.INIT`, наступающее раньше состояния `State.BEGIN`.** Такое простейшее изменение сделает новые версии объектов совершенно несовместимыми со старыми. Подобный, немного более слабый аргумент возникает при использовании `state.toString` для создания маршалируемого представления состояния.

Отображение объекта на его представления в `Parcel` происходит в рамках отдельно взятого процесса сериализации. Сериализация — это не обязательно неотъемлемый атрибут самого объекта. Вполне возможно, что конкретный объект будет иметь абсолютно разные представления в зависимости от того, какой механизм производит его сериализацию. Чтобы проиллюстрировать этот принцип (правда, это уже излишняя мера, раз уж тип `State` уже определен локально), необходимо отметить, что карта, используемая для маршалинга `state`, — это независимый и явно определяемый член класса `parcelable`.

`SimpleParcelable`, как было показано выше, компилируется без ошибок. Он даже может маршализоваться в пакет данных. Но на данном этапе работы получить его обратно мы не можем. Для этого нам потребуется инструмент демаршалинга:

```
public class SimpleParcelable implements Parcelable {

    // Код опущен...

    private static final Map<String, State> unmarshalState;
    static {
        Map<String, State> m = new HashMap<String, State>();
        m.put("begin", State.BEGIN);
        m.put("middle", State.MIDDLE);
        m.put("end", State.END);
        unmarshalState = Collections.unmodifiableMap(m);
    }

    // инструмент демаршалинга
    public static final Parcelable.Creator<SimpleParcelable> CREATOR
        = new Parcelable.Creator<SimpleParcelable>() {
        public SimpleParcelable createFromParcel(Parcel src) {
```

```

        return new SimpleParcelable(
            src.readLong(),
            src.readString());
    }

    public SimpleParcelable[] newArray(int size) {
        return new SimpleParcelable[size];
    }
};

private State state;
private Date date;

public SimpleParcelable(long date, String state) {
    if (0 <= date) { this.date = new Date(date); }
    if ((null != state) && (0 < state.length())) {
        this.state = unmarshalState.get(state);
    }
}

// Код опущен...
}

```

В данном фрагменте показан лишь только что добавленный код инструмента демаршалинга: общедоступное статичное поле `final`, которое называется `CREATOR`, и содействующие ему компоненты. Поле является ссылкой на реализацию `Parcelable`. `Creator<T>`, где `T` — тип `parcelable`-объекта, к которому применяется демаршалинг (в данном случае речь идет о `SimpleParcelable`). Все эти вещи необходимо делать правильно! Если `CREATOR` будет защищенным (`protected`), а не общедоступным (`public`), не будет статичным или будет записано как `Creator`, фреймворк не сможет произвести демаршалинг объекта.

Реализация `Parcelable.Creator<T>` — это объект с единственным методом `createFromParcel`, который выполняет демаршалинг одного, и только одного экземпляра из `Parcel`. Идиоматический способ решения этой задачи заключается в считывании всех компонентов состояния из `Parcel` в том самом порядке, в каком эта информация записана в `writeToParcel` (это важно), и в последующем вызове конструктора с демаршализованным состоянием. Поскольку конструктор для демаршалинга вызывается из области видимости класса, данный конструктор может быть видим только для классов из того же пакета, в котором находится сам, или вообще закрыт (`private`).

Классы, поддерживающие сериализацию

API `Parcel` может работать не только с шестью примитивными типами данных, которые упомянуты в предыдущем разделе. В документации **Android** приводится полный список `parcelable`-типов, и их удобно подразделять на четыре группы.

К первой группе — простые типы — относятся `null`, шесть примитивных типов (`int`, `float` и т. д.) и упакованные (`boxed`) версии шести примитивных типов (`Integer`, `Float` и т. д.).

Следующая группа включает в себя типы объектов, реализующие `Serializable` или `Parcelable`. Эти объекты не являются простыми, но им известно, как сериализовать себя.

К еще одной группе — типы коллекций — относятся массивы, списки, карты, пакеты и разреженные массивы объектов тех типов, которые входят в две предыдущие группы (`int[]`, `float[]`, `ArrayList<?>`, `HashMap<String, ?>`, `Bundle<?>`, `SparseArray<?>` и т. д.).

Наконец, есть несколько особых случаев: `CharSequence` и активные объекты (`IBinder`).

AIDL и вызовы удаленных процедур

Достаточно сложная межпроцессная коммуникация в Android поддерживается благодаря инструменту, называемому AIDL. Чтобы определить интерфейс на языке AIDL, требуется несколько элементов.

- Файл AIDL, описывающий интерфейс программирования приложений (API).
- Для каждого непростого типа, используемого в API, создается подкласс от `Parcelable`, который определяет этот тип, и файл AIDL, именующий тип как `parcelable`. При этом необходимо помнить: нужно позволить распространение исходника этих классов ко всем клиентам, которым может потребоваться их сериализация.
- Сервис, возвращающий реализацию заглушки API в ответ на `onBind`. `onBind` должен быть приспособлен для каждого намерения, которым он может быть вызван. Возвращаемый экземпляр предоставляет точные реализации методов API.
- С клиентской стороны реализация `ServiceConnection`. `onServiceConnected` должна приводить к нужному типу переданный связующий элемент при помощи `API.Stub.asInterface(binder)` и сохранять результат в виде ссылки на API сервиса. Метод `onServiceDisconnected` должен будет аннулировать ссылку. Он вызывает `bindService` с намерением, которое предоставляется сервисом API, `ServiceConnection` и флагами, контролирующими создание сервиса.
- Связывание является асинхронным. То, что `bindService` возвращает значение `true`, не означает, что уже готова ссылка на службу. Чтобы можно было работать со службой, необходимо освободить поток и дождаться активации `onServiceConnected`.

В то время как все эти типы допускают маршалинг в `Parcel`, с типами `Serializable` и `Map` так лучше не поступать. Как было указано выше, Android поддерживает на-

тивную сериализацию Java. Ее реализация сильно уступает по эффективности остальным частям Parcelable. Реализация интерфейса Serializable — не лучший способ сделать объект Parcelable. Напротив, объекты должны реализовывать Parcelable и добавлять объект CREATOR и метод writeToParcel, как это описано в подразделе «Объект Parcelable для передачи данных» выше. Данная задача может быть довольно трудоемкой, если иерархия объектов сложна. Но оптимизация производительности, которая при этом достигается, обычно стоит потраченных усилий.

Еще один тип, который не следует делать Parcelable, — это Map. Вообще, Parcel не поддерживает карты; исключение представляют только такие карты, ключи которых являются строковыми.

Специфичный для Android тип Bundle предоставляет аналогичную функциональность — карту со строковыми ключами, — но к тому же он гарантирует безопасность типов. Объекты добавляются к Bundle при помощи таких методов, как putDouble и putSparseParcelableArray, по одному методу для каждого Parcelable-типа. Bundle очень похож на карту, за тем исключением, что он может содержать различные типы объектов для разных ключей, гарантируя при этом полную безопасность типов. Пользуясь Bundle, вы избегаете целого класса ошибок, которые сложно находить и которые возникают, когда, например, сериализованный float ошибочно принимается за int.

Безопасность типов — еще одна причина, по которой методы writeTypedArray и writeTypedList предпочтительнее их нетипизированных аналогов writeArray и writeList.

Сериализация и жизненный цикл приложения

Как упоминалось выше, приложение Android не может рассчитывать на такую роскошь, как избыток виртуальной памяти. На небольшом устройстве нет вторичного хранилища, куда можно было бы переместить работающую, но скрытую программу, чтобы освободить пространство для нового видимого приложения. Тем не менее, чтобы пользователю было удобно работать, нужно гарантировать, что, когда пользователь вернется к приложению, работу с которым прервал, оно будет в том же виде, в котором пользователь его оставил. Задача сохранения состояния на то время, пока работа приостановлена, решается самим приложением. Приятно отметить, что во фреймворке Android **сохранение состояния приложения** устроено очень просто.

Пример в подразделе «Сериализация в Java» данного раздела демонстрирует общий механизм, действующий во фреймворке и позволяющий приложению сохранять состояние на время, пока работа с ним приостановлена. Всякий раз, когда приложение выгружается из памяти, вызывается его метод onSaveInstanceState с Bundle, куда приложение может записать любое необходимое состояние. Когда приложение перезапускается, фреймворк передает тот же Bundle методу onCreate, чтобы приложение могло восстановить свое состояние. Оптимально кэшируя контент в ContentProvider и сохраняя легкое состояние (например, видимую в данный момент

страницу) в `onSaveInstanceState`, приложение может вернуться к работе без каких-либо сложностей.

Во фреймворке есть еще один инструмент для сохранения состояния приложения. Класс `View` — базовый тип для всего, что мы видим на экране, — имеет перехватывающий метод (hook method) `onSaveInstanceState`, вызываемый в ходе процесса выгрузки приложения из памяти. На самом деле вызов осуществляется из `Activity.onSaveInstanceState`, поэтому реализация вашего метода в этом приложении всегда должна вызывать `super.onSaveInstanceState`.

Данный метод обеспечивает максимально точное сохранение состояния. Например, почтовое приложение может пользоваться данным методом для сохранения точного положения курсора в тексте черновика электронного сообщения.

4 Передача программы пользователю

В этой главе рассмотрены все аспекты, важные при передаче вашего приложения пользователю. В предыдущих главах книги мы сообщили вам все знания, необходимые для чтения образцов кода и написания простых приложений. Здесь мы завершим эту вводную часть и расскажем, что нужно для широкого распространения ваших приложений, их продажи (если вы ставите перед собой такую цель) и последующего получения денег от Google, которая управляет рынком Android Market¹.

Возможно, вы пока не собираетесь отправлять свое приложение на Android Market, но, зная о том, как происходит такая отправка, вы сможете учесть тонкости этого процесса при проектировании и реализации вашего приложения. Бизнес заставляет учитывать факторы, значительно отличающиеся от других аспектов, связанных с разработкой ПО. В частности, нужно научиться идентифицировать себя на рынке Android Market и для своих покупателей, получать права доступа, необходимые для использования определенных API, защищать свои регистрационные данные и готовить приложение к эксплуатации на достаточно разном оборудовании. Кроме того, нужно уметь время от времени обновлять приложение.

Подписывание приложения

Подписывание приложения, также называемое подписыванием кода, позволяет сообщать устройствам Android на рынке Android Market, а также при других вариантах распространения программы, какие приложения принадлежат владельцу данного цифрового сертификата разработчика. Кроме того, такой «документ» гарантирует, что код не претерпел никаких изменений с тех пор, как был подписан.

Шифрование с открытым ключом и криптографическое шифрование

В основе шифрования с открытым ключом лежит следующий математический принцип: крупные простые числа легко перемножать, но исключительно сложно разложить на множители произведение двух крупных простых чисел. На умножение

¹ На момент подготовки русскоязычного издания Google объединила Android Market с некоторыми другими своими сервисами в Google Play (<https://play.google.com/store>). — Примеч. ред.

уходит несколько миллисекунд, а на разложение такого произведения могут уйти сотни или даже миллионы лет, причем эта задача под силу только астрономически колоссальному компьютеру.

Это асимметрия между умножением и разложением означает, что ключ, полученный путем перемножения двух крупных простых чисел, можно сделать открытым. Чтобы расшифровать два зашифрованных сообщения, необходимо знать пару двух простых чисел, которые входят в состав закрытого ключа. Это означает, что документы, зашифрованные открытым ключом, надежно защищены и расшифровать их может лишь тот, кто владеет закрытым ключом.

Подписывание приложений Android цифровыми сертификатами (то, чем мы будем заниматься) связано с другими свойствами шифрования с открытым ключом, также относящимися к этой операции.

Чтобы поставить на документ цифровую подпись, необходимо сделать следующее.

1. Рассчитать уникальный номер (так называемый хеш) из документа. Этот номер также называется *профилем сообщения*¹ (message digest).
2. «Зашифровать» профиль сообщения закрытым ключом автора подписи. Так получается подпись.

После этого у вас появится номер — цифровая подпись, — связанный с документом алгоритмом хеширования, а также связанный с закрытым ключом автора подписи.

Чтобы верифицировать подписанный документ, нужно выполнить следующее.

1. Рассчитать уникальный номер (хеш) из документа.
2. «Дешифровать» цифровую подпись при помощи открытого ключа, который должен результировать в то же число, что и хеш.

Выясняется кое-что интересное. Документ (в нашем случае компьютерная программа) получен от человека, у которого есть закрытый ключ, парный для открытого ключа, который вы уже использовали при верификации. И вы можете быть уверены, что документ не был изменен: в противном случае хеш, расшифрованный из цифровой подписи, отличался бы от того, что был рассчитан при создании документа.

При верификации цифровой подписи мы также удостоверяемся, что цифровая подпись не была скопирована в другой документ. Цифровые подписи неразрывно связаны с документом, при создании которого они были рассчитаны.



Вы, наверное, обратили внимание на то, что мы ставим слова «зашифровывать» и «расшифровывать» в кавычки, когда говорим о шифровании профиля сообщения или хеша. Дело в том, что при использовании системы открытых и закрытых ключей не происходит шифрования в традиционном понимании. Мы защищаем приложение от несанкционированного доступа, шифруя его открытым ключом, так, чтобы только человек, обладающий закрытым ключом, мог прочесть это сообщение.

В данном случае «зашифровать» означает просто «рассчитать число». Вы не скрываете информацию, когда «зашифровываете» профиль сообщения или хеш закрытым ключом

¹ Встречается также перевод «свертка сообщения». — *Примеч. пер.*

обладателя сертификата. Слова «*шифровать*» и «*расшифровывать*» употребляются потому, что при расшифровке с помощью открытого ключа вы получаете тот же профиль сообщения или хеш, что и при расшифровке с помощью закрытого ключа.

Кто угодно, обладающий открытым ключом и обнародованным алгоритмом, может «расшифровать» профиль сообщения — в этом и заключается смысл верификации. Вы удостоверяетесь, что получили тот же хеш, что и человек, отправивший приложение на рынок, и это, кроме того, *доказывает*, что у отправителя есть закрытый ключ, парный для открытого, и, наконец, что отправитель подписывал именно этот документ.

Поскольку для проведения верификации и соответствующего расчета достаточно открытого ключа, ваша система Android — а также любая другая заинтересованная сторона — может удостовериться, что приложение действительно было подписано конкретным ключом и не было изменено с момента подписи.

В общем смысле любой электронный документ — любой набор битов — может иметь криптографическую подпись, а криптографические подписи, также называемые цифровыми подписями, можно использовать для подписывания документов, и этот способ не менее надежен, чем использование обычной подписи (автографа) владельца документа.

Как цифровые подписи защищают пользователей и издателей программ и обеспечивают безопасный обмен данными

Вы как пользователь компьютерной программы уже, возможно, думаете: «Хотелось бы знать, откуда пришла моя программа и не была ли она изменена до того, как попала ко мне на устройство». Подписанные приложения позволяют не беспокоиться об этом. Гарантии, связанные с цифровыми подписями, вам уже встречались, например, при работе в Интернете. Допустим, когда вы заходите на сайт интернет-магазина, криптографические подписи позволяют вам быть уверенными, что этот сайт действительно заслуживает доверия, а не принадлежит мошеннику, который хочет заполучить ваши деньги и скрыться. В случае интернет-магазина клиент проверяет цифровую подпись сертификата, переданного сервером, пользуясь открытым ключом, полученным из сертифицирующей организации. В вашем браузере уже записаны используемые для этих целей ключи от нескольких сертифицирующих организаций.

Роль сертифицирующей организации заключается в уменьшении количества сторон, которым приходится доверять, своеобразной «консолидации». Вы доверяете производителю вашего браузера и полагаетесь на то, что он использует только те ключи, которые получены от достойных доверия сертифицирующих органов, а владельцы электронных магазинов получают сертификаты от организаций, которым доверяют производители браузеров. Сертифицирующие органы обязаны удостоверять, что сайт, называющий себя Amazon.com, действительно является сайтом Amazon.com. Теперь, когда ваш браузер открывает защищенное соединение с сайтом Amazon.com, вы можете быть уверены в двух вещах. Во-первых, ваши данные защищены от перехвата шифрованием, которое способен дешифровать

лишь сервер этого электронного магазина. И во-вторых, вы можете быть вполне уверены, что сервер, к которому вы подключаетесь, использует сертификат, выданный сертифицирующим органом именно для той компании, с которой вы хотите связаться, так как уже сам сертифицирующий орган удостоверился, что знает, кому именно выдает сертификаты.

Самозаверяющие сертификаты программ Android

При подписывании программ Android цифровой сертификат не выдается какими-либо органами. Его может создать человек, выставляющий программу в общий доступ, — в данном случае это вы. В отличие от сделок электронной торговли, в ходе которых перед вами стоит дополнительное условие — убедиться, что любое открываемое соединение происходит между вашим браузером и именно Amazon.com, пусть и по ссылке неизвестного происхождения, при использовании программных продуктов подписавший цифровой сертификат не имеет столь принципиальной важности.

Для организаций, рассматривающих вариант обратиться за подписью в сертифицирующие органы, в документации Google четко указано, что приложение не обязательно должно быть подписано сертификатом, полученным от уполномоченной организации, и что самозаверение сертификатов — нормальная практика в случае работы с программами Android.

Кроме первоначальной проверки личности разработчика приложения, цифровые подписи в Android применяются и при обновлениях приложения, гарантируя, что обновленному приложению будет разрешен доступ к файлам, созданным в более ранней версии приложения, и что обновление для приложения — это на самом деле не зловредная программа, которая пытается похитить пользовательскую информацию.

При уверенности, что программные обновления получены из того же источника, что и сама программа, вы, разумеется, можете быть уверены и в том, что программы, которыми вы пользуетесь, безопасны, что издатель этой программы известен ее распространителю, в данном случае — известен рынку Android Market.

Кроме гарантии того, что обновления для программы приходят от автора ее оригинала, необходимо отметить, что приложения Android работают в безопасном режиме и требуют прав доступа, описанных по адресу <http://developer.android.com/guide/topics/security/security.html>, если при работе эти программы будут иметь доступ к функциям, способны повредить данные или спровоцировать на устройстве действия, тарифицируемые мобильным оператором.

Подписывание приложения

Концепция криптографического подписывания отличается сложностью и множеством нюансов. Правда, всей этой сложностью управляют инструменты, входящие в состав Android SDK. Когда вы компилируете и запускаете код Android на устройстве или эмуляторе, вы работаете с подписанным кодом.

Отладочные сертификаты

Если вы внимательно отслеживали описанный в книге процесс создания приложения Android и запуска его на эмуляторе или устройстве, то, наверное, заметили, что и без создания сертификата ваше приложение свободно устанавливается на устройстве с Android, несмотря на то что у кода Android должна быть цифровая подпись. Этим удобством мы обязаны тому, что приложение начинает работу с автоматически создаваемым отладочным сертификатом (**debug certificate**). Рассмотрим такой сертификат.

Откройте папку android в домашнем каталоге (home). Там будет файл, который называется debug.keystore. С помощью команды keytool можно получить информацию из этого файла:

```
keytool -list -keystore debug.keystore
```

Когда система отобразит приглашение для ввода пароля, введите android. В ответ вы получите следующий вывод:

```
Keystore type1: JKS  
Keystore provider2: SUN
```

```
Your keystore contains 1 entry3
```

```
androiddebugkey, May 13, 2010, PrivateKeyEntry,  
Certificate fingerprint4 (MD5):  
95:04:04:F4:51:0B:98:46:14:74:58:15:D3:CA:73:CE
```

В графе о типе хранилища ключей и провайдере в качестве хранилища ключей указано Java keystore, совместимое с криптографической архитектурой Java и классами Java, которые позволяют использовать подписывание кода и другие криптографические инструменты. О криптографической архитектуре Java подробнее рассказано по адресу <http://download.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html>⁵.

Команда keytool входит в состав JDK (комплекта для разработки ПО на Java). Кратко она была рассмотрена в пункте «Keytool» подраздела «Другие инструменты SDK» раздела «Компоненты комплекта для разработки ПО» главы 1.

Более подробно о команде keytool вы можете узнать по адресу <http://developer.android.com/guide/publishing/app-signing.html#cert>. Подробная документация о ней приводится на сайте <http://download.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html>.

Последняя строка, выводимая параметром list команды keytool, — это отпечаток сертификата (certificate fingerprint). Это уникальный номер, сгенерированный из ключа.

¹ Тип хранилища ключей. — *Примеч. пер.*

² Провайдер хранилища ключей. — *Примеч. пер.*

³ В вашем хранилище ключей содержится одна запись. — *Примеч. пер.*

⁴ Отпечаток сертификата. — *Примеч. пер.*

⁵ Сокращенный перевод на русский язык находится по адресу http://volgablob.ru/wiki/Java_Cryptography_Architecture. — *Примеч. пер.*

Срок действия такого сертификата достаточно недолог, и с его помощью программы Android можно распространять только в целях тестирования. Еще раз подчеркиваем, что отладочные сертификаты задуманы просто как средство, облегчающее разработку, а не как соблазнительная возможность распространять приложения, не подписывая для них сертификаты!

Создание самозаверяющего сертификата

Вы готовы подписать какой-то код и выпустить его на рынок? Сначала давайте создадим закрытый ключ, воспользовавшись командой `keytool`:

```
keytool -genkey -v -keystore my-release-key.keystore -alias alias_name \
-keyalg RSA -keysize 2048 -validity 50000.
```



Символ `\` указывает разрыв строки и применяется в многострочных командах, которые встречаются в Unix и Mac OS X. Однако в Windows всю команду придется записывать целиком, без символов `\`.

Вы можете сами выбрать имя, заменив им `my-release-key`, а также указать псевдоним и поставить его вместо `alias_name`. Параметры `-keysize` и `-validity` должны остаться такими, как в предыдущем коде.

Как показано в следующем коде, `keytool` запросит у вас пароль к хранилищу ключей, который потребуется запомнить для доступа к этому хранилищу. Кроме того, нужно будет ответить на вопросы о себе, структуре вашей организации и вашем местоположении. `keytool` сгенерирует закрытый ключ, который можно будет использовать для подписывания сертификата. Срок действия такого сертификата — около 150 лет, сертификат будет размещен в файле `<my-release_key>.keystore`:

```
example-user@default-hostname:~$ keytool -genkey -v \
-keystore example-release-key.keystore -alias example_alias_name \
-keyalg RSA -keysize 2048 -validity 50000
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: Example Examp1enik
What is the name of your organizational unit?
[Unknown]: Example
What is the name of your organization?
[Unknown]: Example
What is the name of your City or Locality?
[Unknown]: Example
What is the name of your State or Province?
[Unknown]: Massachusetts
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=Example Examp1enik, OU=Example, O=Example, L=Example, ST=Massachusetts,
C=US correct?
[no]: yes
```

```
Generating 2,048 bit RSA key pair and self-signed certificate (SHA1withRSA)
with a validity of 50,000 days for: CN=Example Examp1enik, OU=Example,
O=Example, L=Example, ST=Massachusetts, C=US
```

```
Enter key password for <example_alias_name>
(RETURN if same as keystore password):
Re-enter new password:
[Storing example-release-key.keystore]
```

Теперь у вас в хранилище есть действующий ключ.

Не потеряйте сертификат!

Хотя во многих отношениях криптографические цифровые подписи более надежны и безопасны, чем обычные автографы, они имеют серьезное отличие от автографов: вы можете утратить способность скреплять документ цифровой подписью.

Если вы потеряете цифровой сертификат, то потеряете свое удостоверение личности в мире устройств Android и на рынке Android Market. Это означает, что, хотя вы и компилируете и выпускаете тот же код, что и ранее, вы не можете пользоваться этим новым кодом для обновления ваших приложений, которые уже есть на рынке, поскольку ни **Android Market**, ни **какое-либо устройство Android** не распознает вас как издателя этого уже существующего кода.

Обязательно храните несколько копий цифрового сертификата на разных носителях (в том числе записывайте его на бумаге), причем эта информация должна находиться в разных местах. Кроме того, такие резервные копии должны быть надежно спрятаны. Если вашим цифровым сертификатом воспользуется кто-то, кроме вас, этот человек может заменить ваши программы на пользовательских устройствах Android своими.

Подробные рекомендации с сайта разработчиков Android о том, как обращаться с цифровым сертификатом, приводятся по адресу <http://developer.android.com/guide/publishing/app-signing.html#secure-key>.



И наоборот, ваша цифровая подпись является вашей лишь потому, что только вы один ею владеете. До тех пор пока вы не пожелаете опубликовать приложение Android и далее считаться его издателем, вы можете генерировать, использовать и аннулировать свои подписи так, как вам вздумается. Не бойтесь экспериментировать и учиться!

Использование самозаверяющего сертификата для подписывания приложения

Теперь подпишем приложение. В Eclipse выберите проект того приложения, которое хотите подписать для выпуска, и выполните команду **File** ▶ **Export** (**Файл** ▶ **Экспортировать**). Может возникнуть вопрос: а почему именно «экспортировать»? Прежде всего, если вы хотите дать кому-нибудь свое приложение «на пробу», то нельзя просто отдать копию архива APK из каталога **bin** в иерархической структуре файлов проекта. Ситуация действительно условна, как это и кажется: диалоговое окно **Export** (**Экспортировать**) — это сокровищница разных функций, и в нем очень удобно выполнять процедуру, которая совсем не сводится к «развертыванию».

В данном примере мы воспользуемся проектом **TestActivity**, но вместо него можно использовать любое приложение — ваше собственное либо пример проекта из этой книги.

Вам будет предложен список настроек экспорта, которые распределены по каталогам. Перейдите в каталог **Android** и далее выберите **Export Android Application** (**Экспортировать приложение Android**) (рис. 4.1). Затем нажмите кнопку **Next** (**Далее**).

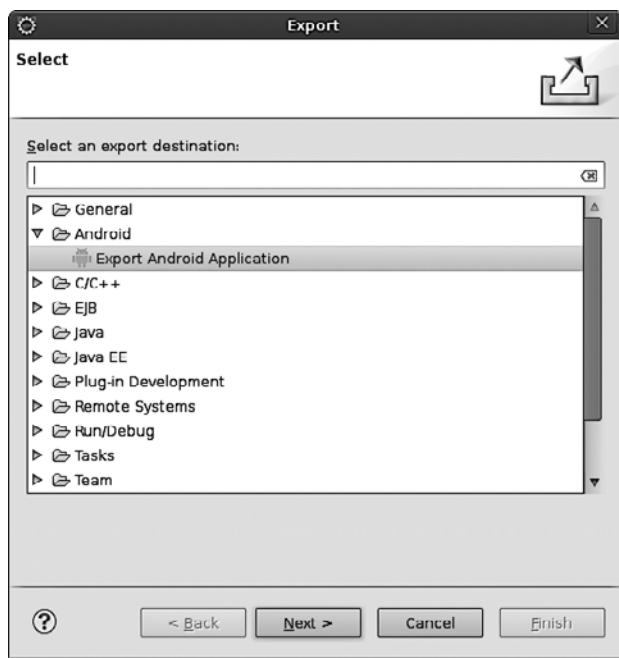


Рис. 4.1. Экспорт приложения Android

Сначала вы увидите, нет ли в конфигурации вашего приложения таких ошибок, которые могут помешать его опубликовать. Одна из возможных ошибок — атрибут `debuggable` в файле описания может иметь значение `true`. Если приложение будет готово к следующему этапу, то вы увидите окно, показанное на рис. 4.2. В нем не указано никаких ошибок.

Следующие диалоговые окна в этой многоэтапной последовательности посвящены в основном подписыванию. Запрашиваемая информация подобна тем данным, которые вы указывали при создании ключа версии в пункте «Создание самозаверяющего сертификата» выше.

Далее нужно выбрать хранилище ключей (рис. 4.3). В хранилище ключей находится ваш ключ.

После того как вы введете название хранилища ключей и пароль, нажмите **Next** (Далее) и переходите к следующему этапу: выберите псевдоним ключа и введите пароль для псевдонима (рис. 4.4).

Если вы внимательно выполнили все этапы, перечисленные в пункте «Создание самозаверяющего сертификата» выше, то сейчас у вас в хранилище ключей должен быть ровно один ключ и один псевдоним. Введите пароль и нажмите **Next** (Далее). Далее следует указать назначение для файла APK и пройти несколько проверок, чтобы определить, нет ли все-таки каких-либо неисправностей в вашем приложении. Если все в порядке, то вы увидите примерно такой экран, как на рис. 4.5.



Рис. 4.2. Приложение Android, с которым не возникает никаких проблем, не позволяющих подписать его и опубликовать



Рис. 4.3. Выбор хранилища ключей



Рис. 4.4. Выбор псевдонима для ключа

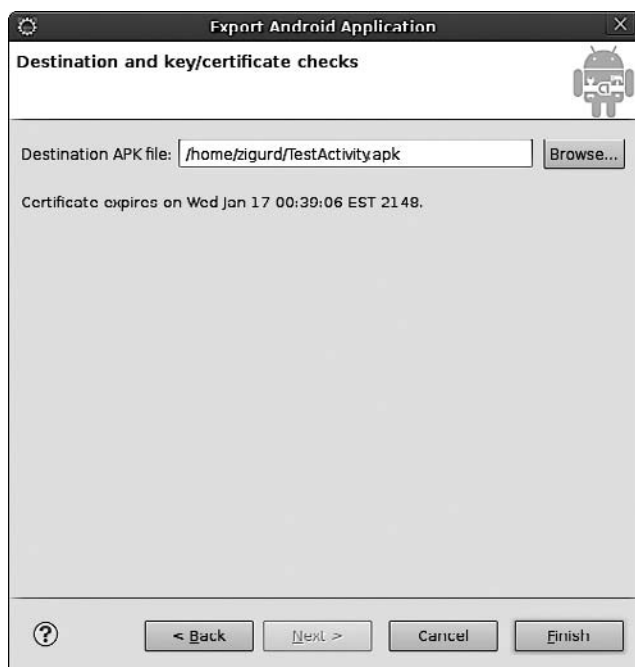


Рис. 4.5. Выбор места назначения и последние проверки

Нажав **Finish** (Готово), вы получаете подписанный файл APK, расположенный в указанном вами месте.

Размещение программы на Android Market для распространения

Отправить программу на рынок **Android Market** исключительно просто, особенно по сравнению с тем, насколько сложен аналогичный процесс в **Apple iTunes AppStore**. Единственное предварительное условие — нужно создать аккаунт **Google**, например почту **Gmail**. Если аккаунт есть, остается уплатить по безналичному расчету (например, с помощью карточки) \$25 и сообщить некоторую информацию о себе. И все, можно приступать к загрузке приложений на рынок. Чтобы взимать плату за использование приложений и получать деньги, потребуется ввести еще некоторую информацию и уладить сущие мелочи — вы даже можете не иметь сайта и статуса юридического лица. (Правда, перед продажей продукции не помешает проконсультироваться с адвокатом. Например, адвокат может посоветовать вам создать фирму или предпринять иные меры, которые помогут защитить ваши личные активы от обязательств, проистекающих из коммерческой деятельности.)

Как стать официальным разработчиком Android

Зарегистрироваться в качестве официального разработчика **Android** можно на сайте **Android Market**. Для регистрации перейдите по следующей ссылке: <http://market.android.com/publish/signup>.

На сайте вас попросят указать личные данные, а также уплатить взнос в размере \$25 по системе онлайн-платежей **Google Checkout**. После регистрации в качестве разработчика вы сможете заходить через свой аккаунт **Google** на рынок **Android Market**.

На данном этапе компания **Google** должна получить гарантии, что вы именно тот, за кого себя выдаете, и что вы можете произвести финансовую транзакцию, связанную с конкретным физическим лицом, которое может платить по кредитным счетам. Это, а также тот факт, что ваши приложения подписаны, означает для **Google**, что ключ, которым вы подписали приложения, находится в собственности именно того человека, который зарегистрировал аккаунт на **Android Market**. И он это сделал, собираясь загружать приложения на рынок. Если окажется, что вы спамер либо рассылаете зловерное программное обеспечение, ваш аккаунт будет аннулирован и вам придется раздобыть новые личные данные, с помощью которых можно будет создать новые аккаунты для **Google Checkout** и **Android Market**.

Загрузка приложений на рынок

Приложения загружаются на рынок со страницы <https://market.android.com/publish/Home#AppEditorPlace>¹. Здесь вы сможете найти новейшие требования и варианты

¹ На момент подготовки русскоязычного издания данная ссылка выглядит следующим образом: <https://play.google.com/apps/publish/signup#AppEditorPlace>. — *Примеч. ред.*

предоставления информации о вашем приложении. На странице есть специальная кнопка для загрузки APK-файла приложения, а также скриншотов, видеороликов и тому подобного контента, большая часть которого не является обязательной. Если у вас есть приложение, которое вы хотели бы оставить на рынке, чтобы его скачивали другие посетители, нужно прочитать описания того, какие рекламные и объяснительные материалы можно загружать на рынок и как этот материал использовать. Пока давайте сделаем так, чтобы наше приложение выполняло необходимый минимум требований.

Первым делом на сайт нужно загрузить файл APK. Чтобы попробовать, как это делается, можете воспользоваться тем APK-файлом, который вы создали в пункте «Использование самозаверяющего сертификата для подписывания приложения» выше. Не волнуйтесь, что это не ваше приложение и что это всего лишь пример. Приложение можно опубликовать, а потом сразу же удалить (депубликовать). Как это сделать, будет рассказано далее в этом разделе.

Самая нужная информация — это либо часть вашего профиля, который вы создали как разработчик Android, либо **часть файла описания приложения**. На момент написания этого текста на рынок требуется загружать два скриншота предлагаемого приложения и изображение-ярлык (значок). Изображения, подходящие для этой цели, вы найдете в каталоге doc проекта-примера. Если эти требования изменятся — а **Android Market существенно изменился с момента своего появления**, — вы заметите все пропущенные поля или загрузки, когда нажмете кнопку Publish (Опубликовать) в нижней части страницы. Все, что вы пропустите, будет подсвечено, и вы сможете вернуться назад, заполнить поля или выполнить загрузки, которые требуются, чтобы ваше приложение было опубликовано.

Нажмите кнопку Publish (Опубликовать).

Поздравляем, вы опубликовали приложение для Android. Если вы вернетесь на страницу <https://market.android.com/publish/Home>, то в списке приложений увидите, что у вас опубликована одна программа (если ранее вы не публиковали чего-нибудь еще). Если перейти по адресу <https://market.android.com> и задать в поиске, скажем, вашу фамилию, поисковик найдет все приложения, которые вы уже опубликовали, и выведет их в виде списка — именно так, как их увидит пользователь, если наберет такой же поисковый запрос на рынке **Android Market**. **С этого места можно подробно просмотреть страницу приложения на рынке.**

Теперь можете вернуться на домашнюю страницу (Home), где указано ваше приложение, и выбрать его, нажав ссылку в списке. Так вы перейдете на страницу с той информацией, которую вы указали при публикации приложения. Эта информация отображается так, что вы можете изменить ее и обновить листинг приложения. Кроме того, вы можете удалить свое приложение с этой страницы, нажав кнопку Unpublish (Депубликовать) в нижней части страницы. Вот и все! Скоро к вам начнут поступать обращения о технической поддержке от пользователей.

Приложение, которое было «депубликовано», не удаляется из системы рынка окончательно. Оно остается в списке ваших приложений, но становится недоступным для загрузки. Вы можете изменить решение и повторно опубликовать приложение в любое время — при помощи кнопки Publish (Опубликовать).

Как заработать

Google Checkout — это платежная система рынка Android Market, то есть Android Market обеспечивает скорейший способ зарегистрироваться в качестве продавца в системе Google Checkout.

Если вы решаете опубликовать платное приложение, то вас перенаправят на страницу, где нужно будет создать коммерческий аккаунт (merchant account). Может показаться, что это звучит угрожающе, но Google разработала очень простую систему получения выплат. Вам не нужно ни регистрировать фирму, ни заводить деловой банковский счет.



Нужно проконсультироваться с адвокатом насчет того, стоит ли заводить юридическое лицо для занятий бизнесом. Кроме того, ваши личные счета и счета для продаж на Android Market должны быть раздельными.

Получение денег на рынке Android Market предполагает связь вашего банковского счета и коммерческого аккаунта в системе Google Checkout. Платежи, которые будут поступать по Google Checkout за продажи приложений, будут перечисляться на ваш банковский счет. Полное описание условий использования, условий платежей и другая подобная информация находятся в разделе сайта Google Checkout, посвященном продажам, по адресу <http://checkout.google.com/support/sell/bin/answer.py?hl=en&answer=113730>.

Альтернативные способы распространения

Еще один признак, отличающий рынок Android от рынка Apple, заключается в том, что на первом существует множество каналов продаж вашего приложения. Google не препятствует сторонним предпринимателям в организации их собственных рынков приложений со своими правилами. Действительно, несколько компаний создали собственные электронные магазины, обслуживающие конкретные рынки. Эти рынки значительно отличаются по своей целевой аудитории, процедурам отправки приложений на рынок, а также по выстраиванию отношений между разработчиком и клиентом. Следует отметить два наиболее значительных рынка, занимающихся продажей приложений Android. Это Verizon Apps (Verizon) и Appstore (Amazon).

Хотя отправка одного приложения на несколько рынков может показаться беспроигрышным ходом — чем больше рынков, тем больше аудитория, — у такого подхода есть и недостатки. На каждом рынке действуют собственные правила по продвижению, выпуску версий (релизов) и поддержке пользователей, а все это не может отвлекать от самой разработки. Разумный и практичный план — начать с одного рынка и постепенно расширяться на новые, по мере того как вы привыкнете к требованиям рынков, на которых уже работаете.

Приложения Verizon для Android

Магазин Verizon Apps для Android обладает некоторыми важными отличиями и потенциальными преимуществами по сравнению с Android Market. Основное

отличие заключается в том, что приложения сначала рецензируются и лишь потом размещаются в магазине. Таким образом, в Verizon отсеиваются низкокачественные и вредоносные приложения — на устройства пользователей они не попадают. Такой процесс утверждения также позволяет Verizon проверять правильность работы приложения в сети.

Кроме того, на рынке Verizon Apps разработчикам предоставляется поточный механизм получения платежей и биллинга, называемый Carrier Billing. Магазин интегрирован с каждым пользовательским аккаунтом. Когда пользователь скачивает приложение, это действие автоматически тарифицируется без необходимости регистрации на внешнем платежном сервисе. Прибыль от продажи приложений распределяется между Verizon и разработчиками: 30 % Verizon и 70 % разработчику.

Verizon также предоставляет API, обеспечивающий биллинг «внутри приложения». Таким образом, сама программа может взимать с пользователей плату за работу с элементами, находящимися внутри программы, например за открытие новых уровней в игре. Платежные записи отражаются на абонентском счете беспроводных услуг.

Как отправить приложение для Android на рынок Verizon Apps

Чтобы отправить приложение для Android на рынок Verizon Apps, выполните следующие шаги.

1. Создайте учетную запись по адресу <http://developer.verizon.com>.
2. Отправьте ваше приложение с указанием следующих данных:
 - 1) сообщите базовую информацию о приложении;
 - 2) просмотрите и примите лицензионное соглашение сообщества разработчиков приложений Verizon (Verizon Development Community App License Agreement);
 - 3) заполните анкеты с вопросами об установке и сетевом использовании приложения. При этом потребуется выбрать подходящий дескриптор приложения;
 - 4) выберите до четырех скриншотов приложения и/или видео для предварительного просмотра;
 - 5) выберите до четырех баннеров приложения для поддержки продаж;
 - 6) задайте модель ценообразования. В частности, можно указать вариант **Always Free** (Всегда бесплатно);
 - 7) выберите рейтинг, соответствующий содержанию вашего приложения;
 - 8) загрузите на сайт свое приложение в двоичном формате; задайте мобильные устройства и операционные системы, которые поддерживает ваше приложение;
 - 9) дождитесь от Verizon уведомления о том, что присланное вами приложение допущено на рынок.
3. Получите финансовую сертификацию. При этом платежный процесс станет проще и быстрее, так как сертификация гарантирует, что Verizon Wireless будет

располагать точной информацией о том, сколько рынок должен вам за продажи приложения.

4. Получайте выручку после того, как пользователи будут устанавливать ваше приложение на свои устройства через Verizon Apps.

Технические рекомендации по разработке приложений Android для рынка Verizon

Когда вы пишете приложение, рассчитывая разместить его на рынке Verizon Apps, обратите внимание на некоторые полезные инструменты, предоставляемые Verizon.

- *Внешнее связывание (deep linking)*. Рынок Verizon Apps для Android поддерживает внешнее связывание. Это действующий в Android механизм программирования, благодаря которому приложение Android может активировать пользовательский интерфейс рынка Verizon Apps для установки другого приложения. Такой механизм удобен для тех разработчиков приложений, которые предлагают пользователю усовершенствованное бесплатное приложение, установив его расширенную платную версию. Кроме того, данный механизм способствует так называемому «совместному маркетингу» приложений, когда через одно приложение мы стимулируем пользователя установить другое. На сайте разработки Verizon приведены примеры кода для реализации такой процедуры.
- *NAVBuilder Inside*. Verizon поддерживает продукт под названием NAVBuilder Inside (NBI) — картографическую программу, напоминающую навигационную систему Google Maps. Программа NBI по умолчанию устанавливается на большинстве мобильных устройств Android. Многие устройства, поставляемые с программным обеспечением Verizon, также имеют предустановленную версию NBI. На сайте Verizon для разработчиков предоставляется Android SDK и документация, помогающая создавать приложения для работы с NBI. NBI — бесплатная кросс-платформенная программа, которая поддерживает купонную систему оплаты.
- *Сетевой API*. Verizon открыл свою магистральную сеть связи для разработки веб-служб при помощи API, основанных на SOAP и REST. Хотя к этим API нет прямого доступа от приложений мобильного устройства, можно настроить посредник для веб-сервисов. Посредник поддерживает непрямую активацию мобильного устройства. Эти API поддерживают следующие сервисы:
 - *SMS- и MMS-сообщения* — обеспечивает отправку и прием SMS- и MMS-сообщений;
 - *местоположение как услуга* — предоставляет информацию о местоположении вызывающей стороне, работающей с API;
 - *информация об операторе* — позволяет вызывающей стороне получать информацию об операторе, который обслуживает конкретный телефонный номер. Работает с номерами, не подписанными непосредственно на Verizon.

Все сетевые API Verizon используют единый безопасный и единообразный интерфейс шлюза. Пользователи мобильных устройств должны специально согласиться на получение любой информации через API, например, это касается информации о местоположении.

Ссылки по теме

Чтобы подробнее познакомиться с темой, посмотрите следующие ссылки сообщества Verizon:

- Verizon Apps for Android Submission Guide (<http://developer.verizon.com/content/vdc/en/verizon-app-submit.html>);
- NAVBuilder Inside SDK (http://developer.verizon.com/content/vdc/en/verizon-tools-apis/verizon_apis/navbuilder-inside-sdk.html);
- The Verizon Network API (http://developer.verizon.com/content/vdc/en/verizon-tools-apis/verizon_apis/network-api.html).

Приложения Amazon для Android

Amazon Appstore — очень известный рынок, на котором вы можете напрямую заказать Kindle Fire. На момент написания книги этот планшет является безусловным лидером по популярности среди планшетов Android. Процесс отправки приложения на рынок основан на веб-взаимодействиях, он прост и очень хорошо документирован на портале разработчиков по адресу <https://developer.amazon.com/welcome.html>. Чтобы начать работу, просто зарегистрируйтесь на сайте как разработчик. В настоящий момент Amazon не взимает с разработчиков ежегодного взноса за первый год. Вообще ежегодный взнос равен \$99. Сразу после регистрации вы можете отправить на рынок столько приложений, сколько захотите: просто следуйте инструкциям, размещенным на портале.

Как и на большинстве других рынков, в процессе отправки приложения вам будет предложено предоставить различные промоматериалы. Как минимум вам потребуется загрузить ярлык, миниатюру и три скриншота. Дополнительно может понадобиться предоставить более крупные промоизображения и даже до пяти коротких видеоклипов, которые портал Amazon может использовать для рекламы вашего приложения.

Amazon выплачивает за вашу программу лицензионные отчисления, которые могут составлять не более 70 % цены за приобретение программы или 20 % номинальной (объявленной) стоимости программы. Когда вы подаете заявление на включение программы в список товаров на рынке, вы и Amazon вступаете в юридические отношения. Как и обычно в таких случаях, вы должны внимательно изучить заключаемое соглашение. В данном случае это App Distribution Agreement (Дистрибьюторское соглашение Amazon) (<https://developer.amazon.com/help/da.html>). Обратите внимание на несколько аспектов.

- Amazon предполагает, что вы будете заниматься технической поддержкой приложения на протяжении всего времени, пока оно находится в Appstore. Если Amazon сочтет, что с вашим приложением возникла критическая проблема, то вы должны отреагировать на соответствующее уведомление в течение 24 часов. На все остальные пользовательские запросы вы должны отвечать в течение пяти дней.
- Amazon устанавливает цену для вашего приложения. Как правило, цена составляет \$0,99 или 1,99. Amazon часто проводит промоакции, в ходе которых

приложения раздаются бесплатно. Возможно, администрация портала решит включить в такую раздачу ваше приложение. Необходимо взвесить, сможет ли продажная цена приложения покрыть расходы на обслуживание сервисов серверной части (backend services), применяемых в ходе таких промоакций.

- На большинстве рынков вы подписываете ваше приложение ключом, который есть только у вас. Когда пользователь устанавливает приложение, он может быть уверен, что получает тот самый код, который вы (разработчик) отправили на рынок. Как бы то ни было, за подлинность приложения отвечаете вы сами. В Amazon Appstore ситуация обстоит иначе. Amazon обертывает ваш код и использует для подписывания ваших приложений, продаваемых на рынке, ключи, которые ассоциированы с вашим аккаунтом разработчика. Amazon предъявляет такое требование, чтобы администрация рынка могла изменять код вашего приложения для сбора аналитических данных об использовании приложения и составлении отчетности по ним. Правда, на рынке предоставляется и такой сервис, при помощи которого вы можете сами подписывать приложение собственным частным ключом, но только после того, как Amazon модифицирует этот ключ.

Кроме того, Amazon предоставляет сервис управления цифровыми правами (digital rights management). В ходе процесса отправки вы можете, установив специальный флажок, бесплатно защитить свое приложение при помощи системы Amazon для управления цифровыми правами. Amazon заявляет, что защищенное таким образом приложение будет работать только на тех устройствах, где также установлено ПО для Amazon Appstore, и только при условии, что Amazon сможет верифицировать действующую лицензию на ваше приложение.

После отправки приложения на рынок Amazon рецензирует качество и содержание вашей программы. Применяемые на рынке стандарты для оценки контента также должны показаться вам знакомыми: рынок может присвоить приложению определенный рейтинг доступности либо вообще отклонить его, если приложение является незаконным, порнографическим или вредоносным. Когда процесс рецензирования будет завершен, ваше приложение будет предоставлено вниманию всей огромной пользовательской аудитории Amazon.

Ключи к интерфейсу программирования приложений (API) для работы с картами Google

Ключи к интерфейсу программирования приложений для работы с картами Google (Google Maps API) вместе с ключами для подписывания приложений идентифицируют вас в Google и позволяют Google обеспечивать соблюдение условий использования карт Google. В основе Google Maps лежит информация, на сбор и покупку которой Google тратит значительные средства. По данной причине сервис необходимо защищать от незаконного присвоения и другого злоупотребления.

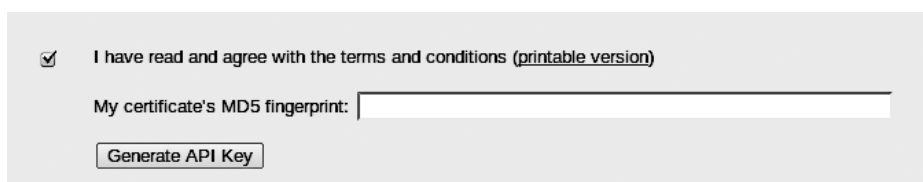
Если вам случалось разрабатывать приложения при помощи Google Maps API, вы должны были получить ключ к API, связанный с отладочной подписью вашего приложения. Этот ключ к API не будет работать при отправке приложения на

рынок. Интерфейс Google Maps API описан по адресу <http://code.google.com/android/maps-api-signup.html>.

При отправке приложения на рынок вам потребуется ключ к Google Maps API, связанный с ключом подписи, которым вы пользуетесь при распространении вашего приложения. То есть вам понадобится новый ключ к API, создаваемый при помощи MD5-отпечатка вашего ключа подписи. С помощью параметра `list` команды `keytool` вы сможете получить MD5-отпечаток вашего ключа подписи следующим образом:

```
keytool -list -keystore my-release-key.keystore
```

Этот ключ мы получаем тем же способом, что и отладочный ключ к API. То есть нужно посетить страницу подачи заявок на получение ключа к Android Maps API по адресу <http://code.google.com/android/maps-api-signup.html>. Здесь нужно указать в форме MD5-отпечаток вашего ключа подписи (рис. 4.6).

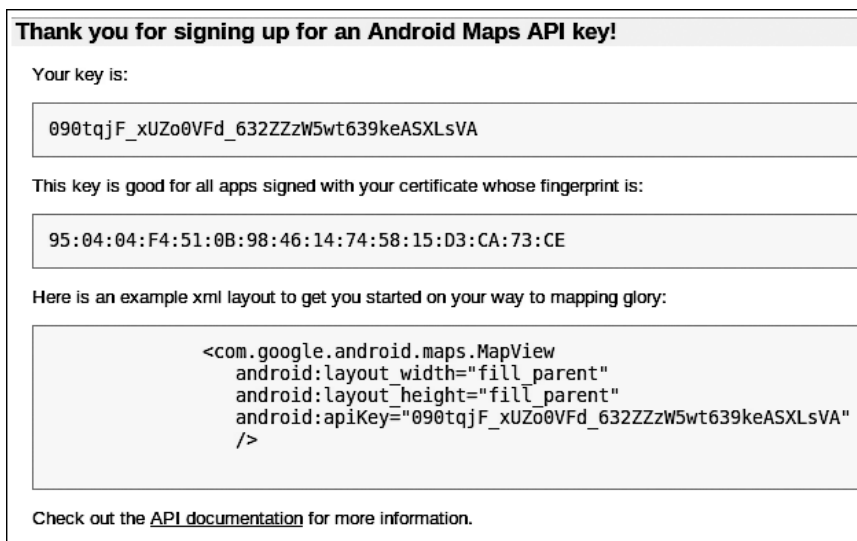
The image shows a web form for generating a Google Maps API key. It has a checkbox with a checkmark and the text "I have read and agree with the terms and conditions (printable version)". Below this is a label "My certificate's MD5 fingerprint:" followed by a text input field. At the bottom is a button labeled "Generate API Key".

☒ I have read and agree with the terms and conditions ([printable version](#))

My certificate's MD5 fingerprint:

Рис. 4.6. Получение ключа к Google Maps API

При нажатии кнопки `Generate API key` (Сгенерировать ключ к API) откроется веб-страница, на которой отобразится ключ, сгенерированный из отпечатка вашего сертификата (рис. 4.7).

The image shows a confirmation page for the Google Maps API key. It has a title "Thank you for signing up for an Android Maps API key!". Below the title is the text "Your key is:" followed by a text box containing the key "090tqjF_xUZo0VFd_632ZZzW5wt639keASXLsVA". Below this is the text "This key is good for all apps signed with your certificate whose fingerprint is:" followed by a text box containing the fingerprint "95:04:04:F4:51:0B:98:46:14:74:58:15:D3:CA:73:CE". Below this is the text "Here is an example xml layout to get you started on your way to mapping glory:" followed by a text box containing an XML snippet for a MapView. At the bottom is the text "Check out the API documentation for more information.".

Thank you for signing up for an Android Maps API key!

Your key is:

090tqjF_xUZo0VFd_632ZZzW5wt639keASXLsVA

This key is good for all apps signed with your certificate whose fingerprint is:

95:04:04:F4:51:0B:98:46:14:74:58:15:D3:CA:73:CE

Here is an example xml layout to get you started on your way to mapping glory:

```
<com.google.android.maps.MapView
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="090tqjF_xUZo0VFd_632ZZzW5wt639keASXLsVA"
/>
```

Check out the [API documentation](#) for more information.

Рис. 4.7. Ключ к интерфейсу Android Maps API, сгенерированный из самозаверяющего сертификата



Цифровой сертификат и ключи к Google Maps API вы должны создавать себе сами. Нельзя копировать эти данные из приведенных здесь скриншотов, нельзя брать ключи из примеров кода к этой книге и, наконец, нельзя использовать отладочные ключи, выпуская продукт на рынок.

Обеспечение совместимости на уровне интерфейса программирования приложений

В любой момент на рынке присутствует несколько рабочих версий Android, и не у всех потенциальных пользователей вашего приложения обязательно будет новейшая версия. Указывая в файле описания информацию о совместимости приложения, вы можете контролировать, в каких системах с **Android будет устанавливаться** ваше приложение, и, соответственно, вы гарантируете, что ваши программы не попадут в системы, использующие несовместимые с ними API.

В примере из подраздела «Создание проекта **Android**» раздела «Проверка работоспособности» главы 1 мы указывали один и тот же уровень API и для целевой сборки, и в поле с номером минимальной версии SDK. В таком случае программа будет работать только в системах, имеющих указанный или более высокий уровень API.

Поскольку уровень API можно проверить и во время исполнения, в некоторых случаях бывает необходимо внедрить приложение в системах с более низким API, чем тот, с которым работаете вы. Придется тестировать систему на API с таким уровнем и использовать методы и классы лишь с более высокими уровнями API — если такие методы и классы доступны. В таких случаях уровень API целевой сборки будет выше, чем уровень минимальной приемлемой версии SDK.

Совместимость с экранами нескольких разновидностей

При создании Android учитывался тот фактор, что система должна работать с экранами разнообразных размеров, а также адаптироваться к изменениям ориентации экрана. Наилучший способ, позволяющий справляться с несовпадением параметров экрана в различных устройствах Android, заключается в том, что компоновка ваших страниц должна быть настолько гибкой, насколько это возможно. Изображения, используемые в вашей программе, могут неидеально выглядеть на очень больших и нетипично малых экранах, но можно подобрать варианты компоновки, которые будут вполне удобны для работы на любом экране: от мельчайшего дисплея с умеренным разрешением до высококачественного экрана с разрешающей способностью 1920 × 1080.

Иными словами, не пытайтесь справиться с экранными различиями, разрабатывая множество вариантов компоновки и разновидностей ресурсов для экранов с различным разрешением. Вместо этого сделайте всего пару макетов, каждый из которых будет достаточно гибок, чтобы подходить для большой группы схожих экранов. Работая таким образом, вы готовите ваше приложение к использованию почти на любом новом устройстве, которое может появиться на рынке.

Тестирование совместимости с размером экрана

Тестирование исключительно важно для обеспечения совместимости программы с дисплеями. В комплекте для разработки ПО и в диспетчере виртуальных устройств Android можно задать конфигурации для любых параметров экрана, которые только могут быть у смартфонов, работающих с Android. Как было рассказано в подразделе «Создание виртуального устройства Android (AVD)» раздела «Проверка работоспособности» главы 1, можно указать заданные и настраиваемые пользователем размеры экрана уже на этапе создания виртуального устройства Android.

Квалификаторы ресурсов и размеры экранов

Когда у вас будут варианты компоновки, пригодные для работы в подавляющем большинстве случаев, вы, возможно, захотите улучшить отображение вашей программы на некоторых конкретных дисплеях. Возможно, придется отдельно использовать макеты в случаях, когда потребуется с максимальной пользой задействовать пространство широкого экрана, а не просто растянуть страницу во весь экран. (Например, можно выделить отдельную секцию для предварительного просмотра.) В таких ситуациях, а также при работе со специализированными приложениями, которые приходится просматривать на исключительно мелких экранах, можно делать варианты компоновки для особых случаев, применяя при этом квалификаторы ресурсов.

Квалификаторы ресурсов — это набор правил наименования для каталогов с ресурсами. Эти правила позволяют предоставлять альтернативные ресурсы для работы в условиях, описываемых квалификатором. Примерами таких условий могут быть, в частности, высокая или низкая плотность пикселей, язык, страна и доступность определенных аппаратных ресурсов. Весь спектр квалификаторов ресурсов представлен по адресу <http://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources>.

5 Среда Eclipse для разработки программ Android

Тема Eclipse довольно противоречива. Это история об огромном успехе программы, которая начиналась как свободный проект с открытым кодом. Eclipse — наиболее популярная интегрированная среда разработки (IDE) для Java. Среда Eclipse очень мощная и является центром богатейшей экосистемы настроек и производных продуктов, используемых при разработке программного обеспечения. По определенным причинам именно Eclipse стала тем ориентиром, для которого разрабатывается множество плагинов. Эти подключаемые модули приспособляют Eclipse для написания именно таких программ, которые ориентированы на операционную систему Android. При этом Eclipse критикуют за некоторое неудобство для пользователя и за то, что ее сложно изучать.

Eclipse не похожа на большинство программ с графическим пользовательским интерфейсом, избавляющих пользователя от возможности активизации операций, которые потом не удастся завершить. Разработчики Eclipse старались достичь максимальной мощности и модульности, не особенно уделяя внимание сглаживанию шероховатостей. Например, при запуске программы-примера вы, вероятно, сразу заметили, что Eclipse позволяет делать с вашими приложениями для Android такие вещи, которые, как кажется, не имеют особого практического смысла. Допустим, можно смоделировать работу приложения на сервере или в качестве апплета (рис. 5.1).

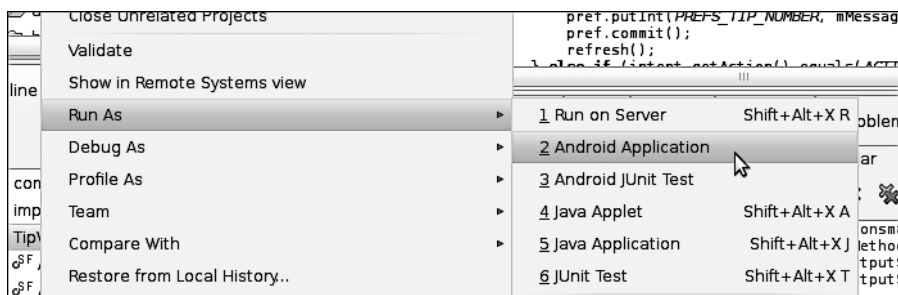


Рис. 5.1. Запуск Eclipse в качестве апплета, заведомо невыполнимая задача

Мы не будем ни критиковать Eclipse, ни приводить аргументы в ее защиту. Но мы объясним, почему Eclipse работает так, а не иначе. Рассмотрим, как компоненты Eclipse образуют единую отлаженную систему. Мы расскажем вам именно о том, что вы ищете, когда открываете Eclipse и начинаете писать в ней код. Вооружившись знаниями из этой главы, вы сможете эффективно использовать Eclipse, и она покажется вам гораздо менее мудреной, чем на первый взгляд.

Документация, касающаяся Eclipse, приводится по адресу <http://www.eclipse.org/documentation>.

Концепции и терминология Eclipse

В Eclipse используется собственная номенклатура (система наименований) с ключевыми концепциями, необходимыми для понимания данной среды. Эти концепции развивались на протяжении уже довольно долгой истории программы. Все началось с VisualAge — инструмента для разработки приложений, написанного на языке SmallTalk в середине 1980-х. Современные реализации Eclipse написаны на языке Java на основе фреймворка Equinox, в котором реализуется спецификация для модульных программных систем на Java. Эта спецификация называется OSGi. OSGi — это способ указания в файле описания такой информации, как данные о жизненном цикле и зависимостях динамически загружаемых модулей, которые называются наборами (*bundles*). То есть Eclipse можно охарактеризовать как коллекцию модулей во фреймворке. При добавлении или удалении модулей существующие между ними зависимости по возможности будут удовлетворяться автоматически.

Более подробная информация о реализации Equinox OSGi содержится по адресу <http://eclipse.org/equinox/documents/quickstart.php>.

Плагины

Когда вы занимаетесь настройкой инструментов для разработки программ в Android, вы добавляете в Eclipse плагины для разработки в Android (ADT). Плагины — это наборы, входящие в OSGi.

Комплект Android SDK добавляет в Eclipse два плагина. Для их просмотра перейдите в окно просмотра плагинов, выполнив **Window ▶ Show View ▶ Other** (Окно ▶ Показать вид ▶ Прочие) и раскрыв элемент **Plug-in development** (Разработка плагинов). Затем выберите из списка видов вариант **Plug-ins** (Плагины). Откроется список, показанный на рис. 5.2.



Обратите внимание, что здесь и далее в этой книге скриншоты Eclipse показаны в так называемом **раздельном режиме**, то есть в отдельном окне. Это значит, что не приходится обрезать со скриншота окружающие виды и панели инструментов. Как конкретный вид будет выглядеть у вас на экране, зависит от того, в какой перспективе Eclipse вы работаете, и от других видов этой перспективы. Если щелкнуть правой кнопкой мыши на заголовке вида, появится меню с командами настройки вида. В их числе будет команда **Detached** (Раздельный режим).

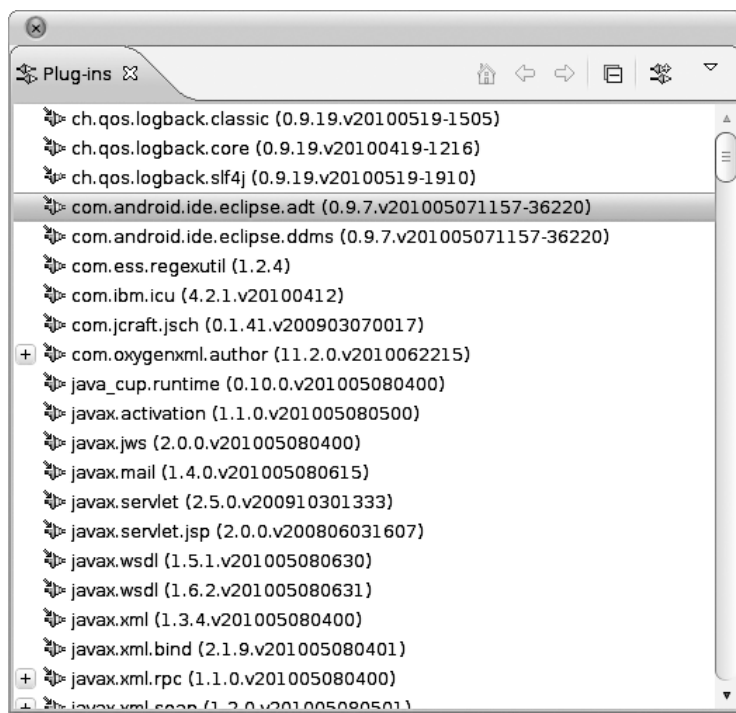


Рис. 5.2. Список плагинов, работающих в среде Eclipse

Плагины перечислены в алфавитном порядке. В верхней части списка вы найдете два плагина, необходимых для работы с Android SDK: `com.android.ide.eclipse.adt` и `com.android.ide.eclipse.ddms`. Не закрывайте список **Plug-ins** (Плагины).

Ниже в этой главе мы подробнее изучим данные плагины, в частности опишем, как они изменяют среду Eclipse. Кроме того, если внимательно рассмотреть список плагинов, то становится очевидно, что Eclipse на самом деле состоит из плагинов, вплоть до уровня реализации Equinox OSGi.

Рабочие пространства

В Eclipse содержится множество состояний. Они располагаются в рабочих пространствах (workspaces). При первом запуске Eclipse система спрашивает, хотите ли вы создать рабочее пространство. После этого при каждом запуске Eclipse работа начинается с того места, на котором вы остановились. Это относится ко всем проектам, файлам и видам, имевшимся в работе на момент закрытия программы. Предыдущее состояние среды считывается из рабочего пространства. Рабочие пространства в Eclipse реализованы на основе каталогов.

Любой проект относится к рабочему пространству. По умолчанию новые проекты и каталоги создаются в каталоге рабочего пространства. Даже если проект создается из источника, находящегося вне рабочего пространства, в рабочем пространстве остается большая часть метаданных об этом проекте.

Рабочие пространства независимы друг от друга. Настройки, которые вы принимаете в одном рабочем пространстве, в нем сохраняются. Можно одновременно использовать несколько рабочих пространств, разделяя, таким образом, проекты, ориентированные на различные платформы, например проект для Rails и проект для Android. Можно использовать несколько рабочих пространств и одновременно запускать несколько экземпляров Eclipse. **Предположим, вы применяете инструменты из арсенала Eclipse для разработки в каком-нибудь фреймворке веб-приложений и эти инструменты несовместимы с той версией Eclipse, в которой вы занимаетесь разработкой для Android.** Отводя для разработки в Android отдельное рабочее пространство, вы можете поддерживать отдельные состояния и даже одновременно работать с обеими версиями Eclipse.

Среды разработки для Java

При написании программ для Java в Eclipse используются три различных окружения для Java.

Среда времени исполнения Java для Eclipse

Первой средой является та, в которой работает сама Eclipse. В подразделе «Интегрированная среда разработки Eclipse» раздела «Установка комплекта разработки ПО (SDK) Android и необходимые условия» главы 1 мы рассказали, как установить инструменты, необходимые для разработки на Java, а также рассмотрели установку сред времени исполнения. Если для Eclipse вам требуется использовать иную среду времени исполнения, то вы можете сконфигурировать окружение необходимым образом в файле `eclipse.ini`, находящемся в каталоге, в котором установлена программа Eclipse. Если, например, в Eclipse начнется дефицит памяти, то вам придется откорректировать выбранную вами среду времени исполнения.

Компилятор Java

Второе рабочее окружение используется для компиляции кода. В состав Eclipse входит собственный пошаговый (инкрементный) компилятор для Java. **Этот инструмент создает не только скомпилированные файлы .class для Java, но и сообщения об ошибках, отображаемые в редакторе Java. Кроме того, данный инструмент выдает информацию о типах объектов, которой Eclipse пользуется, например, при живом поиске, автозавершении и т. д.** Среда конфигурируется при помощи узла **Java ▶ Compiler (Java ▶ Компилятор)** в окне **Preferences (Настройки)**, но вы можете изменить этот стандартный порядок в конкретном проекте. Это можно сделать в разделе настроек проекта.

Кроме того, в данной среде содержится описание библиотек, на основании которых компилируется приложение. Если внимательно изучить команду **Preferences ▶ Build Path (Настройки ▶ Путь сборки)** для приложения Android, можно заметить, что в списке библиотек, на которых строится разработка проекта, нет среды времени исполнения Java. Вместо этого проект зависит от библиотек Android. Тем не менее, поскольку инструменты Android тесно связаны с Eclipse, вы не сможете изменить версию библиотеки Android непосредственно в подокне **Build Path (Путь сборки)**. Это следует делать в подокне с настройками Android.

Среда времени исполнения приложения

Третьей является среда, в которой будут работать ваши приложения. В данном случае это один из эмуляторов Android. При настройке вашей среды разработки, которая происходит на этапе установки Android SDK или настройки плагина ADT, вы устанавливаете одно виртуальное устройство Android (AVD) или более. Создавая новый проект Android, вы связываете его с одним из виртуальных устройств. Плагин использует соответствующий профиль, чтобы настроить как среду компиляции, так и эмулятор, применяемый для работы с приложением. Тем самым снижается вероятность несовпадения во время исполнения. Так, например, приложение, скомпилированное на базе библиотек для платформы Android 2.2, не сможет работать на платформе 1.5.

Проекты

С точки зрения разработчика, каждый проект Eclipse соответствует одной из создаваемых программ, то есть для разработчика программ для Android каждый проект — это программа Android. Внутри Eclipse проекты являются теми единицами, по которым плагины ориентируются, с каким программным обеспечением предполагается работать. При создании проекта Android в состав проекта входит в том числе информация, которую Eclipse применяет для выборки кода из различных плагинов и осуществления различных операций. Плагины ADT активируются для того, чтобы помочь создать проект с правильным набором файлов, а также структурой каталогов, принятой в приложениях Android. Если вы работаете с файлами в проекте для Android, то при открытии XML-файлов, в частности файлов компоновки и описания, используются правильно подобранные редакторы. При изменении файлов в приложении для построения программы вызываются подходящие компоновщики.

Компоновщики и артефакты

Во фреймворке Eclipse под компоновщиком (Builder) понимается средство, генерирующее из исходного кода артефакты. *Артефакты* — это файлы, построенные на основе файлов-исходников. В плагине Android для Eclipse определяется несколько новых компоновщиков, которые создают файлы DEX из файлов CLASS, константы Java, идентифицирующие ресурсы, описанные на XML, а также файлы APK и осуществляют другие Android-специфичные операции, происходящие в процессе преобразования кода в устанавливаемый пакет. Eclipse заново генерирует устанавливаемый пакет всякий раз, когда вы вносите в программу изменения. Вы всегда должны иметь возможность приступить к запуску или отладке пакета.

Преобразование файлов CLASS, которые являются выводом компилятора Java, в файлы DEX, представляющие собой байт-код, интерпретируемый виртуальной машиной Dalvik, — это довольно красивый ход. Так вы можете программировать на Java, пользуясь высокоразвитыми инструментами редактирования и рефакторинга, предлагаемыми при работе на этом языке, а также применять и многочисленные другие инструменты, которые призваны повысить надежность и производительность кода Java.

Расширения

Расширения — это все те места, в которых плагины расширяют функционал Eclipse. Вы занимаетесь разработкой для Android, поэтому не будете непосредственно иметь дело с расширениями или изменять их, но, раз уж у нас открыта перспектива Plug-ins (Плагины), давайте кратко остановимся на некоторых расширениях, основанных на использовании плагинов Android. Так мы сможем точнее обрисовать взаимосвязи, существующие между плагинами ADT и остальными компонентами системы Eclipse. В перспективе Plug-ins (Плагины), показанной на рис. 5.2, дважды щелкните кнопкой мыши на плагине `com.android.ide.eclipse.adt`. Откроется вид Extensions (Расширения), в котором перечислены расширения плагина (рис. 5.3).

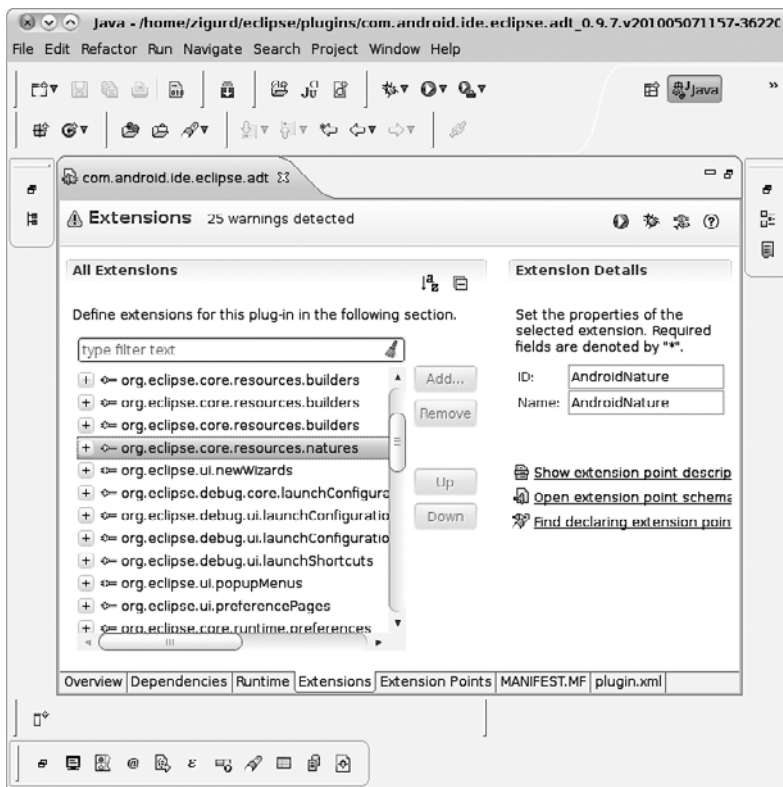


Рис. 5.3. Список расширений плагина ADT

Например, можно выбрать все расширения в `org.eclipse.core.resources.builders`, и в правой части вида Extensions (Расширения) вы увидите названия всех этих расширений: Android Resource Manager (Диспетчер ресурсов Android), Android Pre Compiler (Инструмент предварительной компиляции в Android) и Android Package Builder (Компоновщик пакетов Android).

Эти расширения необходимы для обработки ресурсов Android, предварительной компиляции на языке AIDL (язык определения интерфейса Android) (как было

описано в главе 3) в код Java; а также для превращения файлов CLASS, создаваемых при помощи компоновщика Java, в файлы DEX. И, наконец, для построения файла APK, который может быть развернут на реальном или виртуальном устройстве Android.

Если развернуть элемент `org.eclipse.ui.editors`, вы увидите список редакторов, которыми плагин ADT дополняет систему Eclipse: Android Manifest Editor (Редактор описаний Android), Resources Editor (Редактор ресурсов), Graphical Layout Editor (Графический редактор компоновки), Menu Editor (Редактор меню) и XML Resources Editor (Редактор XML-ресурсов). Этими расширениями список не ограничивается. По величине списка можно судить, какой объем кода требуется для адаптации Eclipse к разработке программ для системы Android. Компонентов, которые мы здесь исследовали, вполне достаточно для понимания, как строятся программы Android и какие компоненты добавляются в среду Eclipse для редактирования Android-специфичных файлов, в том числе файлов описания на языке XML, макетов страниц и других ресурсов.

Если аналогичным образом рассмотреть другой плагин ADT, можно узнать, как в Eclipse добавляются функции инструмента Dalvik Debug Monitor Server (DDMS).

Ассоциации

Ассоциации описывают связи между файлами внутри проекта, в частности, как файлы связаны с редакторами, предназначенными для их обработки. Например, файлы Java в составе проекта Android обрабатываются в том же редакторе, что и файлы обычного проекта на Java, но XML-файлы обрабатываются в специфичном для Android XML-редакторе, например в редакторе файлов описания или в редакторе ресурсов Android. Этим редакторам известно, как работать с конкретными структурами, содержащимися в файлах того или иного формата, но в других сферах от них мало пользы, например, они не подходят для универсального редактирования, которое происходит в виде Outline (Визуализация). Если вы хотите открыть файл Android в ином XML-редакторе, который не входит в число вызываемых для работы редакторов, ассоциированных с данным файлом, можно переопределить имеющиеся ассоциации посредством команды Open With (Открыть с помощью). Эта команда находится в контекстном меню исходного файла, которое появляется при щелчке правой кнопкой на этом файле в виде Package Explorer (Просмотр пакетов).

Команда Open With (Открыть с помощью) предлагает на выбор ряд редакторов, которые, вероятно, смогут обработать выбранный вами файл. Если выполнить команду Others (Другие), то появится список всех редакторов, присутствующих в данной конфигурации Eclipse, а также будет предложено открыть файл во внешней программе.

Виды и перспективы Eclipse

Кроме понимания того, каким образом плагины ADT модифицируют Eclipse, вам пригодится общее представление о системе видов и перспектив Eclipse. Ориентируясь в этой системе, вы будете лучше понимать, что видите в том или ином

окне, занимаясь в Eclipse разработкой для Android. **Видом (view) в Eclipse** называется область окна, в которой выводится информация определенного типа либо особым способом. Среди разновидностей информации можно назвать список проектов и файлов проектов, список ошибок, найденных в коде, иерархический вид, демонстрирующий расположение сущностей в классе, и т. д. Перспектива (perspective) — это ряд видов, упорядоченных определенным образом и предназначенных для определенной цели, например для редактирования Java или отладки.

Если в той интегрированной среде Eclipse, с которой работаете вы, не выводится того же набора видов, которые показаны в примерах к этой главе или перечислены среди основных видов, — не волнуйтесь. Различия в поведении могут объясняться разными наборами плагинов и видов. Самыми важными перспективами для написания кода на Java являются Package Explorer (Диспетчер пакетов), Editor (Редактор) и Outline (Структура). Они обязательно должны присутствовать в вашей среде Eclipse.

При первом запуске Eclipse (после того как программа отобразит вводную заставку с приглашением), но еще до того, как будет создан первый проект, вы должны видеть на экране примерно то, что показано на рис. 5.4.

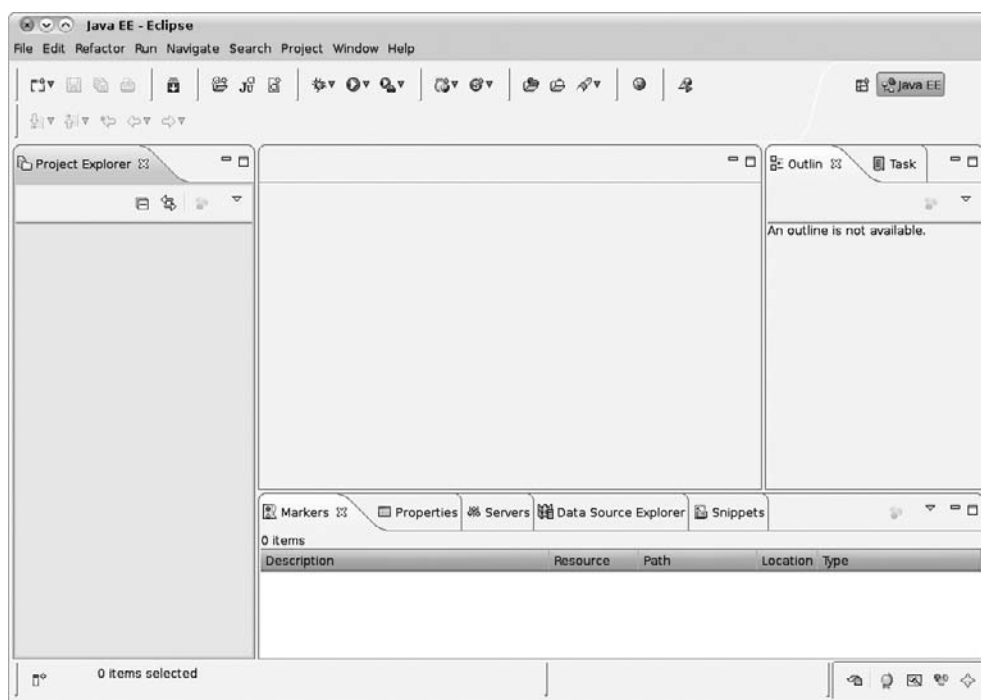


Рис. 5.4. Пустое рабочее пространство, в котором сконфигурирован плагин ADT

Показанное здесь рабочее пространство несколько более загроможденное, чем вы ожидаете. Большинство разработчиков предпочитает пользоваться широкими экранами, чтобы видеть информацию из видов, окружающих перспективу Editor

(Редактор) в окне Eclipse, и оставляют на экране достаточно места, чтобы помещался довольно большой фрагмент кода.

Мы оставили эти перспективы в минимальном стандартном размере, чтобы скриншоты хорошо помещались на странице.

Типичная перспектива для редактирования в Java выглядит как на рис. 5.5. В ней присутствуют виды для просмотра содержимого проектов, списка задач, вывода компоновщиков, а также результатов выполнения других операций и т. д.

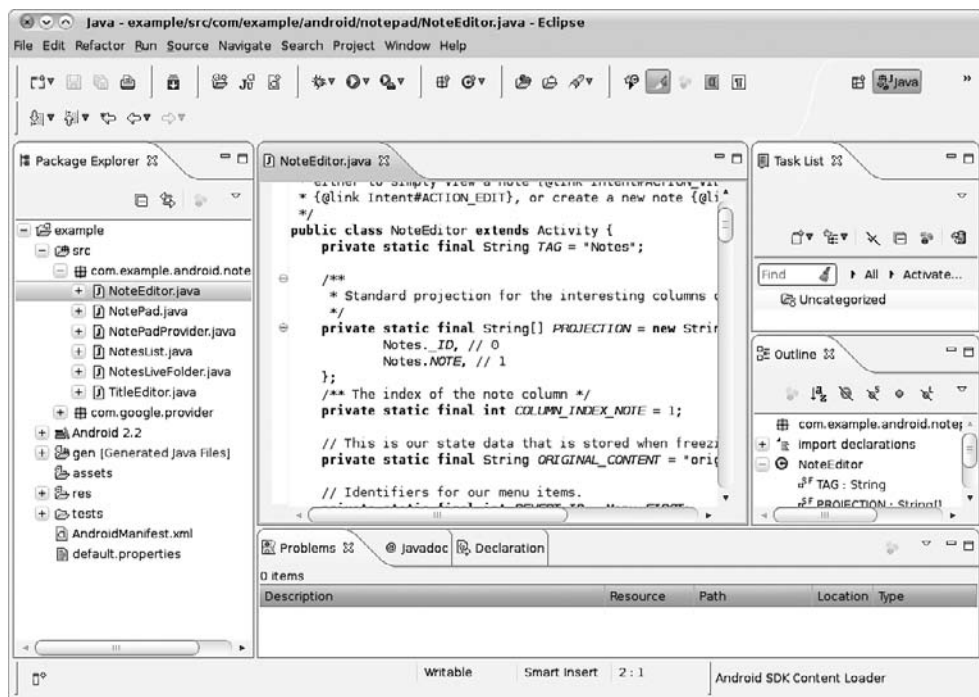


Рис. 5.5. Перспектива редактирования кода Java с проектом Android и исходным файлом Android в редакторе Java

Как видите, для создания проекта Android и редактирования исходного файла с кодом Java для этого проекта в стандартный набор видов вполне можно внести некоторые изменения. Рассмотрим виды, которые в таком случае отображаются по умолчанию.

Вид диспетчера пакетов

Eclipse — не просто программа-редактор, в окне которой присутствуют многочисленные панели инструментов. Большинство видов, отображаемых вокруг редактора в перспективе Eclipse, призваны ускорить навигацию по проекту и его файлам. Зачастую вашей отправной точкой при работе становится вид Package Explorer (Диспетчер пакетов). Здесь вы редактируете файлы с исходным кодом, запускаете и отлаживаете ваши проекты.

Вид списка задач

В этом виде (Task List) перечислены задачи. Они создаются командой **New Task** (Новая задача), расположенной на панели инструментов этого вида. Кроме того, в задачу можно превратить один из элементов, расположенных в виде **Problems** (Проблемы). Можно ссылкой связать список задач с репозиторием исходного кода (или с инструментом регистрации ошибок — **bug tracker**), чтобы решать задачу совместно с коллегами, которые, как и вы, заняты в данном проекте.

Интересно, что в списке задач не перечисляются элементы **TODO**, которыми многие разработчики пользуются в качестве «напоминалок» и которые вставляют прямо в код. Редактор **Java** производит синтаксический разбор этих элементов и помечает их ярлычками в левом поле. В таком списке могут содержаться, например, напоминания о внедрении плагинов, реализующих те или иные функции, если все задачи сложно перечислить в одном месте.

Вид Outline (Структура)

Программа — это одновременно и исходный код, который, как правило, представляет собой обычный текст, и структура, получаемая в результате его синтаксического разбора. В случае **Java** после синтаксического разбора получается структура, состоящая из полей и методов. В виде **Outline** (Структура) показана структура класса **Java**, и вы можете обрабатывать данную структуру при помощи многих из тех команд, которые применяются и к выборке, отображаемой в виде **Editor** (Редактор). Работа редактора **Java** построена на том, что системе известна базовая структура кода. Но в виде **Outline** (Структура) эта структура подробно раскладывается в иерархической форме вместе с ярлыками, сообщающими данные о типе и области действия слева от названия каждого из элементов, образующих такую структуру. Более подробная информация о виде **Outline** (Структура) находится по адресу <http://help.eclipse.org/helios/topic/org.eclipse.jdt.doc.user/reference/views/ref-view-outline.htm>.

Вид Problems (Проблемы)

В **Eclipse** существует концепция компоновщиков, обобщающая идею компилирования исходных файлов в объекты, или, по терминологии **Eclipse**, в артефакты. Проблемы — это обстоятельства и факторы, которые мешают такому процессу. К проблемам можно отнести ошибки компилятора, а также любые ошибки компоновщика. Иногда проблемы не позволяют компоновщику завершить построение артефакта, а в других случаях являются предупреждениями, которые тем не менее не мешают сгенерировать артефакт. В виде **Problems** (Проблемы) отображаются возникающие проблемы, и по их списку обеспечивается быстрая навигация. Если щелкнуть на проблеме правой кнопкой мыши, выводится контекстное меню. Если вы хотите сразу решить проблему, то команда **Go To** (Перейти) открывает файл с проблемой на нужной строке. Если вы сталкиваетесь с предупреждением, которое в итоге нужно исправить, вы можете отслеживать эту проблему при помощи команды **New Task From Marker** (Создать задачу из маркера). Если дважды щелкнуть кнопкой мыши на проблеме, вы также перейдете к ее источнику.

Написание кода Java в Eclipse

Если вы только начинаете работать с Java и Eclipse, то первым делом вам придется во всем разобраться. Но этот первый этап очень скоро закончится, и на первый план выйдет другая задача: писать код быстро и просто. Из всех языков программирования в Java, пожалуй, были предприняты самые значительные усилия по оптимизации труда программиста при помощи таких инструментов, как Eclipse. Поэтому история написания кода Java в Eclipse — это история стремления к максимально продуктивной работе. Существует три ключевых аспекта продуктивности: эффективное создание нового кода, нахождение кода, который нужно прочитать и модифицировать, и внесение в код изменений, затрагивающих более обширный фрагмент программы, чем та строка, над которой вы работаете в настоящий момент.

Редактирование кода Java и автозавершение

Одной из основных функций Eclipse, оптимизирующих редактирование кода Java в любой Java-ориентированной интегрированной среде разработки является автозавершение кода (autocompletion). В терминологии Eclipse эта функция называется content assist. Практически в любом месте исходного файла Java можно нажать сочетание клавиш **Ctrl+Пробел**. Оно выводит всплывающее окно, предлагающее дополнить текст, который вы сейчас вводите. Например, если вы знаете, что существует метод, позволяющий найти (find) что-либо, наберите `fi`, а потом нажмите **Ctrl+Пробел**. То, что вы увидите на экране, будет напоминать рис. 5.6.

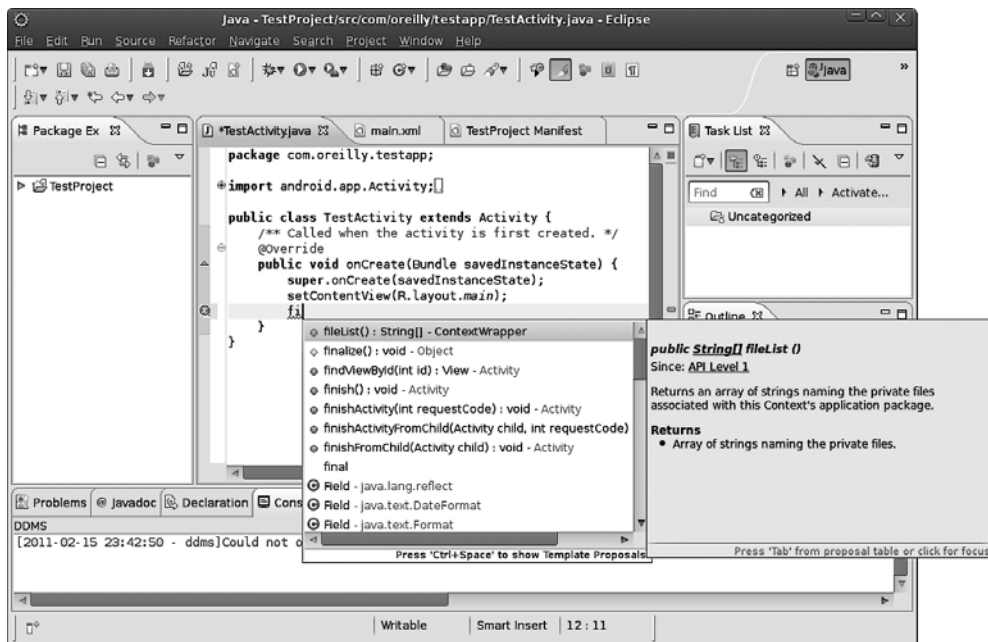


Рис. 5.6. Предложение автозавершения во всплывающем окне

В данном случае функция автозавершения предлагает вставить сигнатуру метода со списком параметров, который вам предлагается заполнить. Например, вы увидите в списке метод `findViewById` и сразу сможете его выбрать, а не вводить вручную название метода и список аргументов.

Если нажать **Ctrl+Пробел**, когда еще не введено никакого текста, в качестве возможных вариантов автозавершения будут предложены все константы и методы класса.

Рефакторинг

Как вы знаете, Java — это язык со статической типизацией, в котором требуется, чтобы все объекты и ссылки перед использованием были явно объявлены. Из-за этого язык Java можно обвинить в буквоедстве и неэlegantности и подумать, что код на Java излишне пространен. Такие интегрированные среды разработки, как Eclipse, облегчают проблемы, связанные с многословностью синтаксиса Java, **так как предлагают функцию автозавершения и другие возможности, ускоряющие работу программиста**. Но есть еще один фактор оптимизации продуктивности кода, который особенно хорош в языках со статической типизацией. Речь идет о рефакторинге.

Рефакторинг заключается во внесении изменений, не изменяющих поведения программы. То есть функциональность программы при рефакторинге остается такой же. Изменяется лишь организация программы. Если гарантировать, что при реорганизации кода поведение программы не изменится, то перед нами открываются возможности для грандиозных трансформаций. Ведь если пользоваться заменой текста, то даже переименование ссылки или изменение названия типа может пагубно сказаться на коде. А при рефакторинге можно быть уверенным, что изменятся только те имена, с которыми это необходимо сделать.

Существует два аспекта, которые значительно улучшают качество производимого рефакторинга. Во-первых, язык должен быть со статической типизацией, а во-вторых — в интегрированной среде разработки должна быть скомпилированная модель программы. Под моделью программы мы понимаем структуру данных, которая представляет собой скомпилированный код. Это такой код, в котором все типы и ссылки, присутствующие в программе, находятся в своей области видимости. Точно зная типы, а также область видимости типа или ссылки, интегрированная среда разработки может практически однозначно идентифицировать любой случай использования этого типа или этой ссылки.

Рефакторинг — яркий пример того, что языки уже нельзя сравнивать только по их синтаксису, эстетичности и выразительности. Такие консервативно построенные языки, как Java, **могут быть и максимально надежными, и в то же время максимально продуктивными** в контексте типичного инструментария, который программист всегда имеет под рукой.

Eclipse и Android

Плагин ADT добавляет в инструментарий Eclipse элементы, специфичные для Android. Большинство из этих инструментов находится в перспективе Android: Window ▸ Open Perspective ▸ Other... DDMS (Окно ▸ Открыть перспективу ▸ Прочие... DDMS). Правда, каждый элемент является отдельным видом Eclipse Window ▸ Open View ▸ Other... DDMS (Окно ▸ Открыть вид ▸ Прочие... DDMS), **и такой элемент мож-**

но добавить в любую другую перспективу так, как вам удобно, в зависимости от доступного свободного экранного пространства. Ниже перечислены некоторые наиболее удобные инструменты.

- LogCat — отображает журналы устройств в подокне, которое можно прокручивать. Можно настроить фильтрацию так, что будут видимы только те журналы, которые вас интересуют, либо так, что вы сможете отслеживать все, вплоть до сборки мусора и загрузки библиотек.
- File Explorer — показывает диспетчер файлов.
- Heap — отображает динамическую память.
- Threads — показывает потоки.
- Pixel Perfect — отображает вид Pixel Perfect (Попиксельное воспроизведение).
- Layout View — показывает вид Layout View (Структура).
- avdmgrr — отображает Android SDK и диспетчер виртуальных устройств Android.

Предотвращение ошибок и поддержание чистоты кода

Eclipse можно считать специализированной операционной системой: она состоит из тысяч файлов, обладает собственной файловой системой, а при работе задействует веб-сервер. Eclipse — открытая система, в которую можно вносить разнообразные дополнения. Плагины, которые играют роль приложений в операционной системе Eclipse, относительно просто пишутся, а во всей экосистеме Eclipse насчитывается гораздо больше расширений, чем может установить и применять какой-либо пользователь. Поскольку код для Android пишется на языке Java, при разработке программ для Android вы можете пользоваться любыми плагинами.

Ниже мы исследуем особенно полезную категорию расширений Eclipse. Речь пойдет о статических анализаторах, которые также можно назвать анализаторами исходного кода.

Статические анализаторы

Если попытаться предельно просто определить статический анализ, то его можно считать процессом, который разворачивается там, где компилятор оставил предупреждения. В Eclipse предупреждения компилятора, в принципе, очень помогают. Если компилятор хороший, он выдаст вам такие предупреждающие сообщения, которые пригодятся при отслеживании потенциальных проблем, которые могут возникнуть во время исполнения. Но тем не менее компилятор не обязан заниматься поиском скрытых проблем. Этим занимаются как раз статические анализаторы.

Данные анализаторы называются статическими потому, что обрабатываемый ими код в момент анализа не работает. Хотя компилятор и выполняет некоторые функции, которые можно отнести к компетенции статического анализатора (а компилятор для Java в Eclipse очень хорошо подчищает всякие огрехи за программистом, например удаляет неиспользуемые переменные и методы), статические анализаторы явно сильнее в этом отношении. Статические анализаторы «заточены» под поиск ошибок, а не просто мелких недоработок.

FindBugs

Начнем изучение статических анализаторов с установки и использования FindBugs. Документация, а также исходный код к FindBugs находится по адресу <http://findbugs.sourceforge.net>. Мы подробно рассмотрим процесс установки, так как он подобен установке любых плагинов для Eclipse. Для установки Find Bugs нужно сначала добавить репозиторий Find Bugs в список сайтов Eclipse, из которого следует устанавливать пакеты. Это делается командой меню Help ► Install New Software (Помощь ► Установить новую программу). Затем нужно нажать кнопку Add (Добавить) в диалоговом окне Install (Установка). Откроется диалоговое окно Add Repository (Добавить репозиторий), в котором можно добавить репозиторий FindBugs, расположенный по адресу <http://findbugs.cs.umd.edu/eclipse> (рис. 5.7).

Следующим шагом при установке FindBugs является установка пакета из репозитория (рис. 5.8). В данном случае на выбор предлагается всего один пакет.

После выбора пакета можно переходить к следующему диалоговому окну, в котором отображается список пакетов для установки (рис. 5.9).

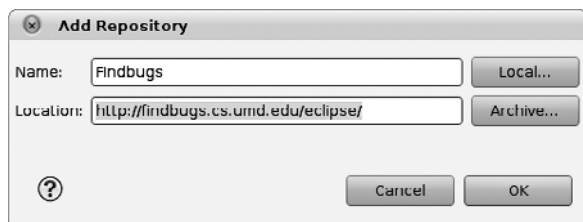


Рис. 5.7. Добавление репозитория, необходимое для последующего добавления плагина в окружение Eclipse



Рис. 5.8. Выбор единственного доступного пакета из репозитория FindBugs

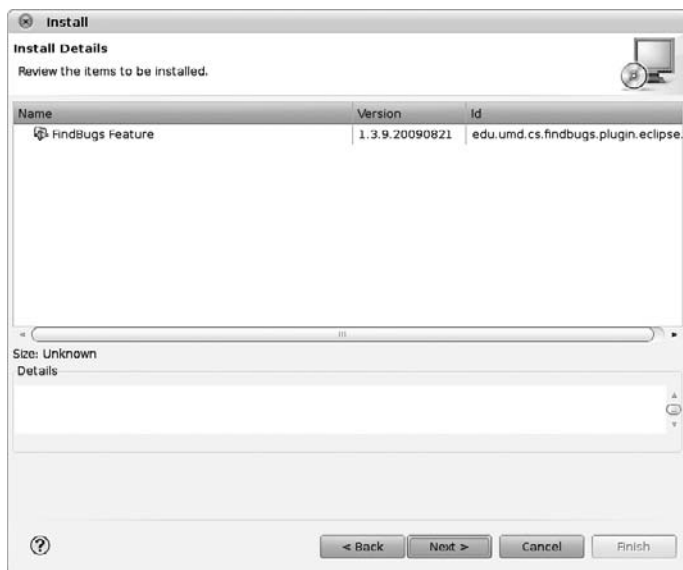


Рис. 5.9. Убеждаемся, что выбран единственный пакет, доступный в репозитории FindBugs

В следующем диалоговом окне предлагается прочитать лицензионное соглашение, прилагаемое к этому пакету, а затем принять или не принять его (рис. 5.10).

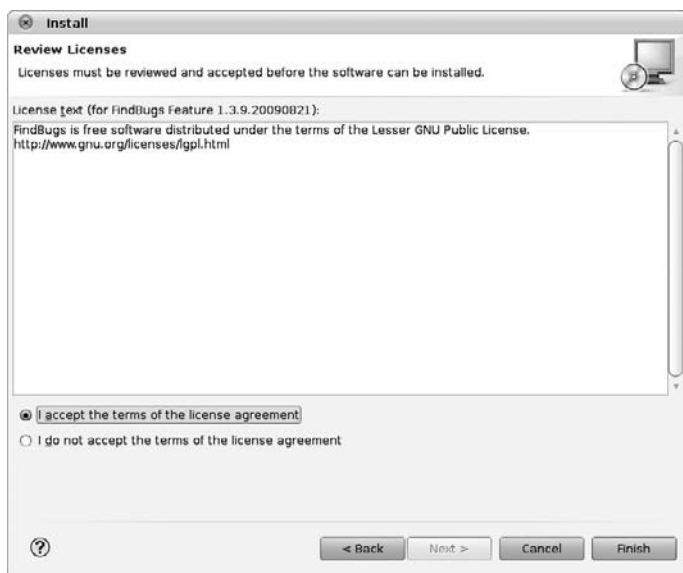


Рис. 5.10. Принимаем лицензионное соглашение FindBugs

При установке этого плагина Eclipse, возможно, возникнет еще одна преграда. Поскольку пакет не подписан, вы получите предупреждение о том, что он небезопасен (рис. 5.11).

После запуска FindBugs можно перейти в перспективу этой программы (рис. 5.14). В перспективу FindBugs включены виды, отображающие иерархический список потенциальных проблем, найденных FindBugs. Проблемы сгруппированы по типам. В виде Editor (Редактор) имеются маркеры для обозначения проблем. Если открыть свойства проблемы, программа выведет ее подробное описание, в том числе объяснит, почему FindBugs может давать «ложноположительные» заключения.

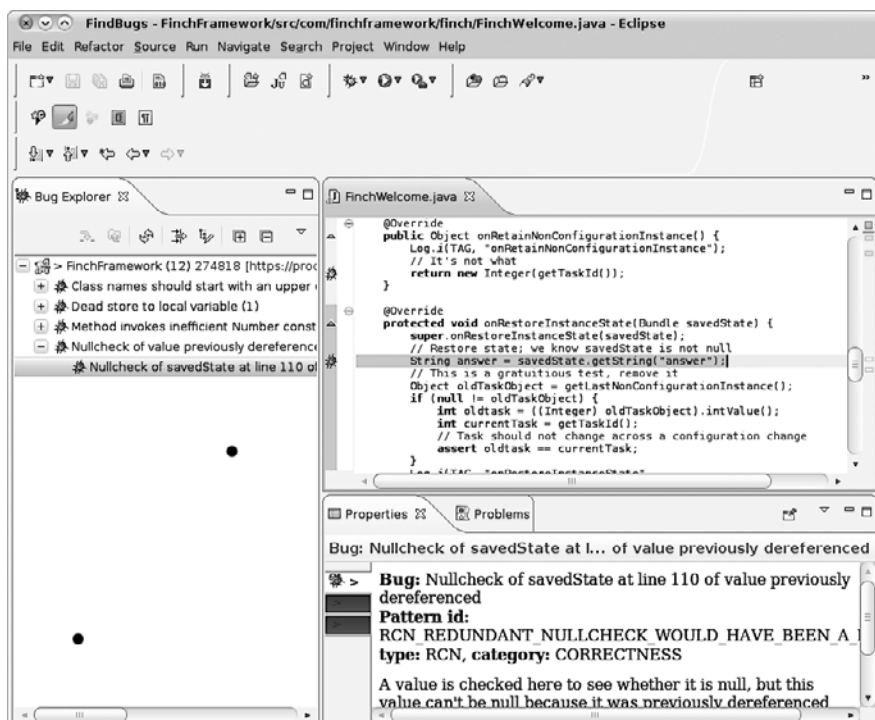


Рис. 5.14. Перспектива FindBugs

В данном случае рассмотрим проблему Null check of a value previously dereferenced (Проверка на ноль ссылки, которая была разыменована), показанную в окне Bug Explorer (Обозреватель ошибок) (рис. 5.15).

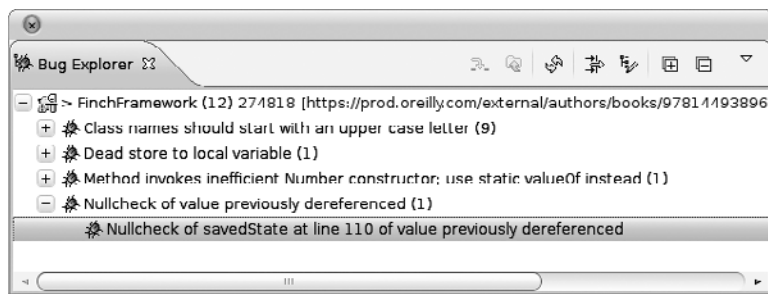


Рис. 5.15. Обозреватель ошибок в программе FindBugs

Проверка на ноль значения, введенного в поле, которая происходит после того, как ссылка на это значение уже разыменована, в синтаксисе Java не возбраняется, но такая ситуация практически наверняка бесполезна или является очевидной ошибкой. В следующем коде видно, что поле `savedState` используется с явным расчетом на то, что оно никогда не должно быть равно нулю, но проверка на ноль происходит в ходе выполнения записи в журнал операций:

```
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    // Восстановление состояния: известно, что savedInstanceState не равно нулю.
    String answer = savedInstanceState.getString("answer");
    // В этой проверке нет необходимости; удалить ее.
    Object oldTaskObject = getLastNonConfigurationInstance();
    if (null != oldTaskObject) {
        int oldtask = ((Integer) oldTaskObject).intValue();
        int currentTask = getTaskId();
        // Задача не должна изменяться при изменении конфигурации.
        assert oldtask == currentTask;
    }
    Log.i(TAG, "onRestoreInstanceState"
        + (null == savedInstanceState ? "" : RESTORE) + " " + answer);
}
```

На самом деле проверка `savedState` на ноль должна происходить до его использования, так как не указано, что значение `savedState` должно быть ненулевым. Мы изменим операцию присваивания, в которой не происходит проверка `savedState` на ноль, на следующую:

```
String answer = null != savedInstanceState ? savedInstanceState.getString("answer") : "";
```

Снова запускаем `FindBugs` и убеждаемся, что это изменение снимает возможную проблему.

Это хороший пример ошибки, которую можно найти при статическом анализе. О такой ошибке вас не предупредит компилятор, поскольку возможны случаи, когда программист хочет сделать в коде именно такой ход. Но обычный логический вывод позволяет статическому анализатору предположить, что он столкнулся с ошибкой. И действительно, в такой ситуации он довольно часто оказывается прав.

Ограничения, связанные со статическим анализом

Статические анализаторы зачастую дают ложноположительные результаты. Это объясняется теми методами, которые применяются в таких программах для нахождения уязвимостей в коде. В этом отношении результаты статического анализа не похожи на предупреждения компилятора. Если сообщение компилятора квалифицирует ситуацию как ошибку, то она будет считаться ошибкой, хотя фактически такая ситуация может не представлять никакой проблемы.

Одной из наиболее слабых сторон статических анализаторов является поиск кода, в котором не соблюдаются соглашения по написанию кода¹ (*coding conven-*

¹ Встречается также перевод «соглашения по программированию». — *Примеч. пер.*

tions). Например, предупреждение `Class names should start with an upper case letter` (Имена классов всегда должны начинаться с заглавной буквы), показанное на рис. 5.15, спровоцировано автоматически сгенерированным кодом, который программист вообще не проверяет — за исключением случаев, когда подозревается ошибка в самом генераторе кода.

Многие опытные программисты зачастую скептически относятся к статическим анализаторам, поскольку в создаваемом ими коде содержится относительно мало проблем, которые такой анализатор способен идентифицировать. Поэтому анализ дает лишь множество ложноположительных результатов. Общеизвестно, что при статическом анализе кода, написанного опытным программистом, удастся найти лишь часть багов. Следовательно, статический анализ не заменяет модульного тестирования и хороших навыков проведения отладки. Однако если вы относительно недолго работаете с Java, а также с Android, то статические анализаторы будут для вас значительным подспорьем, наряду с теми предупреждениями, которые выдает компилятор.

Характерные особенности Eclipse и альтернативные инструменты

Теперь, когда нам известно, что многие возможности SDK Android основаны на Eclipse, и мы знаем, как плагин Eclipse и архитектура расширений позволяют инструментам Android «захватывать» такое множество функций интегрированной среды разработки, может возникнуть вопрос: зачем же предлагается альтернативный вариант запуска приложения Android на сервере или в качестве апплета? Особенно досадно то, что инструмент, который, казалось бы, должен оптимизировать работу, попросту сбивает нас с толку. Ведь при работе с Eclipse просто необходимо быстро находить нужные команды в крайне длинных меню.

Далее выполним следующую операцию и посмотрим, что произойдет. Выберите любой проект Android в рабочем пространстве Eclipse, щелкнув правой кнопкой мыши на имени этого проекта, а затем выполните **Run As ▶ Java Applet** (Запустить как ▶ Апплет Java). Откроется диалоговое окно, показанное на рис. 5.16.

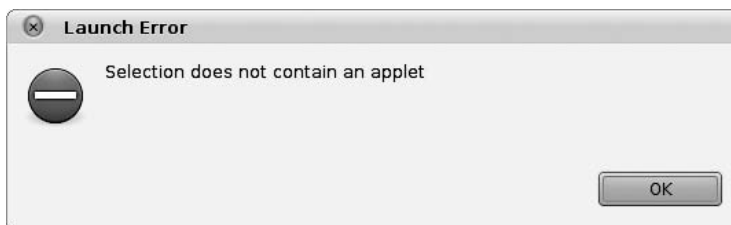


Рис. 5.16. Это диалоговое окно отображается, когда в выборке не содержится ни одного апплета

Ничего страшного, но, как говорится, осадок остается. Eclipse, а также все плагины, задействованные в определенный момент, не должны предлагать вам совершить

операцию, которая гарантированно закончится ошибкой. Eclipse — плохой пример, избавьте пользователей ваших приложений для Android от таких ситуаций. Если в выборке не содержится апплетов, то не предлагайте пользователю команду, которая могла бы запустить выборку как апплет. Это фундаментальный принцип, действующий при разработке графических пользовательских интерфейсов, а также базовая идея программирования типичных (generic) операций, применяемых к выборке. Как только пользователь совершит выбор, программе должны быть известны все варианты операций, которые могут быть применены при данном выборе, а остальные операции должны игнорироваться. Хороший интерфейс — особенно если речь идет о большом, сложном интерфейсе — не отпугивает пользователя, а вдохновляет его на исследование программы, которое должно проходить спокойно и предсказуемо.

Почему в пользовательском интерфейсе Eclipse обнаруживаются такие, казалось бы, тривиальные недоработки? Подобные вещи особенно раздражают, если сравнить их с огромной мощностью Eclipse и простотой, с которой в этой среде организованы рефакторинг и другие функции. В основном Eclipse решительно безупречна. Можно предположить, что выверенная до тонкостей модульность Eclipse, которая неизбежно приводит к лавинообразному разрастанию интерфейсов расширений программы, и вызывает такую комбинацию мощного функционала и мелких неудобств. Меньшее зло заключается в том, что разработчикам плагинов необходимо сочленять модуль с огромным количеством интерфейсов, чтобы пользователю было удобно работать с новым плагином. Большее зло — приходится мириться с тем, что в некоторых случаях Eclipse просто не приспособлена для выполнения нужной операции. Вот почему некоторые специалисты подыскивают альтернативные инструменты.

ЧАСТЬ II

Фреймворк Android

Глава 6. Создание вида

Глава 7. Фрагменты и многоплатформенная поддержка

Глава 8. Рисование двухмерной и трехмерной графики

Глава 9. Обращение с данными и их долговременное хранение

Фреймворк Android — это набор базовых классов, лежащих в основе приложений Android и компонентов системных программ Android. Вместе они дают совокупность приложений и библиотек (userland) Android. В этой части книги различные API Android будут представлены так, чтобы стало понятнее, как мы собираемся достигать нашей основной цели — а именно внедрять приложения, которые максимально эффективно используют системную архитектуру Android.

6 Создание вида

В Android существует немало требований, соблюдение которых неизбежно приводит к усложнению внутренней организации пользовательского интерфейса. Пользовательский интерфейс в Android — это **многопроцессная система**, поддерживающая множество параллельно исполняемых приложений, принимающая разнообразные варианты ввода, исключительно интерактивная и достаточно гибкая для того, чтобы поддерживать самые разнообразные устройства как в настоящее время, так и в будущем. Пользовательский интерфейс Android **одновременно является и насыщенным, и простым** в обращении.

В этой главе будут рассмотрены приемы реализации графического пользовательского интерфейса в Android. Здесь будет объяснена архитектура инструментария для написания пользовательских интерфейсов Android, на практических примерах будет показано, как используются простейшие элементы интерфейса, в частности кнопки и текстовые поля. Кроме того, здесь мы поговорим об обработке событий, использовании нескольких потоков для разгрузки долговременных задач, чтобы пользовательский интерфейс не зависал, и коснемся других вопросов, помогающих сделать пользовательский интерфейс красивым и эффективным.

Архитектура графического пользовательского интерфейса в Android

Среда Android добавляет в экосистему Java еще один инструментарий для создания графических пользовательских интерфейсов, дополнительно к AWT, Swing, SWT, LWUIT и др. Если вы работали с какими-либо из этих технологий, фреймворк пользовательского интерфейса Android **покажется вам знакомым. Как и вышеперечисленные системы, он однопоточный, событийно-управляемый и основан на библиотеке вкладываемых друг в друга компонентов.**

Фреймворк пользовательского интерфейса Android, как и другие фреймворки пользовательских интерфейсов Java, **организован на базе распространенного паттерна «Модель-вид-контроллер»**, который схематически изображен на рис. 6.1. Здесь предоставляются инструменты и обеспечивается структура для построения контроллера, обрабатывающего пользовательский ввод (например, нажатия клавиш

или прикосновения к экрану), а также вида, который отображает на экране графическую информацию.

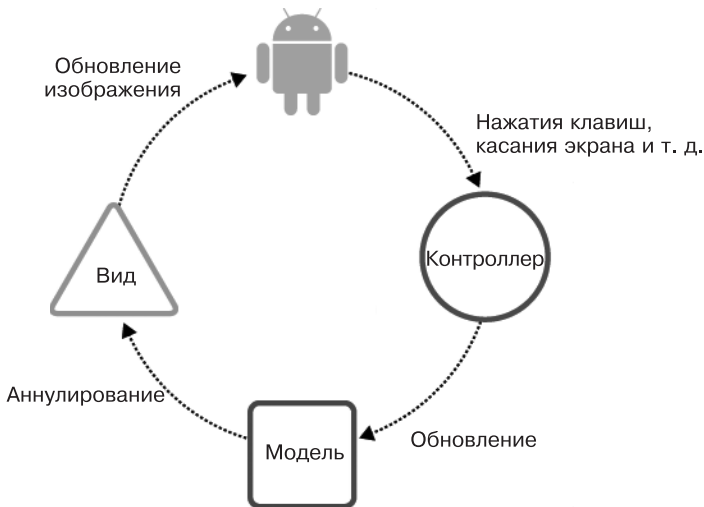


Рис. 6.1. Концепция «Модель-вид-контроллер»

Модель

Модель — это сущность вашего приложения, то, чем фактически занимается программа. Например, модель может представлять собой базу данных музыкальных файлов, записанных на вашем устройстве, и код для воспроизведения музыки. Или это может быть ваш список контактов и код для выполнения телефонных вызовов, а также отправки мгновенных сообщений. В целом модели посвящена значительная часть последующего материала книги.

В то время как вид и контроллер отдельно взятого приложения обязательно будут отражать свойства модели, с которой они работают, отдельно взятая модель может использоваться несколькими приложениями. Рассмотрим, например, MP3-плеер и приложение, преобразующее файлы формата MP3 в формат WAV. Модель обоих приложений включает в себя формат файлов MP3. Однако в первом приложении есть привычные элементы управления в виде кнопок «Стоп», «Пуск» и «Пауза», и оно воспроизводит звуки. Второе может не издавать вообще никаких звуков. Зато у него будут элементы для управления таким показателем, как скорость передачи информации (битрейт). Модель — это в первую очередь сущность для работы с данными.

Вид

Вид — это визуализация модели. В качестве общего определения вид можно охарактеризовать как часть приложения, отвечающую за вывод изображения, отправку аудиоинформации в динамик, генерирование отклика на прикосновение и т. д. Графическая часть фреймворка пользовательского интерфейса Android, подробно описанная в главе 8, реализуется как дерево подклассов класса View. Графически

каждый объект представляет собой прямоугольную область на экране, которая полностью вписана в прямоугольную область своего родительского элемента в дереве подклассов. Корень этого дерева — это окно приложения.

Например, изобразительным компонентом гипотетического МРЗ-плеера может быть эскиз обложки альбома, композиция из которого сейчас воспроизводится. Другой вид может отображать название песни, воспроизводимой в данный момент, а третий — содержать более мелкие виды, например кнопки «Стоп», «Пуск» и «Пауза».

Фреймворк пользовательского интерфейса наполняет экран изображениями, проходя по дереву видов и приказывая каждому компоненту отрисоваться в порядке прямого обхода (алгоритм «посетить корень, обойти левое поддерево, обойти правое поддерево»). Иными словами, каждый вид отрисовывает себя, а потом приказывает всем своим дочерним видам сделать то же самое. Когда отобразится все дерево, более мелкие, вложенные компоненты, которые можно сравнить с листьями дерева (и которые поэтому отображаются в последнюю очередь), оказываются выше компонентов, расположенных ближе к корню и отрисовываемых раньше.

На практике фреймворк пользовательского интерфейса **Android** более эффективен, чем можно представить себе по данному упрощенному описанию. Он не отрисовывает область родительского вида, если есть гарантия, что позже эту область отрисует какой-нибудь из дочерних элементов. Было бы напрасной тратой времени отрисовывать фон под непрозрачным объектом. Кроме того, напрасной была бы работа по перерисовке частей вида, которые не изменяются.

Контроллер

Контроллер — это часть приложения, отвечающая на внешние воздействия, например на нажатие клавиши, на прикосновение к экрану, на входящий вызов и т. д. Контроллер реализуется в виде *очереди событий* (event queue). Каждое внешнее действие представляется как уникальное событие в очереди. Фреймворк по порядку удаляет события из очереди и распределяет (диспетчирует) их.

Например, когда пользователь нажимает какую-либо клавишу телефона, система Android генерирует событие `KeyEvent` и добавляет его в очередь событий. После того как закончится обработка событий, попавших в очередь ранее, `KeyEvent` удаляется из очереди и передается в качестве параметра вызова метода `dispatchKeyEvent` того вида `View`, который выбран в данный момент.

После того как событие отправлено в компонент, который находится в фокусе, этот компонент может совершить действие, необходимое для изменения внутреннего состояния программы. Например, в приложении, представляющем собой МРЗ-плеер, когда пользователь нажимает на экране кнопку «Пуск/Пауза» и событие направляется к объекту этой кнопки, метод-обработчик может обновить модель для возобновления проигрывания какого-то выбранного ранее звукового файла.

В этой главе описывается конструкция контроллера для приложения Android.

Все вместе

Итак, мы познакомились со всеми концепциями, необходимыми для описания всей системы пользовательского интерфейса. Когда происходит внешнее воздействие — пользователь прокручивает экран, перетаскивает элементы или нажимает кнопку

либо, например, поступает входящий вызов или МРЗ-плеер достигает конца списка воспроизведения, — система Android ставит событие, представляющее данное действие, в очередь событий. Наконец событие покидает очередь — по принципу «что раньше пришло, то раньше обслуживается» — и направляется системой к подходящему обработчику событий. Обработчик событий — зачастую фрагмент кода, который вы пишете для вашего приложения, — реагирует на событие, уведомляя модель, что произошло изменение состояния. Модель в ответ предпринимает необходимое действие.

Практически любое изменение состояния модели требует соответствующего изменения в виде. Например, в ответ на нажатие клавиши компонент `EditText` должен отобразить только что введенный символ в точке вставки. Аналогично в приложении «Телефонный справочник» при нажатии контакта этот контакт будет подсвечен, а контакт, который был подсвечен ранее, — померкнет.

Когда модель обновляет собственное состояние, она практически наверняка должна будет изменить текущее отображение, чтобы отразить, таким образом, произошедшие внутренние изменения. Чтобы обновить изображение на дисплее, модель должна уведомить фреймворк пользовательского интерфейса о том, что какая-то часть изображения, присутствующего на дисплее, уже устарела и должна быть перерисована. Запрос на перерисовку — не что иное, как другое событие, поставленное в очередь событий в том же фреймворке, где только что находилось событие контроллера. Событие перерисовки обрабатывается в общем порядке, как и любое другое событие пользовательского интерфейса.

Рано или поздно событие перерисовки удаляется из очереди и направляется обработчику. Обработчиком событий перерисовки является `View`. Дерево видов перерисовывается; каждый вид отвечает за отображение собственного состояния в тот момент, когда он отрисовывается.

Чтобы конкретизировать все сказанное, проследим описанный цикл в гипотетической программе для воспроизведения МРЗ.

1. Когда пользователь нажимает на экране изображение кнопки «Пуск/Пауза», фреймворк создает новое событие `MotionEvent`, в котором среди прочего содержится информация о координатах точки на экране, где произошло нажатие. Фреймворк ставит новое событие в хвост текущей очереди событий.
2. Как было описано в подразделе «Контроллер» выше, когда событие доходит до конца очереди, фреймворк удаляет его из этой очереди и передает по дереву видов к тому виджету («листу» этого дерева), который занимает прямоугольную область на экране, где и произошло нажатие.
3. Поскольку данный виджет представляет собой кнопку «Пуск/Пауза», код приложения, обрабатывающий нажатие кнопки, сообщает модели, что следует возобновить воспроизведение звукового файла.
4. Код модели приложения начинает воспроизводить выбранный звуковой файл. Кроме того, он посылает во фреймворк пользовательского интерфейса запрос на перерисовку изображения на экране.
5. Запрос на перерисовку добавляется в очередь событий и, наконец, обрабатывается, как это описано в подразделе «Вид» выше.
6. Экран перерисовывается. При этом кнопка переходит в состояние «Пуск» и все снова оказывается синхронизированным.

Объекты, являющиеся компонентами пользовательского интерфейса, например кнопки и текстовые поля, на самом деле реализуют и методы вида, и методы контроллера. Только так и нужно делать. Когда вы добавляете к пользовательскому интерфейсу вашей программы кнопку (Button), то хотите, чтобы она появилась на экране, а также чтобы она выполняла какое-то действие, когда пользователь ее нажимает. Даже притом, что два логических элемента пользовательского интерфейса, и вид и контроллер, реализуются на одном объекте, нужно позаботиться, чтобы между ними не было непосредственного контакта. Например, методы контроллера никогда не должны напрямую изменять изображение. Пусть код, который изменяет состояние, сначала посылает запрос на перерисовку, а потом вызывает методы отображения, которые позволяют компоненту отразить новое состояние. Если писать код таким образом, то сводятся к минимуму проблемы с синхронизацией, программа остается надежной и убергается от ошибок.

Необходимо заострить внимание еще на одном важном аспекте пользовательского интерфейса Android: он однопоточный. Единственный поток удаляет события из очереди, чтобы делать обратные вызовы контроллера и отображать вид. Это важно по нескольким причинам.

Простейшее следствие применения однопоточного пользовательского интерфейса заключается в том, что при координации состояний вида и контроллера можно обойтись без блоков `synchronized`. Это очень ценная оптимизация.

Еще одно достоинство однопоточного пользовательского интерфейса заключается в том, что в нашем приложении каждое событие, находящееся в очереди, гарантированно будет обработано полностью и именно в том порядке, в каком оно поступило в очередь. Это может казаться совершенно очевидным, но при таком условии написание кода для пользовательского интерфейса значительно упрощается. Когда компонент пользовательского интерфейса вызывается для обработки события, мы гарантируем, что не начнется никакая другая работа, связанная с пользовательским интерфейсом, пока не будет получен ответ на первый вызов. То есть, например, если компонент требует сделать несколько изменений в состоянии программы — каждое из которых дает соответствующий запрос на перерисовывание части экрана, — то гарантируется, что перерисовка *не* начнется до тех пор, пока программа не закончит обработку, не выполнит всех обновлений и не вернет ответ. Короче говоря, обратные вызовы пользовательского интерфейса не зависят друг от друга.

Третья причина, по которой не следует забывать, что только один поток занимается изъятием и распределением событий пользовательского интерфейса из их очереди, заключается в том, что, если ваш код по каким-то причинам остановит выполнение этого потока, пользовательский интерфейс зависнет! Если отклик компонента на событие прост (например, сводится к изменению состояния переменных, созданию новых объектов и т. д.), то совершенно оправданно обрабатывать такие действия в основном потоке событий. Если же, напротив, обработчик должен получить ответ от какой-нибудь удаленной сетевой службы или запустить сложный запрос к базе данных, то весь пользовательский интерфейс застынет, пока не будет получен ответ на запрос. Определенно работать с таким интерфейсом будет неудобно! Если на выполнение задачи уходит много времени, ее нужно делегировать другому потоку, как это описано в подразделе «Продвинутые способы подключения: фокус и поточность» раздела «Множественные указатели и жесты» данной главы.

Сборка графического интерфейса

Во фреймворке пользовательского интерфейса Android предоставляется и полный набор инструментов рисования, с помощью которых этот интерфейс строится, и богатая коллекция заготовленных компонентов, основанных на этих инструментах. Как будет показано в главе 8, графические инструменты фреймворка обеспечивают широкую поддержку приложений, в которых требуется создавать собственные элементы управления или отображать специфические виды. Многие приложения могут вполне качественно работать, используя только стандартные виды из инструментария. На самом деле классы `MapActivity` и `MyLocationOverlay` позволяют создавать самые изысканные приложения, при этом вам не придется вообще ничего рисовать.

Пару раз мы уже употребляли термин «*виджет*», не объясняя, что это именно такое. Напомним, что экран отображается деревом компонентов. Во фреймворке пользовательского интерфейса Android все эти компоненты являются подклассами `android.view.View`. Виды, являющиеся «листьями» или почти «листьями», выполняющую основную рисовальную работу, и в контексте пользовательского интерфейса приложения такие виды обычно именуются виджетами. Как было указано выше, виджет обычно реализует как функционал вида, так и функционал контроллера.

Внутренние узлы, иногда называемые *контейнерными видами* (container views), — это особые компоненты, которые обладают другими, дочерними компонентами. Во фреймворке пользовательского интерфейса Android **контейнерные виды являются подклассами `android.view.ViewGroup`**, а этот класс, в свою очередь, приходится подклассом `View`. Обычно здесь происходит достаточно мало рисования. Вместо этого контейнерные виды занимаются упорядочением своих дочерних видов на экране и сохранением их порядка по мере того, как вид изменяет контуры, ориентацию и т. д. Выполнять такие задачи бывает непросто.

Для создания сложных дисплеев вам потребуется собрать дерево контейнеров для видов, которые вы собираетесь использовать в своем приложении. В примере 6.1 показано приложение с деревом видов глубиной в три уровня. Вертикальный линейный макет содержит два горизонтальных линейных макета. Каждый горизонтальный макет, в свою очередь, содержит два виджета.

Пример 6.1. Сложное дерево видов

```
package com.oreilly.android.intro;
```

```
import android.app.Activity;
import android.graphics.Color;
import android.os.Bundle;
import android.view.Gravity;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.EditText;
import android.widget.LinearLayout;
```

```
public class AndroidDemo extends Activity {
    private LinearLayout root;
```

```
    @Override
    public void onCreate(Bundle state) {
```

```
super.onCreate(state);

LinearLayout.LayoutParams containerParams
    = new LinearLayout.LayoutParams(
        ViewGroup.LayoutParams.FILL_PARENT,
        ViewGroup.LayoutParams.WRAP_CONTENT,
        0.0F);

LinearLayout.LayoutParams widgetParams
    = new LinearLayout.LayoutParams(
        ViewGroup.LayoutParams.FILL_PARENT,
        ViewGroup.LayoutParams.FILL_PARENT,
        1.0F);

root = new LinearLayout(this);
root.setOrientation(LinearLayout.VERTICAL);
root.setBackgroundColor(Color.LTGRAY);
root.setLayoutParams(containerParams);

LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.HORIZONTAL);
ll.setBackgroundColor(Color.GRAY);
ll.setLayoutParams(containerParams);
root.addView(ll);

EditText tb = new EditText(this);
tb.setText(R.string.defaultLeftText);
tb.setFocusable(false);
tb.setLayoutParams(widgetParams);
ll.addView(tb);

tb = new EditText(this);
tb.setText(R.string.defaultRightText);
tb.setFocusable(false);
tb.setLayoutParams(widgetParams);
ll.addView(tb);

ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.HORIZONTAL);
ll.setBackgroundColor(Color.DKGRAY);
ll.setLayoutParams(containerParams);
root.addView(ll);

Button b = new Button(this);
b.setText(R.string.labelRed);
b.setTextColor(Color.RED);
b.setLayoutParams(widgetParams);
ll.addView(b);

b = new Button(this);
b.setText(R.string.labelGreen);
b.setTextColor(Color.GREEN);
```

```

        b.setLayoutParams(widgetParams);
        ll.addView(b);

        setContentView(root);
    }
}

```

Обратите внимание: в коде сохранена ссылка на корень дерева видов — для последующего использования.

В данном примере применяются три вида `LinearLayout`. Вид `LinearLayout`, как следует из его названия¹, отображает свои дочерние элементы в виде строки или столбца, это определяется свойством его ориентации. Дочерние виды отображаются в том порядке, в каком они *добавлялись* к `LinearLayout` (независимо от порядка, в котором они *создавались*), в привычном для европейского читателя направлении слева направо и сверху вниз. Например, кнопка **Green** (Зеленый) находится в правом нижнем углу этого макета, так как она была добавлена второй к горизонтальному виду `LinearLayout`. Он, в свою очередь, является вторым элементом, добавленным к корневому виду `LinearLayout`.

На рис. 6.2 показано, как все это увидит пользователь. Семь видов дерева структурированы так, как изображено на рис. 6.3.



Рис. 6.2. Как зритель видит панели



Рис. 6.3. Иерархия панелей вида

¹ В переводе — «линейный макет» или «линейная компоновка». — *Примеч. пер.*

Во фреймворке Android хорошо организована возможность отделения ресурсов данных от кода. Она, в частности, полезна при построении компоновки видов. Предыдущий пример можно заменить радикально более простым кодом из примера 6.2 и XML-определением компоновки вида из примера 6.3.

Пример 6.2. Сложный вид, использующий файл разметки формы (layout resource)
package com.oreilly.android.intro;

```
import android.app.Activity;
import android.os.Bundle;

/**
 * Демонстрационная версия пользовательского интерфейса Android.
 */
public class AndroidDemo extends Activity {
    private LinearLayout root;

    @Override public void onCreate(Bundle state) {
        super.onCreate(state);
        setContentView(R.layout.main);
        root = (LinearLayout) findViewById(R.id.root);
    }
}
```

Пример 6.3. XML-определение файла разметки формы сложного вида

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/root"
    android:orientation="vertical"
    android:background="@drawable/lt_gray"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <LinearLayout
        android:orientation="horizontal"
        android:background="@drawable/gray"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <EditText
            android:id="@+id/text1"
            android:text="@string/defaultLeftText"
            android:focusable="false"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_weight="1"/>

        <EditText
            android:id="@+id/text2"
            android:text="@string/defaultRightText"
            android:focusable="false"
```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"/>
</LinearLayout>

<LinearLayout
    android:orientation="horizontal"
    android:background="@drawable/dk_gray"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <Button
        android:id="@+id/button1"
        android:text="@string/labelRed"
        android:textColor="@drawable/red"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"/>

    <Button
        android:id="@+id/button2"
        android:text="@string/labelGreen"
        android:textColor="@drawable/green"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"/>
</LinearLayout>
</LinearLayout>
```

В данной версии кода, как и в первой, также сохраняется ссылка на корень дерева видов. Это делается с помощью отметки виджета в XML-шаблоне (в данном случае речь о корневом `LinearLayout`) тегом `android:id`. После этого используется метод `findViewById` класса `Activity`, получающий ссылку.

Рекомендуется определять компоновку дерева видов именно в таком файле разметки формы — лучше, если это войдет в привычку. Поступая так, вы отделяете визуальное представление вида от того кода, который заставляет его работать. Можно повозиться с компоновкой на экране, и для этого не требуется перекомпилировать код. Но важнее всего то, что вы сможете строить ваш пользовательский интерфейс при помощи таких инструментов, которые позволяют создавать экраны виды прямо в специальном визуальном редакторе пользовательских интерфейсов.



На конференции Google I/O 2011 команда разработчиков инструментов Android представила новый редактор макетов, который сразу же вызвал огромный интерес. В нем даже обеспечивается предварительный просмотр анимации и видов, созданных программистом; большинству разработчиков даже не придется заглядывать в XML, не говоря уж о встроенном коде, перекомпоновывая свои виды сколько угодно.

Подключение контроллера

В разделе «Сборка графического интерфейса» выше мы показали вид с двумя кнопками. Конечно, это симпатичные кнопки, но они бесполезны. При их нажатии ничего не происходит.

В подразделе «Контроллер» предыдущего раздела было описано, как фреймворк Android преобразует внешние воздействия (прикосновения к экрану, нажатия клавиш и т. д.) в события, которые выстраиваются в очередь, а потом передаются приложению. В примере 6.4 показано, как добавить обработчик событий к одной из кнопок демонстрационного примера так, чтобы при нажатии происходило определенное действие.

Пример 6.4. Подключение кнопки

```
@Override public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    final EditText tb1 = (EditText) findViewById(R.id.text1);
    final EditText tb2 = (EditText) findViewById(R.id.text2);

    ((Button) findViewById(R.id.button2)).setOnClickListener(
        new Button.OnClickListener() {
            // mRand – член класса данных
            @Override public void onClick(View arg0) {
                tb1.setText(String.valueOf(mRand.nextInt(200)));
                tb2.setText(String.valueOf(mRand.nextInt(200)));
            }
        }
    );
}
```

После запуска такой вариант приложения по-прежнему очень напоминает рис. 6.2. Но, в отличие от предыдущего примера в данной версии, всякий раз, когда пользователь нажимает кнопку **Green** (Зеленый), цифры в полях EditText изменяются (рис. 6.4).



Рис. 6.4. Рабочая кнопка

Обычное изменение цифр — не такая уж интересная функция, но этот небольшой пример демонстрирует стандартный механизм, при помощи которого

приложение реагирует на события пользовательского интерфейса. Важно отметить что, несмотря на создающееся впечатление, данный пример не нарушает правила разделения ответственности, принятого в MVC! В ответ на вызов `setText` данной реализации `OnClickListener` объект `EditText` обновляет внутреннее представление текста, который следует отобразить, а затем вызывает собственный метод `invalidate`. Немедленной отрисовки на экране не происходит. Вообще, очень немногие правила в программировании являются абсолютными. Замечание о необходимости разделения модели, вида и контроллера — почти абсолютное правило.

В данном примере экземпляр класса `Button` подключается к его поведению при помощи обратного вызова, этот процесс подробно описан в подразделе «Переопределения и обратные вызовы» раздела «Шаблон приложения Android» главы 3. `Button` — это подкласс `View`, определяющий интерфейс под названием `OnClickListener` и метод, называющийся `setOnClickListener`, при помощи которого регистрируется слушатель. Интерфейс `OnClickListener` определяет единственный метод — `onClick`. Когда кнопка `Button` получает событие от фреймворка, то дополнительно ко всем другим задачам, которые ему приходится выполнить, кнопка проверяет событие и определяет, квалифицируется ли оно как щелчок. (Кнопка из первого примера при нажатии уже подсвечивается, даже пока к ней еще не добавлен слушатель.) Если событие квалифицируется как щелчок, а к кнопке добавлен слушатель щелчков, то активируется метод `onClick`, относящийся к слушателю.

Слушатель щелчков свободно может реализовывать любые необходимые пользовательские поведения. В частности, в предыдущем примере пользовательское поведение создает два случайных числа в диапазоне от 0 до 200 и ставит по одному числу в каждое из двух текстовых полей. Можно и не делать подкласс от `Button` и не переопределять его методы обработки событий. Все, что требуется для расширения существующего поведения, — зарегистрировать слушатель щелчков, реализующий желаемое поведение. Конечно же, так гораздо проще!

Обработчик щелчков особенно интересен еще и тем, что в самом сердце системы Android — в очереди событий фреймворка — нет такого феномена, как событие щелчка! Вместо этого при обработке событий `View` концепция щелчка синтезируется из других событий. Если устройство имеет сенсорный экран, то, например, легкий удар пальцем по экрану воспринимается как щелчок. Если на устройстве есть центральная клавиша на игровой крестовине либо клавиша `Enter`, то при нажатии и отпускании каждой из этих клавиш также будет регистрироваться щелчок. Клиентам вида неважно, какое именно событие воспринимается как щелчок и как оно генерируется на конкретном устройстве. Клиенты должны обрабатывать только событие в общем виде, а деталями выполнения следует заниматься фреймворку.

У `View` может быть только один `OnClickListener`. При повторном вызове метода `setOnClickListener` к тому или иному `View` старый слушатель будет удален, а на его место поставлен новый. С другой стороны, отдельный слушатель может принимать сигналы не только от одного `View`. Так, код из примера 6.5 — это часть другого приложения, которое выглядит точно так же, как программа из примера 6.2. Но в данном случае содержимое текстового поля будет обновляться при нажатии *любой из двух* кнопок.

Такая возможность может быть очень удобна в приложении, где несколько действий вызывают одинаковое поведение. Но не поддавайтесь соблазну создать единственный, колоссальный слушатель для всех ваших виджетов! Если сделать в коде множество мелких виджетов, каждый из которых реализует одиночное, четкое поведение, то такой код будет гораздо проще поддерживать и модифицировать.

Пример 6.5. Слушание нескольких кнопок

```
@Override public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);

    final EditText tb1 = (EditText) findViewById(R.id.text1);
    final EditText tb2 = (EditText) findViewById(R.id.text2);

    Button.OnClickListener listener = new Button.OnClickListener() {
        @Override public void onClick(View arg0) {
            tb1.setText(String.valueOf(rand.nextInt(200)));
            tb2.setText(String.valueOf(rand.nextInt(200)));
        }
    };

    ((Button) findViewById(R.id.button1)).setOnClickListener(listener);
    ((Button) findViewById(R.id.button2)).setOnClickListener(listener);
}
```

Слушание модели

Во фреймворке пользовательского интерфейса Android повсеместно применяется паттерн установки обработчиков. Хотя в предыдущих примерах мы работали только с видами `Button`, многие другие виджеты Android также определяют слушатели. Класс `View` определяет несколько повсеместно используемых событий и слушателей, скоро мы подробно рассмотрим их. Другие классы при этом определяют иные, специализированные типы событий и предоставляют обработчики для этих событий. Такие обработчики имеют значение только в рамках указанных классов. Это стандартная идиома, позволяющая клиентам специально настраивать (кастомизировать) поведение виджета, не образуя от него подклассов.

Такой паттерн также отлично подходит для того, чтобы программа могла обрабатывать собственные внешние, асинхронные действия. Если вы, например, отвечаете на изменение состояния, произошедшее на удаленном сервере, или обновляете информацию, которая получена от службы определения местоположения, то приложение может определить на эти случаи собственные события и обработчики, чтобы клиенты могли реагировать на такие события.

До сих пор примеры были элементарными и обходили некоторые острые углы. В этих примерах продемонстрировано соединение вида и контроллера, но в них не было настоящих моделей. (В частности, в примере 6.4 использовалась строка `String`, принимавшаяся реализацией `EditText` в качестве модели.) Далее мы сделаем небольшое отступление и построим настоящую модель, которой можно будет пользоваться.

Два следующих класса, показанных в примере 6.6, составляют модель, которая будет поддерживать расширения для демонстрационного приложения. Эти классы обеспечивают возможность хранения списка объектов, каждый из которых имеет координаты *x* и *y*, обладает цветом и размером. Кроме того, они предоставляют способ для регистрации слушателя и интерфейс, реализовывать который должен слушатель. В основе этих примеров лежит обычная модель слушателя, поэтому примеры достаточно просты.

Пример 6.6. Модель Dots

```
package com.oreilly.android.intro.model;

/** Точка: координаты, цвет и размер. */
public final class Dot {
    private final float x, y;
    private final int color;
    private final int diameter;

    /**
     * @param x координата по горизонтали
     * @param y координата по вертикали
     * @param color цвет
     * @param diameter диаметр точки
     */
    public Dot(float x, float y, int color, int diameter) {
        this.x = x;
        this.y = y;
        this.color = color;
        this.diameter = diameter;
    }

    /** @return координаты по горизонтали */
    public float getX() { return x; }

    /** @return координаты по вертикали. */
    public float getY() { return y; }

    /** @return цвета */
    public int getColor() { return color; }

    /** @return диаметра точки. */
    public int getDiameter() { return diameter; }
}

package com.oreilly.android.intro.model;

import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

/** список точек */
public class Dots {
```

```

/** слушатель DotChangeListener */
public interface DotsChangeListener {
    /** @param dots точек, которые изменились */
    void onDotsChange(Dots dots);
}

private final LinkedList<Dot> dots = new LinkedList<Dot>();
private final List<Dot> safeDots = Collections.unmodifiableList(dots);

private DotsChangeListener dotsChangeListener;

/** @param l нового слушателя изменений */
public void setDotsChangeListener(DotsChangeListener l) {
    dotsChangeListener = l;
}

/** @return возврат последней добавленной точки или возврат null */
public Dot getLastDot() {
    return (dots.size() <= 0) ? null : dots.getLast();
}

/** @return возврат списка точек */
public List<Dot> getDots() { return safeDots; }

/**
 * @param x координата по горизонтали
 * @param y координата по вертикали
 * @param color цвет
 * @param diameter размер точки.
 */
public void addDot(float x, float y, int color, int diameter) {
    dots.add(new Dot(x, y, color, diameter));
    notifyListener();
}

/** удаление всех точек */
public void clearDots() {
    dots.clear();
    notifyListener();
}

private void notifyListener() {
    if (null != dotsChangeListener) {
        dotsChangeListener.onDotsChange(this);
    }
}
}

```

Дополнительно к этой модели изучим и следующий пример, в котором будет показан библиотечный виджет DotView, используемый для просмотра данной модели. Его задача — отрисовать точки, предоставленные моделью. Они должны иметь

нужный цвет и правильно располагаться. Полный исходный код для данного примера приведен на сайте книги.

В примере 6.7 показано новое демонстрационное приложение, к которому теперь добавлены модель и вид.

Пример 6.7. Демонстрационная версия Dots

```
package com.oreilly.android.intro;
```

```
import java.util.Random;
```

```
import android.app.Activity;
import android.graphics.Color;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.LinearLayout;
```

```
import com.oreilly.android.intro.model.Dot;
import com.oreilly.android.intro.model.Dots;
import com.oreilly.android.intro.view.DotView;
```

```
/** Демонстрационная версия пользовательского интерфейса Android. */
```

```
public class TouchMe extends Activity {
    public static final int DOT_DIAMETER = 6;
```

```
    private final Random rand = new Random();
```

```
    final Dots dotModel = new Dots();
```

```
    DotView dotView;
```

```
    /** Вызывается при первом создании активности. */
```

```
    @Override public void onCreate(Bundle state) {
        super.onCreate(state);
```

```
        dotView = new DotView(this, dotModel);
```

```
        // установка вида View
```

```
        setContentView(R.layout.main);
```

```
        ((LinearLayout) findViewById(R.id.root)).addView(dotView, 0); 1
```

```
        // подключение контроллера Controller
```

```
        ((Button) findViewById(R.id.button1)).setOnClickListener(  
            new Button.OnClickListener() { 2
```

```
                @Override public void onClick(View v) {  
                    makeDot(dots, dotView, Color.RED); 3
```

```
                } });
```

```
        ((Button) findViewById(R.id.button2)).setOnClickListener(  
            new Button.OnClickListener() { 2
```

```

        @Override public void onClick(View v) {
            makeDot(dots, dotView, Color.GREEN); 3
        }
    });
    final EditText tb1 = (EditText) findViewById(R.id.text1);
    final EditText tb2 = (EditText) findViewById(R.id.text2);
    dots.setDotsChangeListener(new Dots.DotsChangeListener() { 4
        @Override public void onDotsChange(Dots d) {
            Dot d = dots.getLastDot();
            tb1.setText((null == d) ? „" : String.valueOf(d.getX()));
            tb2.setText((null == d) ? „" : String.valueOf(d.getY()));
            dotView.invalidate();
        }
    });
}

/**
 * @param dots точки, которые мы рисуем
 * @param view вид, в котором мы рисуем точки
 * @param color цвет точки
 */
void makeDot(Dots dots, DotView view, int color) { 5
    int pad = (DOT_DIAMETER + 2) * 2;
    dots.addDot(
        DOT_DIAMETER + (rand.nextFloat() * (view.getWidth() - pad)),
        DOT_DIAMETER + (rand.nextFloat() * (view.getHeight() - pad)),
        color,
        DOT_DIAMETER);
    }
}

```

Вот комментарии к фрагментам кода, отмеченным цифрами.

- 1** Новый DotView добавляется поверх макета, полученного из XML-определения.
- 2** Обратные вызовы `onClick` добавляются к кнопкам Red (Красный) и Green (Зеленый). Эти обработчики событий отличаются от обработчиков, приведенных в предыдущем примере, только тем, что здесь их поведение опосредовано локальным методом `makeDot`. Этот новый метод создает точку (элемент 5).
- 3** Вызов `makeDot` осуществляется в `onClick` (тогда вызов происходит при нажатии кнопки).
- 4** Самое существенное изменение в примере происходит там, где модель подключается к виду. При этом для установки `dotsChangeListener` используется обратный вызов. Когда модель изменяется, вызывается данный новый слушатель. Он записывает координаты *x* и *y* последней точки соответственно в левое и правое текстовые поля и посылает запрос DotView, чтобы этот вид перерисовал себя (вызов `invalidate`).
- 5** Это определение `makeDot`. Данный новый метод создает точку, убеждается, что она находится в пределах границ DotView, и добавляет ее к модели. Кроме того, в данном случае цвет точки можно указать в качестве параметра.

На рис. 6.5 показано, как выглядит работающее приложение.

Если нажать кнопку **Red** (Красный), то в **DotView** появится новая красная точка. Если нажать **Green** (Зеленый), возникнет зеленая. Текстовые поля содержат координаты последней добавленной точки.

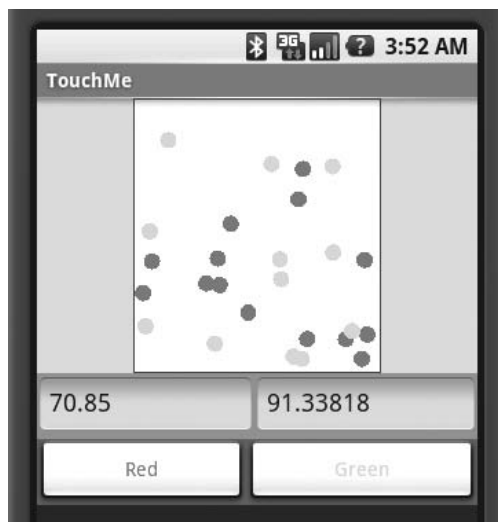


Рис. 6.5. Запуск демонстрационного приложения Dots

Базовая структура примера 6.2 по-прежнему узнаваема, но мы видим и некоторые расширения.

Вот цепь событий, которые происходят после нажатия, например, кнопки **Green** (Зеленый).

1. При нажатии кнопки вызывается ее `onClickHandler`.
2. Из-за этого происходит вызов `makeDot` с цветовым аргументом `Color.GREEN`. Метод `makeDot` генерирует случайные координаты и добавляет новую зеленую точку (`Dot`) в модель в этих координатах.
3. Когда модель обновится, она вызывает `onDotsChangeListener`.
4. Слушатель обновляет значения текстовых видов и запрашивает перерисовку `DotView`.

Слушание событий касания

Как вы уже, конечно же, догадались, чтобы модифицировать демонстрационное приложение для реагирования на прикосновения к экрану, нужно просто добавить к коду обработчик касаний. Код из примера 6.8 дополняет приложение так, чтобы в `DotView`, там, где происходит прикосновение к экрану, появлялась голубая точка. Этот код нужно добавить в демонстрационное приложение (см. пример 6.7) в начале функции `onCreate`, прямо после вызова к ее родительскому методу. Обратите внимание: поскольку код, отображающий координаты `x` и `y` последней добавленной

точки, подключен только к модели, он продолжает работать правильно, независимо от того, как именно вид добавляет точку.

Пример 6.8. Точки от прикосновений

```
dotView.setOnTouchListener(new View.OnTouchListener() {
    @Override public boolean onTouch(View v, MotionEvent event) {
        if (MotionEvent.ACTION_DOWN != event.getAction()) {
            return false;
        }
        dots.addDot(event.getX(), event.getY(), Color.CYAN, DOT_DIAMETER);
        return true;
    }
});
```

Событие `MotionEvent`, переданное обработчику, имеет несколько других свойств, кроме информации о месте того касания, которое вызвало появление этой точки. Из примера видно, что в нем также содержится информация о типе события. Таких типов четыре: `DOWN`, `UP`, `MOVE` или `CANCEL`. Обычное касание генерирует одно событие `DOWN` и одно событие `UP`. Прикосновение и перетаскивание генерирует событие `DOWN`, несколько событий `MOVE` и, наконец, событие `UP`.

Возможности обработки жестов, обеспечиваемые `MotionEvent`, очень интересны. Событие содержит данные о размере области, которой коснулся пользователь, а также о силе нажатия. Это означает, что на устройствах, поддерживающих такие функции, приложение сможет отличать нажатие одним пальцем от нажатия двумя пальцами, либо очень легкое прикосновение от сильного щелчка.

В мобильном программировании очень важна эффективность кода. Фреймворк пользовательского интерфейса при отслеживании событий, связанных с сенсорным экраном, и при сообщении о них сталкивается с определенной дилеммой. Если сообщать о слишком малом количестве событий, то, возможно, системе не удастся проследить движение с достаточной точностью — а, например, при распознавании почерка нужна высокая точность отслеживания движений. С другой стороны, сообщая о большом количестве актов касания, оформляя каждый такой акт как отдельное событие, можно обременить систему неприемлемо высокой нагрузкой. Во фреймворке пользовательского интерфейса Android эта проблема решается путем объединения групп образцов в пакеты. Так удастся снизить нагрузку на систему, сохраняя при этом точность ее работы. Чтобы просмотреть все образцы, ассоциированные с событием, используется функция отслеживания истории событий (*history facility*), реализуемая при помощи методов `getHistoricalX`, `getHistoricalY` и т. д.

В примере 6.9 показано, как пользоваться этой функцией. Код дополняет демонстрационную программу таким образом, чтобы можно было отслеживать жесты пользователя, который касается сенсорного экрана. Фреймворк передает образцы координат *x* и *y* методу `onTouch` объекта, установленного как `OnTouchListener` для `DotView`. Во всех полученных координатах этот метод отображает голубую точку.

Пример 6.9. Отслеживание движений

```
private static final class TrackingTouchListener
    implements View.OnTouchListener
{
```

```

private final Dots mDots;

TrackingTouchListener(Dots dots) { mDots = dots; }

@Override public boolean onTouch(View v, MotionEvent evt) {
    switch (evt.getAction()) {
        case MotionEvent.ACTION_DOWN:
            break;

        case MotionEvent.ACTION_MOVE:
            for (int i = 0, n = evt.getHistorySize(); i < n; i++) {
                addDot(
                    mDots,
                    evt.getHistoricalX(i),
                    evt.getHistoricalY(i),
                    evt.getHistoricalPressure(i),
                    evt.getHistoricalSize(i));
            }
            break;

        default:
            return false;
    }

    addDot(
        mDots,
        evt.getX(),
        evt.getY(),
        evt.getPressure(),
        evt.getSize());

    return true;
}

private void addDot(Dots dots, float x, float y, float p, float s) {
    dots.addDot(
        x,
        y,
        Color.CYAN,
        (int) ((p * s * Dot.DIAMETER) + 1));
}
}

```

На рис. 6.6 показано, как могла бы выглядеть дополненная версия приложения после нескольких щелчков и перетаскиваний.

В данной реализации для определения диаметра точки, которую следует отобразить, используются показатели размера той площади, к которой приложено давление, и силы этого давления. К сожалению, эмулятор Android не имитирует площади и силы нажатия, поэтому в нем все точки имеют одинаковый диаметр. Значения площади и силы нажатия стандартизированы на различных устройствах и имеют вид чисел с плавающей точкой в диапазоне от 0,0 до 1,0. При этом в за-

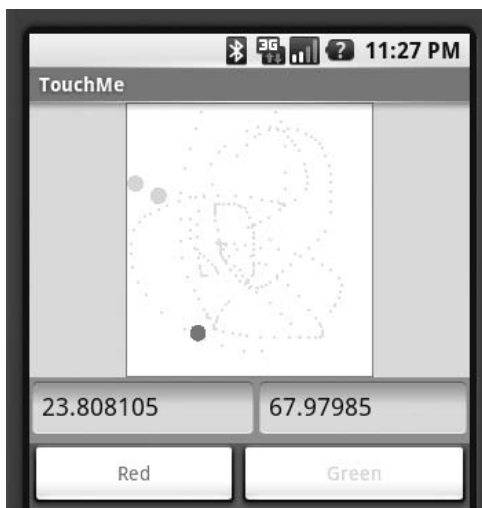


Рис. 6.6. Результат, достигаемый после продолжительной работы демонстрационной программы Dots

висимости от калибровки экрана возможны значения, превышающие 1,0. Эмулятор всегда воспринимает силу и площадь нажатия как равные нулю, то есть имеющие минимальное значение.

Цикл обработки событий ACTION_MOVE работает с событиями из зафиксированной истории, собранными в пакеты. Если касания сменяются быстрее, чем фреймворк может об этом сообщать, то он объединяет их в единое событие. Метод `getHistorySize`, относящийся к `MotionEvent`, возвращает количество образцов в пакете, а различные методы `getHistory` дают информацию о специфике подсобытий.

Устройства с трекболами (шариковыми манипуляторами) также генерируют события `MotionEvent` при перемещении трекбола. Эти события подобны тем, что возникают при касаниях сенсорного экрана, но обработка их происходит иначе. События трекбола `MotionEvent` передаются в вид `View` при вызове метода `dispatchTrackballEvent`, а не `dispatchTouchEvent`, применяемого для обработки касаний. Притом что `dispatchTrackballEvent` передает событие методу `onTrackballEvent`, он перед этим не передает данное событие слушателю! Генерируемые трекболом события `MotionEvent` не только невидимы для обычного механизма обработки касаний, более того — для отклика на них виджет должен создать подкласс от `View` и переопределить метод `onTrackballEvent`.

События `MotionEvent`, генерируемые трекболом, доступны благодаря механизму обратных вызовов. Если эти события не *потребляются* (об этом процессе — чуть ниже), то превращаются в события крестовины. Это вполне объяснимо, ведь на большинстве устройств есть или трекбол, или крестовина, а вот одновременно эти элементы почти не встречаются. Без такого преобразования было бы невозможно генерировать события крестовины на устройстве, где есть только трекбол. Конечно, все это подразумевает, что приложение, обрабатывающее события трекбола, должно делать это настолько аккуратно, чтобы не нарушить преобразования.

После преобразования движение трекбола воспринимается приложением как серия нажатий клавиши крестовины.

Множественные указатели и жесты

Многие устройства позволяют одновременно отслеживать более одного указателя. Эта технология иногда называется мультитач. Если пользователь касается сенсорного экрана в нескольких местах сразу, след от каждого такого контакта обрабатывается отдельно. Такие отдельные следы могут использоваться для различения сложных жестов с конкретными значениями. К числу подобных жестов относится прокрутка (scroll), масштабирование (zoom), листание (next page) и т. д.

Все методы событий, рассмотренные выше и возвращающие информацию о событии MOVE, — `getX`, `getY`, `getHistoricalX`, `getHistoricalY` и т. д. — поддерживают мультитач. Для этого они принимают дополнительный аргумент, указывающий конкретный след, к которому относится вызов. Например, кроме `getX()`, существует метод `getX(int pointerIndex)`. Индекс указателя позволяет вызывающей стороне получать доступ к нескольким отдельным следам, а метод `getPointerCount` возвращает количество отдельных следов, зафиксированных при событии. К сожалению, индекс конкретного события в рамках события может изменяться. Иными словами, если пользователь касается экрана большим и указательным пальцем, то след от большого пальца может быть выражен в виде последовательных событий движения сначала с индексом 0, потом с индексом 1, а затем снова с индексом 0. Чтобы проследить один след на протяжении нескольких событий, нужно использовать идентификатор (ID) следа, а не индекс. Для этого воспользуйтесь методами `getPointerId` и `findPointerIndex` — они применяются для преобразования ID в индекс и наоборот.

Код из примера 6.10 дополняет метод `onTouch` из предыдущего примера так, чтобы можно было одновременно отслеживать несколько следов.

Пример 6.10. Отслеживание движения

```
@Override public boolean onTouch(View v, MotionEvent evt) {
    int n;
    int idx;
    int action = evt.getAction();
    switch (action & MotionEvent.ACTION_MASK) {
        case MotionEvent.ACTION_DOWN:
        case MotionEvent.ACTION_POINTER_DOWN:
            idx = (action & MotionEvent.ACTION_POINTER_INDEX_MASK)
                >> MotionEvent.ACTION_POINTER_INDEX_SHIFT;
            tracks.add(Integer.valueOf(evt.getPointerId(idx)));
            break;

        case MotionEvent.ACTION_POINTER_UP:
            idx = (action & MotionEvent.ACTION_POINTER_INDEX_MASK)
                >> MotionEvent.ACTION_POINTER_INDEX_SHIFT;
            tracks.remove(Integer.valueOf(evt.getPointerId(idx)));
            break;
    }
}
```

```

        case MotionEvent.ACTION_MOVE:
            n = evt.getHistorySize();
            for (Integer i: tracks) {
                idx = evt.findPointerIndex(i.intValue());
                for (int j = 0; j < n; j++) {
                    addDot(
                        mDots,
                        evt.getHistoricalX(idx, j),
                        evt.getHistoricalY(idx, j),
                        evt.getHistoricalPressure(idx, j),
                        evt.getHistoricalSize(idx, j));
                }
            }
            break;

        default:

            return false;
    }

    for (Integer i: tracks) {
        idx = evt.findPointerIndex(i.intValue());
        addDot(
            mDots,
            evt.getX(idx),
            evt.getY(idx),
            evt.getPressure(idx),
            evt.getSize(idx));
    }

    return true;
}

```

В этом коде следует обсудить несколько моментов. Обратите внимание на то, что переключение оператора выбора теперь происходит не на действии события, а на замаскированной версии этого действия. Такой вариант оперирования отдельными битами (bit banging) кажется несколько архаичным. Необходимо поддерживать обратную совместимость вызовов. Чтобы игнорировать множественные следы, не выполняйте маскирование и обязательно добавляйте к оператору переключения вариант default.

Далее обратите внимание на два новых варианта в операторе переключения: `MotionEvent.ACTION_POINTER_DOWN` и `MotionEvent.ACTION_POINTER_UP`. В этом простом примере они используются для индикации начала и конца нового следа. Поскольку непрерывный след идентифицируется по его ID, код просто добавляет еще один ID с началом нового следа и удаляет этот ID, когда след заканчивается.

Наконец, отметим, что в событии хранится информация об истории: для всех следов сохраняется одинаковое количество таких записей.

В очень специфическом, но при этом распространенном случае несколько следов могут складываться в жест, имеющий конкретное значение (например, щипок с последующим масштабированием — увеличением или уменьшением). Библиотеки

Android указывают, что поддержка таких операций хорошо реализуется при помощи детекторов жестов. В официальной документации прямо говорится о том, что два предоставляемых сразу детектора жестов задуманы скорее как пример для будущих реализаций, чем как законченное решение. Действительно, два имеющихся детектора, `GestureDetector` и `ScaleGestureDetector`, «навскидку» поддерживают всего несколько распространенных жестов — одиночное касание, двойное касание, длинное касание и щипок.

В принципе, для использования детектора жестов требуется создать экземпляр, зарегистрировать слушатель (приемник), добавить детектор к обработчику `onTouch` и передавать ему каждое новое событие. Подробности функционирования детекторов — в частности, необходимые типы приемников — специфичны для каждого конкретного детектора.

Слушание событий клавиатуры

Обработка ввода с клавиатуры, пригодная для использования на нескольких платформах, реализуется непросто. На некоторых устройствах гораздо больше кнопок, чем на других, некоторые требуют тройного щелчка для символического ввода и т. д. Тройной щелчок — отличный пример функционала, который нужно по возможности реализовывать во фреймворке, в классе `EditText` или каком-либо его подклассе.

Чтобы дополнить функционал виджета по обработке событий клавиатуры `KeyEvent`, используется метод `setOnKeyListener`. Он относится к виду `View` и устанавливает метод `OnKeyListener`. Слушателю будет приходить несколько событий `KeyEvent` при каждом пользовательском нажатии клавиши, по одному событию для каждого из трех типов действия: `DOWN`, `UP` и `MULTIPLE`. Типы действия `DOWN` и `UP` указывают, что клавиша была нажата или отпущена, как и в случае с классом `MotionEvent`. Действие типа `MULTIPLE`, связанное с клавишами, говорит, что клавиша удерживается (автоматический повтор ввода). Метод `getRepeatCount`, относящийся к событиям `KeyEvent`, указывает, какое количество нажатий представлено событием типа `MULTIPLE`.

В примере 6.11 показан образец обработчика нажатий клавиш. Если добавить этот фрагмент к демонстрационной программе, он вызывает появление на экране точек в случайных координатах. Это происходит при нажатии и отпускании клавиш. Розовая точка добавляется при нажатии и отпускании клавиши пробела, желтая точка — при нажатии и отпускании клавиши «Ввод», голубая точка — при нажатии и отпускании любой другой клавиши. Хотя этот обработчик нажатий и работает именно так, он является упрощенным! В нем есть недоработки, о которых мы поговорим чуть ниже.

Пример 6.11. Обработка нажатий клавиш

```
dotView.setFocusable(true);
```

```
dotView.setOnKeyListener(new OnKeyListener() {  
    @Override public boolean onKey(View v, int keyCode, KeyEvent event) {  
        if (KeyEvent.ACTION_UP != event.getAction()) {  
            int color = Color.BLUE;
```

```
        switch (keyCode) {
            case KeyEvent.KEYCODE_SPACE:
                color = Color.MAGENTA;
                break;
            case KeyEvent.KEYCODE_ENTER:
                color = Color.YELLOW;
                break;
            default: ;
        }

        makeDot(dots, dotView, color);
    }

    return true;
} });
```

Выбор обработчика событий

Вы, вероятно, заметили, что рассмотренные до сих пор методы типа `on...`, относящиеся к слушателям, — в том числе метод `onKey` — возвращают булево значение (`boolean`). Такой принцип работы слушателей позволяет им контролировать последующую обработку события, производимую вызывающей стороной.

Когда событие контроллера передается виджету, код фреймворка в виджете направляет это событие в зависимости от типа события к подходящему методу: `onKeyDown`, `onTouchEvent` и т. д. Эти методы, находящиеся либо в классе `View`, либо в его подклассах, реализуют поведение виджета. Но, как было описано выше, сначала фреймворк предлагает событие подходящему слушателю (`onTouchListener`, `onKeyListener` и т. д.), если таковой существует. Значение, возвращаемое слушателем, определяет, будет ли событие затем направлено к методам класса `View`.

Если слушатель возвращает `false`, то событие направляется к методам `View`, так как обработчик не существует. Если же, напротив, слушатель возвращает `true`, то принято говорить, что событие *потребляется* (`consumed`). Класс `View` отменяет всяческую его дальнейшую обработку. Методы `View` никогда не вызываются применительно к потребленному событию, не могут его обрабатывать или отвечать на него. С точки зрения методов `View` эта ситуация такова, как если бы события не существовало.

Итак, есть три сценария, по которым может развиваться обработка события.

- Слушатель отсутствует — событие направляется к методам класса `View` для нормальной обработки. Реализация виджета, разумеется, может переопределять эти методы.
- Слушатель существует и возвращает `true` — обработка события слушателем полностью заменяет обычную обработку события виджетом. Событие никогда не направляется к `View`.
- Слушатель существует и возвращает `false` — событие обрабатывается сначала слушателем, а затем — классом `View`. После того как слушатель завершит обработку события, событие передается к `View` для обычной обработки.

Рассмотрим, например, что бы произошло, если бы слушатель нажатий на клавиши из примера 6.11 был добавлен к виджету `EditText`. Поскольку метод `onKey` всегда возвращает `true`, фреймворк отменит любую дальнейшую обработку события `KeyEvent` сразу после того, как метод вернет значение. Поэтому механизм обработки нажатий клавиш из `EditText` даже не будет видеть нажатий клавиши и в текстовых полях так и не появится никакого текста. Очевидно, такое поведение нас не устраивает!

Если же, напротив, метод `onKey` вернет `false` для некоторых нажатий клавиш, то фреймворк диспетчирует эти события виджету `EditText` для дальнейшей обработки. Механизм `EditText` увидит эти события, и связанные с ними символы будут добавлены в поле `EditText`, как и ожидается. В примере 6.12 показан дополненный код из примера 6.11. Теперь мы не только добавляем новые точки в модель, но и фильтруем символы, добавляемые в поле `EditText`. Но это решение по-прежнему неполное, так как оно не фиксирует нажатия некоторых важных клавиш, например клавиши **Menu** (Меню). Таким образом, мы просто добиваемся нормальной обработки числовых символов.

Пример 6.12. Дополненная обработка нажатий на клавиши

```
new OnKeyListener() {
    @Override public boolean onKey(View v, int keyCode, KeyEvent event) {
        if (KeyEvent.ACTION_UP != event.getAction()) {
            int color = Color.BLUE;
            switch (keyCode) {
                case KeyEvent.KEYCODE_SPACE:
                    color = Color.MAGENTA;
                    break;
                case KeyEvent.KEYCODE_ENTER:
                    color = Color.YELLOW;
                    break;
                default: ;
            }

            makeDot(dotModel, dotView, color);
        }

        return (keyCode < KeyEvent.KEYCODE_0)
            || (keyCode > KeyEvent.KEYCODE_9);
    }
}
```

Если в вашем приложении требуется реализовать совершенно новые способы обработки событий — которым недостаточно простого дополнения поведений и фильтрации при помощи `onKeyHandler`, — то придется разобраться в механизме обработки нажатий клавиш, применяемом в классе `View`, и переопределить этот механизм. В общем виде процесс выглядит так: события направляются к классу `View` посредством метода `DispatchKeyEvent`. Метод `DispatchKeyEvent` реализует поведение, описанное выше, предлагая событие сначала методу `onKeyHandler`, а потом, если обработчик вернет `false`, — методам класса `View`, реализующим интерфейс `KeyEvent.Callback`. Это методы `onKeyDown`, `onKeyUp` и `onKeyMultiple`.

Продвинутые способы подключения: фокус и поточность

Как было показано в примере 6.7 и в разделе «Слушание событий касания», события `MotionEvent` направляются к тому виджету, к рабочему прямоугольнику которого относится точка координат, где произошло касание, сгенерировавшее данное событие. Не так просто определить, какой именно виджет должен получать событие `KeyEvent`. Чтобы это делать, фреймворк пользовательского интерфейса Android, как и другие подобные фреймворки, поддерживает концепцию «выделенной области» (*selection*), которая также называется термином «фокус».

Чтобы виджет мог попасть в фокус, его атрибут `focusable` должен иметь значение `true`. Этого можно добиться либо при помощи атрибутов макета (*layout attributes*) в XML (у видов `EditText` в примере 6.3 атрибут `focusable` имеет значение `false`), либо посредством метода `setFocusable`, как показано в первой строке кода в примере 6.11. Пользователь перемещает фокус с одного объекта `View` на другой, работая с клавишами крестовины или нажимая на экран — если он сенсорный.

Когда виджет находится в фокусе, он обычно отображается с определенной подсветкой, сигнализируя, что именно к нему применяются действия клавиш, нажимаемых в данный момент. Например, когда виджет `EditText` находится в фокусе, он подсвечивается и, кроме того, в точке, с которой начинается ввод текста, ставится курсор.

Чтобы получать уведомления, когда вид попадает в фокус или выходит из него, нужно установить `OnFocusChangeListener`. В примере 6.13 показан слушатель, при помощи которого в демонстрационную программу можно добавить функционал, связанный с фокусом. Этот код автоматически добавляет в `DotView` случайно расположенные точки через случайные интервалы, когда данный вид оказывается в фокусе.

Пример 6.13. Работа с фокусом

```
dotView.setOnFocusChangeListener(new OnFocusChangeListener() {
    @Override public void onFocusChange(View v, boolean hasFocus) {
        if (!hasFocus && (null != dotGenerator)) {
            dotGenerator.done();
            dotGenerator = null;
        }
        else if (hasFocus && (null == dotGenerator)) {
            dotGenerator = new DotGenerator(dots, dotView, Color.BLACK);
            new Thread(dotGenerator).start();
        }
    }
});
```

В `OnFocusChangeListener` нет ничего необычного. Когда виджет `DotView` попадает в фокус, он создает `DotGenerator` и порождает поток для его запуска. Когда виджет выходит из фокуса, `DotGenerator` останавливается и освобождается. Новое поле данных `dotGenerator` (его объявление в примере не показано) является ненулевым лишь тогда, когда `DotView` располагается в фокусе. Есть еще один важный и мощный инструмент, применяемый при реализации `DotGenerator`, чуть ниже мы его рассмотрим.

Чтобы поместить тот или иной виджет в фокус, нужно вызвать один из методов его класса `View` — `requestFocus`. Когда `requestFocus` вызывается применительно к новому виджету, запрос отправляется вверх по дереву, от одного родительского элемента к другому, вышестоящему родительскому элементу, до самого корня дерева. Корень запоминает, какой виджет находится в фокусе, и передает последующие события, связанные с нажатиями клавиш, непосредственно ему.

Именно так фреймворк пользовательского интерфейса переводит фокус на новый виджет в ответ на нажатия клавиш крестовины. Фреймворк идентифицирует виджет, который должен оказаться в фокусе следующим, и вызывает метод `requestFocus` этого виджета. В результате виджет, который находился в фокусе предыдущим, выходит из фокуса, а целевой виджет — попадает в фокус.

Процесс идентификации виджета, который должен попасть в фокус, довольно сложен. Для определения этого виджета навигационный алгоритм выполняет сложные вычисления, которые могут зависеть от того, как расположены на экране все без исключения остальные виджеты!

Рассмотрим, например, что произойдет, если нажата правая клавиша крестовины и фреймворк пытается перевести фокус на виджет, вплотную прилегающий справа к тому виджету, который находится в фокусе в настоящий момент. При взгляде на экран может быть совершенно очевидно, что это за виджет, но ситуация совсем не так ясна для дерева видов. Целевой виджет может находиться на другом уровне дерева и отстоять на несколько «ветвей» от виджета, который находится в фокусе сейчас. Идентификация такого виджета зависит от точных параметров виджетов, располагающихся совсем в других, отдаленных частях дерева. К счастью, несмотря на немалую сложность, реализация этого механизма во фреймворке пользовательского интерфейса Android обычно работает так, как ожидается.

Если что-то идет не так, то можно воспользоваться четырьмя свойствами — они задаются либо методом приложения, либо **XML-атрибутом**, — которые принудительно вызывают желаемое навигационное поведение при переводе фокуса. Эти свойства — `nextFocusDown`, `nextFocusLeft`, `nextFocusRight` и `nextFocusUp`. Если задать одно из этих свойств со ссылкой на конкретный виджет, то можно гарантировать, что при нажатии клавиш крестовины в определенном направлении фокус будет переходить именно на тот виджет, на который указывает ссылка.

Еще одна сложность, связанная с механизмом фокуса, заключается в том, что пользовательский интерфейс Android различает фокус крестовины и фокус, возникающий в результате касаний сенсорного экрана (если такой экран есть на устройстве). Чтобы понять, зачем необходимо такое различие, напомним, что на экране, не допускающем сенсорного ввода, единственный способ нажать кнопку — навести на нее фокус при помощи крестовины, а потом сделать щелчок центральной клавишей крестовины. Но на экране, который принимает сенсорный ввод, вообще не приходится наводить фокус на кнопку. Чтобы нажать кнопку, достаточно легонько ударить по ней пальцем, независимо от того, какой именно виджет находится в фокусе в данный момент. Тем не менее даже на сенсорном экране сохраняется необходимость наводить фокус на виджет, который может воспринимать нажатия клавиш. Таков, например, виджет `EditText`. Его нужно однозначно определять как целевой элемент, к которому будут относиться по-

следующие события, связанные с клавишами. Чтобы правильно обрабатывать обе разновидности фокуса, нужно разобраться с тем, как класс `View` обрабатывает `FOCUSABLE_IN_TOUCH_MODE`, а также изучить методы `isFocusableInTouchMode` и `isInTouchMode`, относящиеся к `View`.

В многооконном приложении механизм фокусировки приобретает еще как минимум один нюанс. Окно может выйти из фокуса, не уведомляя о потере фокуса тот из своих виджетов, который до последнего момента как раз находился в фокусе. Если задуматься, то становится понятно, почему это целесообразно. Ведь если окно, вышедшее из фокуса, вновь окажется поверх всех остальных окон, то виджет, который был в фокусе на момент окончания работы с этим окном, вновь окажется в фокусе — и для этого не нужно выполнять никаких дополнительных действий.

Допустим, вам нужно добавить в справочник номер телефона вашего друга. И при этом вы сразу же переходите в приложение «Телефон», чтобы уточнить последние несколько цифр этого номера. Конечно, вам не понравится, если при перезапуске адресной книги нужно будет снова вручную наводить фокус на то поле `EditText`, куда вы только что вводили текст. Ожидается, что приложение должно быть именно в том состоянии, в котором вы его оставили.

С другой стороны, у такого поведения могут быть необычные побочные эффекты. В частности, при реализации функции автоматического добавления точек из примера 6.13 точки продолжают добавляться в вид `DotView`, даже если в настоящий момент он накрыт другим окном! Если фоновая задача должна выполняться только тогда, когда конкретный виджет находится на виду, то эту задачу нужно удалять при выходе виджета из фокуса, то есть когда `Window` оказывается вне фокуса и работа `Activity` приостанавливается или прекращается.

Реализация механизма фокусировки осуществляется в основном в классе `ViewGroup` при помощи методов вроде `requestFocus` и `requestChildFocus`. Если потребуется реализовать совершенно новый механизм фокусировки, нужно будет подробно изучить методы и правильно их переопределить.

В примере 6.14 мы отвлечемся от темы фокусировки и вернемся к реализации недавно добавленной функции автоматического добавления точек. Здесь представлена реализация `DotGenerator`.

Пример 6.14. Обработка потоков

```
private final class DotGenerator implements Runnable {
    final Dots dots;
    final DotView view;
    final int color;

    private final Handler hdlr = new Handler(); ❶
    private final Runnable makeDots = new Runnable() { ❷
        public void run() { makeDot(dots, view, color); }
    };

    private volatile boolean done;

    // Выполняется в главном потоке.
    DotGenerator(Dots dots, DotView view, int color) { ❸
```

```

        this.dots = dots;
        this.view = view;
        this.color = color;
    }

    // Выполняется в главном потоке.
    public void done() { done = true; }

    // Выполняется в другом потоке!
    public void run() {
        while (!done) {
            try { Thread.sleep(1000); }
            catch (InterruptedException e) { }
            hdlr.post(makeDots); ④
        }
    }
}

```

Вот пояснения к выделенным строкам кода.

- ① Создается объект `android.os.Handler`.
- ② Создается новый поток, который будет запускать `makeDot` в элементе 4.
- ③ В основном потоке запускается `DotGenerator`.
- ④ `makeDot` запускается обработчиком `Handler`, созданным в элементе 1.

Упрощенная реализация `DotGenerator` могла бы просто вызывать `makeDot` напрямую из метода `run`. Тем не менее такая операция небезопасна, поскольку `makeDot` не является потокобезопасным, так же как и `Dots` и `DotView`. Такую ситуацию будет сложно правильно организовать и еще сложнее — поддерживать. Фреймворк пользовательского интерфейса Android запрещает доступ к объекту `View` от нескольких потоков. При запуске упрощенной реализации работа приложения аварийно завершится, выдав исключение `RuntimeException` такого плана:

```

11-30 02:42:37.471: ERROR/AndroidRuntime(162):
android.view.ViewRoot$CalledFromWrongThreadException:
Only the original thread that created a view hierarchy can touch its views1.

```

С такой проблемой мы уже сталкивались в подразделе «Потоки в процессе Android» раздела «Параллелизм в Android» главы 3, когда создавали обработчик `Handler`. Чтобы обойти такое ограничение, `DotGenerator` создает объект `Handler` в своем конструкторе. Объект `Handler` ассоциирован с потоком, в котором он был создан, и этот поток получает безопасный конкурентный доступ к базовой очереди событий.

Поскольку `DotGenerator` создает `Handler` в ходе собственного процесса конструкции, `Handler` ассоциируется с основным потоком. Теперь `DotGenerator` может использовать `Handler` для постановки в очередь объекта `Runnable`, созданного в другом потоке, причем такой объект вызывает `makeDot` из потока пользовательского интерфейса. Оказывается, как вы уже и сами догадались, что базовая очередь событий,

¹ Перевод данной строки: «Только тот поток, который создал иерархию виджетов, имеет доступ к этим видам». — *Примеч. пер.*

на которую указывает `Handler`, — та самая, с которой работает фреймворк пользовательского интерфейса. Вызов `makeDot` удаляется из очереди и обрабатывается, как и любое другое событие пользовательского интерфейса, в обычном порядке. В данном случае в результате запускается `Runnable`, относящийся к `makeDot`. `makeDot` вызывается из основного потока, и пользовательский интерфейс остается однопоточным.

Стоит повторить, что это — важный паттерн написания основного кода для пользовательского интерфейса Android. Если обработка действий, запущенная по инициативе пользователя, продлится более нескольких миллисекунд, то при выполнении этого действия в основном потоке можно замедлить работу всего пользовательского интерфейса или, что еще хуже, надолго его «подвесить». Если основной поток приложения не обслужит свою очередь событий за пару секунд, операционная система Android принудительно завершит приложение, так как оно не отвечает. Классы `Handler` и `AsyncTask` позволяют программисту избегать таких опасных ситуаций, делегируя медленные или подолгу выполняемые задачи другим потокам так, чтобы основной поток мог продолжать обслуживать пользовательский интерфейс. В этом примере демонстрируется использование потока `Thread` с обработчиком `Handler`, который периодически ставит в очередь события обновления пользовательского интерфейса.

Демонстрационное приложение здесь немного упрощено. Оно ставит операции создания новой точки и добавления ее к модели в очередь основного потока. Более сложное приложение могло бы при создании модели передавать ей `Handler` от основного потока, а пользовательскому интерфейсу предоставить возможность получить `Handler`, созданный в потоке модели. Модель, работающая в собственном потоке, использовала бы класс `Looper` для удаления из очереди и диспетчирования сообщений, поступающих от пользовательского интерфейса. Но прежде, чем приступать к построению чего-то настолько сложного, попробуйте использовать `Service` или `ContentProvider` (см. главу 13).

При передаче событий между пользовательским интерфейсом и долговременно исполняемым потоком мы, таким образом, значительно снижаем количество ограничений, которые требуется соблюдать для обеспечения безопасности потоков. В частности, в подразделе «Потоки в процессе Android» раздела «Параллелизм в Android» главы 3 мы упоминали, что поток, ставящий события в очередь, не сохраняет ссылок на объект, поставленный в очередь, или, если такой объект является неизменяемым, не требуется никакой дополнительной синхронизации.

Меню и панель действий

Последний элемент управления приложением, который мы рассмотрим в этой главе, — это меню. В примере 6.15 показано, как реализовать простое меню, переопределив два метода базового класса `Activity`.

Пример 6.15. Реализация меню

```
@Override public boolean onCreateOptionsMenu(Menu menu) {  
    getMenuInflater().inflate(R.menu.simple_menu, menu);  
}
```

```
        return true;
    }

    @Override public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.menu_clear:
                dotModel.clearDots();
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
}
```

Если добавить этот код в класс TouchMe, то при нажатии кнопки MENU (Меню) на устройстве приложение отобразит меню (рис. 6.7).

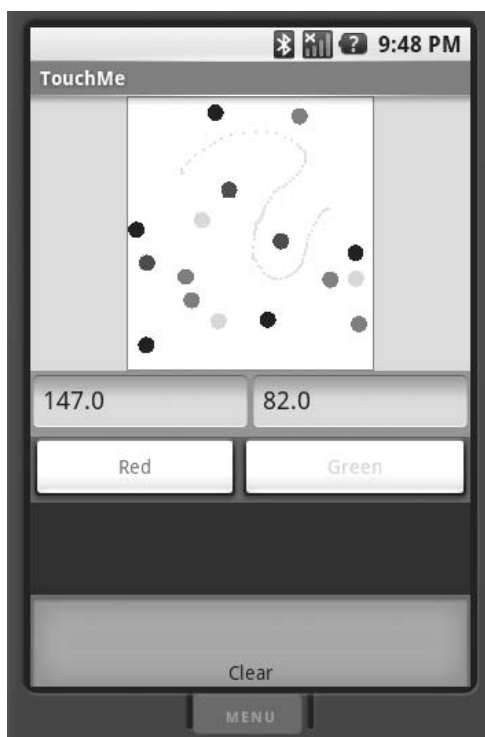


Рис. 6.7. Простое меню

Нажав «Ввод» или снова коснувшись экранной клавиши ввода пальцем, мы очистим весь вид с точками.

Уже в версии Honeycomb (Android 3.0) Google отказался от использования специальной кнопки «Меню» (она считается устаревшей) в пользу *панели действий* (action bar), которая будет подробно рассмотрена в главе 9. Но важно иметь в виду, что код, приведенный выше, обладает прямой совместимостью.

Google рекомендует разработчикам вообще отказаться от концепции меню. Вместо этого желательно создавать пользовательские интерфейсы с такими объектами, которыми можно непосредственно манипулировать. Панель действий обладает хорошо скомпонованным интерфейсом, предназначенным для обработки глобальных поведений. Интерфейс адаптируется к экранам различных размеров.

После того как приложение Dots (Точки) будет усовершенствовано для работы с Ice Cream Sandwich (в описании приложения — `targetSdk Version="14"`) и с версиями Android выше Honeycomb, аналогичный код отобразит специальную кнопку действия на панели действий.

Эта кнопка, изображенная в виде корзины в правом верхнем углу на рис. 6.8, выполняет в приложении функцию очистки (Clear). Как и в предыдущей версии, при нажатии этого значка все точки исчезнут с экрана.

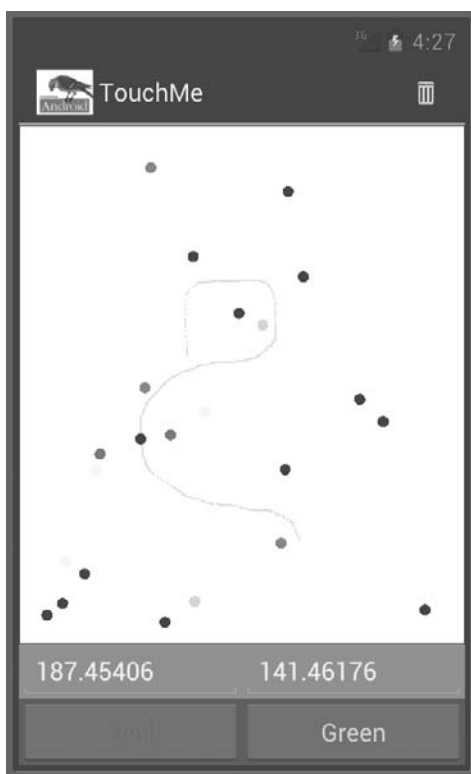


Рис. 6.8. Кнопка, выполняющая функцию очистки

Интересно, что при запуске этого приложения обнаруживается следующая особенность: добавленный элемент-меню работает практически всегда, кроме случаев, когда вид `DotView` находится в фокусе. Вы догадываетесь почему?

Если вы полагаете, что проблема заключается в `OnKeyListener`, установленном в `DotView`, то вы правы! Как показано в примере 6.16, слушатель захватывает

событие нажатия клавиши и возвращает `true` при ее нажатии. Из-за этого нажатие клавиши меню не проходит стандартной обработки, свойственной для `View`. Чтобы меню заработало, `OnKeyListener` нуждается в новом условии для проверки.

Пример 6.16. Улучшенная обработка нажатия на клавишу

```
switch (keyCode) {  
    case KeyEvent.KEYCODE_MENU:  
        return false;  
    // ...
```

Во фреймворке пользовательского интерфейса Android также поддерживаются контекстные меню при помощи класса `ContextMenu`. Такое меню появляется в ответ на долгое нажатие в виджете, который это меню поддерживает. Код, необходимый для добавления контекстного меню в приложение, практически аналогичен коду обычного меню, показанного выше, за тем исключением, что при работе применяются методы `onCreateContextMenu` и `onContextItemSelected`. Нужен еще один вызов. Для поддержки контекстных меню виджету необходимо установить слушатель `View.OnCreateContextMenuListener`, вызывая метод `setOnCreateContextMenuListener`, относящийся к `View` этого виджета. К счастью, поскольку `Activity` реализует интерфейс `View.OnCreateContextMenuListener`, обычная процедура установки выглядит как код из примера 6.17.

Пример 6.17. Установка `ContextMenuListener`

```
findViewById(R.id.ctxtMenuView).setOnCreateContextMenuListener(this);
```

Путем простого переопределения стандартных пустых методов `Activity` ваше приложение получает контекстное меню.

Отладка и оптимизация видов

В Android SDK предоставляются инструменты, помогающие понять принципы функционирования вашего дерева видов. Первый из таких инструментов до недавнего времени назывался `layoutopt`. В последних версиях инструментария Android (Android Tools) все возможности `layoutopt` (а также многие другие) вошли в состав нового инструмента под названием `Lint`. В то время как `layoutopt` был всего лишь инструментом командной строки, `Lint` можно использовать и из командной строки, и, что гораздо удобнее, из Eclipse. Он осуществляет специализированные ситуативные проверки XML-файлов, образующих макет приложения.

Любой, кто работает с Android уже достаточно долго, знает, насколько много проблем возникает с XML-ресурсом (макетом или меню), который синтаксически верен, но содержит семантические ошибки. Если `aapt` — инструменту, обрабатывающему каталог `res`, — не удастся правильно собрать каталог `gen`, то половина файлов проекта не сможет скомпилироваться (из-за отсутствия класса `R`). Например, если вы работаете с Eclipse, то весь ваш проект запрестрит красными ошибками — они будут везде, но не там, где возникла проблема. Инструмент `Lint` может помочь идентифицировать подобные проблемы.

Кроме того, Lint полезен при поиске потенциальных проблем в ресурсах проекта. Он осуществляет общие (так называемые «санитарные») проверки, а также проверки на соответствие рекомендациям; количество таких рекомендаций растет с каждым новым релизом инструментов для разработки в Android. В Eclipse Lint запускается после каждого изменения ресурсов, а найденные им проблемы выводятся как в виде стандартных предупреждений Eclipse, так и в специальном окне Lint Warnings (Предупреждения Lint).

Иногда Lint может выявить такие проблемы, о которых вы даже не подозревали. Порой программа выполняет автоматическое исследование и рефакторинг, чтобы решить эти проблемы. Инструмент пока еще дорабатывается и иногда жалуется на вещи, которые работают вполне нормально, но и потенциальные проблемы он распознает очень хорошо. Никогда не помешает разобраться в проблеме и убедиться, что она действительно несущественна.

Поскольку Lint — это статический анализатор, тщательно проверяющий исходный код, он не в состоянии выявить определенные классы проблем. Например, он не может находить виджеты, динамически добавляемые к виду. К счастью, в инструментарии Android есть и такой компонент, который может анализировать актуальное дерево видов работающего приложения. Это просмотрщик иерархии (hierarchy viewer), многофункциональный инструмент, способный работать в нескольких режимах. Хотя ранее некоторые его части были интегрированы в Eclipse, эти перспективы уже не работают в новейших версиях инструментария. Просмотрщик иерархии необходимо использовать из командной строки. В SDK он находится в каталоге tools и лучше всего работает с версией Android 9 (Gingerbread) или выше.

Просмотрщик иерархии действует в качестве отладчика и анализирует работающее приложение. Необходимо, чтобы он мог найти подключенное устройство или эмулятор. Если инструментарий подключается к работающему устройству, вы можете загрузить иерархию видов при помощи специальной кнопки в верхней части экрана. Нажав эту кнопку, вы получите экран примерно как на рис. 6.9.

В данном окне сообщается обширная информация, а в документации по инструментарию весьма подробно описано, как пользоваться этими сведениями. Но самой важной функцией, разумеется, является возможность просматривать все дерево видов сразу.

Как будет показано в главе 8, затраты на отрисовку дерева видов стремительно возрастают по мере того, как дерево становится глубже. При оптимизации приложения важно, в частности, сохранять его дерево максимально компактным. В этом окне вы сможете просмотреть, насколько глубоким является дерево видов, ведущее к конкретному окну, и решить, как уменьшить его глубину.

Три цветные точки помогают оценить, сколько времени вид тратит на каждый из трех этапов отображения: измерение, компоновку и отрисовку. Вид, помеченный зеленой точкой, проходит этап быстрее, чем это в среднем происходит с видами, отображенными в окне. Вид с желтой точкой тратит больше времени, чем уходит на это в среднем, а виды, обрабатываемые медленнее всех в дереве, помечаются красными точками. Приблизительное сравнение всех элементов дерева видов позволяет выявить проблемные места и подсказать вам, как лучше исправить притормаживающее приложение.

7 Фрагменты и многоплатформенная поддержка

Теперь, когда вы уже написали немного кода для Android, вы знаете, что Activity, View и подклассы View, отвечающие за макет и виджеты, — это одни из важнейших классов Android. Как правило, пользовательский интерфейс создается из виджетов, организованных в макеты, например ListView в LinearLayout. Единая иерархия объектов вида загружается из ресурса (или создается кодом) при запуске Activity. Она инициализируется и отображается на экране устройства.

На небольших экранах все нормально: пользователи переходят с экрана на экран, чтобы попасть в различные части графического интерфейса программы, а класс Activity (активность; так в Android понимается концепция задачи) поддерживает стек переходов назад (back stack), который обеспечивает быстрое и интуитивно понятное перемещение по интерфейсу, имеющему строгую древовидную структуру. Однако ситуация коренным образом меняется, если мы говорим о широком экране планшета. Содержимое в одних частях экрана остается неизменным дольше, чем в других. Содержимое некоторых частей экрана может влиять на содержимое других частей. Метафора карточной колоды этого не отменяет.

Вполне реально создавать такие пользовательские интерфейсы, в которых отдельные части экрана изменяются в ответ на действия в других частях экрана. Изменение происходит путем обычного отображения новых видов и скрытия старых. Но разработчики Android решили, что обычного соглашения будет недостаточно, чтобы стимулировать создание великолепных широкоэкранных пользовательских интерфейсов, обладающих единообразным внешним видом и поведением. Чтобы продвигать такие новые варианты взаимодействия, были реализованы некоторые функции, основанные на классе Fragment, входящем в комплект для разработки ПО Android 3.0 (API 11, Honeycomb).

Объект Fragment совмещает черты вида View и активности Activity. Подобно View, он может входить в состав ViewGroup или быть частью макета страницы. Но фрагмент не является подклассом View, и добавить его в ViewGroup можно только при помощи FragmentTransaction. Подобно Activity, фрагмент обладает жизненным циклом, который реализует интерфейсы ComponentCallbacks и ContextMenuListener. Но в отличие от активности у фрагмента нет контекста (Context), и его жизненный цикл зависит от контекста той активности, к которой этот фрагмент относится.

Фрагменты — это основное нововведение в API Android. Чтобы упростить переход на новый API, Google предоставляет библиотеку обеспечения совместимости,

которая поддерживает такие функции вплоть до версии SDK 1.6 (API 4, Donut). Чуть ниже мы рассмотрим проблему обратной совместимости. Но сначала познакомимся с фрагментами в их нативном окружении — Android 3.0 и выше.

Создание фрагмента

Как и любой другой объект, фрагмент может либо входить в состав XML-определения макета, либо добавляться в вид при помощи программирования. В макете фрагмент выглядит так:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

    <fragment
        class="com.oreilly.demo.android.ch085.contactviewer.DateTime"
        android:id="@+id/date_time"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        />

</LinearLayout>
```

Активность будет использовать этот макет обычным способом:

```
@Override
public void onCreate(Bundle state) {
    super.onCreate(state);
    setContentView(R.layout.main);
}
```

Сейчас такой код уже, вероятно, кажется знакомым. Единственный новый элемент в файле `main.xml` — это `ter fragment`. Тег использует атрибут `class` для указания полностью квалифицированного имени того класса, который реализует фрагмент. На класс реализации фрагмента налагается ряд условий. Данная конкретная реализация класса — `com.oreilly.demo.android.ch085.contactviewer.DateTime`, а условия таковы:

- должен существовать класс с точно таким же именем, которое указано. Этот класс должен быть видим из приложения;
- именованный класс должен быть подклассом от `Fragment`.

Хотя оба этих момента вполне можно проверить статически, текущие инструменты Android этого не делают. Придется проверить выполняемость обоих условий вручную.

Фреймворк Android создает новый экземпляр именованного класса в процессе инфляции шаблона. Такая ситуация предполагает необычные предпосылки. Это означает, что у класса должен быть безаргументный конструктор. Такой конструктор в языке Java предоставляется по умолчанию. В документации по разработке для Android рекомендуется (причем настоятельно) не определять никаких

конструкторов в любых подклассах `Fragment`, поскольку новый объект `Fragment` в момент создания может не иметь согласованного состояния (`consistent state`). В документации указано, что инициализацию фрагмента лучше производить позже в жизненном цикле фрагмента.

Независимо от того, как вы используете фрагмент где-либо в приложении, действует правило: если вы используете его в макете, то должна быть возможность создать его в процессе инфляции, не сообщая при этом параметры инициализации. Кроме того, фрагмент, создаваемый таким образом, должен быть готов для какой-нибудь полезной работы даже без инициализации. Например, фрагмент, отображающий контент из переданного URL, должен иметь возможность обрабатывать ситуацию, в которой URL — и, следовательно, контент — пуст.

Вот, например, очень простой фрагмент:

```
public class DateTime extends Fragment {
    private String time;

    public void onCreate(Bundle state) {
        super.onCreate(state);
        if (null == time) {
            time = new SimpleDateFormat("d MMM yyyy HH:mm:ss")
                .format(new Date());
        }
    }

    @Override
    public View onCreateView(
        LayoutInflater inflater,
        ViewGroup container,
        Bundle b)
    {
        View view = inflater.inflate(
            R.layout.date_time,
            container,
            false); // !!! Это важно!

        ((TextView) view.findViewById(R.id.last_view_time))
            .setText(time);

        return view;
    }
}
```

В этом коде показано несколько очень важных моментов. Само существование метода `onCreate`, относящегося к жизненному циклу, должно напоминать о классе `Activity` и методах его жизненного цикла. Хотя жизненный цикл `Fragment` не идентичен жизненному циклу `Activity`, у них довольно много общих методов. Как и при работе с активностью, метод фрагмента `onCreate` вызывается при инициализации этого фрагмента. Именно здесь лучше всего производить инициализацию, которую мы отложили при работе с конструктором. Пример гарантирует, что значение переменной `time` (именно ее отображением и будет заниматься фрагмент) будет правильно инициализировано.

У фрагментов есть несколько дополнительных методов жизненного цикла, в том числе `onCreateView`, который также используется в данном примере. Метод `onCreateView` вызывается при инициализации вида фрагмента (в отличие от `onCreate`, который вызывается, когда инициализируется сам фрагмент). Обратите внимание: фрагмент создает вид, которым будет управлять, используя переданный `LayoutInflater` для инстанцирования подсекции вида (view shard) `R.layout.date_time`. Простая подсекция вида — состоящая всего лишь из пары `TextView`s в `RelativeLayout` — определяется в собственном файле `layout/date_time.xml` (он здесь не приведен), почти как и основной вид, показанный выше.

Обращаем также ваше внимание на то, что это не стандартная двухаргументная версия `inflate`, здесь есть третий булев параметр, имеющий значение `false`. Это небольшая хитрость. Но она важна! У инфлятора должен быть доступ к `container`, к тому виду, который в итоге окажется родительским элементом только что созданной подсекции вида. Родительский вид необходим для правильной обработки макета. Предположим, например, что `container` — это `RelativeLayout`, указывающий положение только что созданной подсекции вида при помощи директивы `layout_toRightOf`.

С другой стороны, фреймворк фрагментов владеет тем видом, который возвращается методом `onCreateView`, и может управлять этим видом. Код метода `onCreateView` должен прикреплять подсекцию вида к ее контейнеру, как это обычно происходит в процессе инфляции. Этот третий аргумент является флагом, сообщающим инфлятору, что фреймворк фрагментов управляет ситуацией и что инфлятор не обязан прикреплять подсекцию вида к контейнеру.

После того как подсекция вида будет создана, метод `findViewById` можно будет использовать для нахождения других, вложенных виджетов. В данном примере мы найдем `TextView`, в котором отображается время, и поставим в подсекцию вида значение переменной `time`, инициализированной в `onCreate`.

При запуске это приложение будет выглядеть как на рис. 7.1.

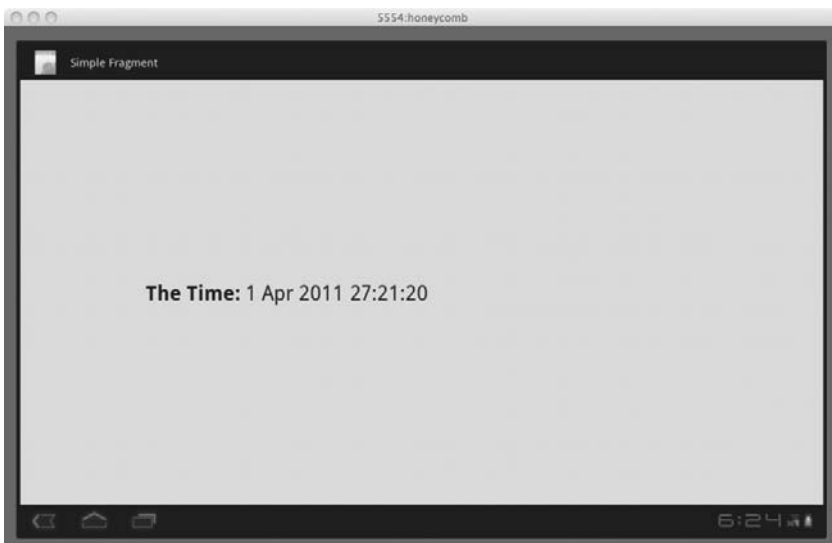


Рис. 7.1. Простой фрагмент

Жизненный цикл фрагмента

Запустив это приложение таким образом, как это было описано выше, и повернув экран во время его работы, вы заметите, что дата изменяется при каждом повороте экрана. При повороте экрана приложение уничтожается и создается заново. В такой версии приложение полностью теряет состояние при каждом повороте экрана.



Есть отличный способ протестировать приложение в эмуляторе. При нажатии Ctrl+F11 экран эмулятора поворачивается на 90°. В ответ на такое действие Android пропускает активность практически через весь жизненный цикл. Таким простым образом вы проверяете почти весь ваш код.

В данном тривиальном приложении-примере потеря состояния не так важна. А вот настоящее приложение не должно терять состояния. Пользователь будет раздражаться, если, например, при переводе телефона в альбомный режим во время просмотра какой-либо веб-страницы браузер будет возвращаться на домашнюю страницу.

Чтобы приложение сохраняло состояние, в код нужно внести всего два небольших изменения. Во-первых, в жизненном цикле фрагмента `DateTime` необходимо переопределить метод `onSaveInstanceState` таким образом, чтобы он сохранял свое состояние. Во-вторых, следует изменить метод `onCreate` так, чтобы он восстанавливал сохраненное состояние. Как и при работе с активностями (см. раздел «Жизненные циклы компонентов» главы 3), фреймворк Android предоставляет для первого метода объект `Bundle`, когда работа фрагмента приостанавливается. Такой же объект (`Bundle`) предоставляется методу `onCreate` при реконструировании клона приостановленного фрагмента.

Для того чтобы добавить поддержку состояния, изменим два следующих метода:

```
@Override
public void onCreate(Bundle state) {
    super.onCreate(state);

    if (null != state) { time = state.getString(TAG_DATE_TIME); }

    if (null == time) {
        time = new SimpleDateFormat("d MMM yyyy HH:mm:ss")
            .format(new Date());
    }
}

@Override
public void onSaveInstanceState(Bundle state) {
    super.onSaveInstanceState(state);
    state.putString(TAG_DATE_TIME, time);
}
```

И все. Если пропустить такую версию программы через весь жизненный цикл, ее состояние больше теряться не будет. Кстати, отметим, что, поскольку переменная `time` (и вообще любое состояние фрагмента) инициализируется в методе `onCreate`, ее нельзя объявить как `final`. Таким образом, снижается польза от применения конструктора при задании состояния фрагмента. К тому же так соблюдается рекомендация, в соответствии с которой подклассы `Fragment` вообще не должны иметь явных конструкторов.

В документации по разработке для Android¹ описан полный жизненный цикл фрагмента. Но один из методов обратного вызова этого жизненного цикла, `onPause`, заслуживает особого внимания. Метод `onPause` важен во фрагменте по той же причине, по которой он важен и в активности. Чтобы приложение хорошо вписывалось в рабочую среду Android, оно не должно ничего делать (использовать процессор, расходовать заряд батареи), пока это приложение является невидимым. Среда Android предусматривает вызов метода `onPause`, относящегося к фрагменту, всякий раз, когда этот фрагмент является невидимым. В этом методе фрагмент должен высвобождать все ресурсы, которые он, возможно, удерживает, завершать все долговременные процессы, которые мог начать, и т. д. Метод `onResume` должен реконструировать ресурсы и соответствующим образом перезапуститься.

Менеджер фрагментов

Как было указано выше, фрагменты можно создавать из программного кода, а также в макетах. Программное управление фрагментами осуществляется при помощи экземпляра класса `FragmentManager`, получаемого от `Activity` посредством метода `getFragmentManager`. Менеджер фрагментов обрабатывает три важные группы операций: тегирование и нахождение фрагмента, выполнение транзакций и работу со стеком переходов назад. Дополним нашу экспериментальную программу всеми этими компонентами и по очереди их исследуем.

Чтобы адаптировать приложение-пример к использованию фрагментов, создаваемых при помощи программирования, требуется внести всего два изменения: одно в макете `main.xml`, а другое — в активности `SimpleFragment`. В макете элемент-фрагмент заменяется практически идентичным `FrameLayout`:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

    <FrameLayout
        android:id="@+id/date_time"
```

¹ Можно найти по адресу <http://developer.android.com/guide/topics/fundamentals/fragments.html#Creating>. — *Примеч. пер.*

```
android:layout_width="fill_parent"  
android:layout_height="fill_parent"  
android:background="@color/green"  
</>
```

```
</LinearLayout>
```

SimpleFragment по-прежнему будет использовать этот макет, как и ранее. Но теперь макет не будет автоматически создавать новый фрагмент. Эта операция будет происходить в следующем коде:

```
@Override  
public void onCreate(Bundle state) {  
    super.onCreate(state);  
  
    setContentView(R.layout.main);  
    FragmentManager fragMgr = getFragmentManager();  
  
    FragmentTransaction xact = fragMgr.beginTransaction();  
    if (null == fragMgr.findFragmentByTag(FRAG1_TAG)) {  
        xact.add(R.id.date_time, new DateTime(), FRAG1_TAG);  
    }  
    xact.commit();  
}
```

Эти изменения не вносят в приложение никаких новых функций, а лишь добавляют менеджер транзакций фрагментов. При запуске эта версия приложения работает так же, как и наша первая версия, построенная на основе макетов.

Важной особенностью данного фрагмента кода является использование тегов. Вполне возможна ситуация, в которой метод onCreate, относящийся к активности, будет вызываться, будучи по-прежнему связанным с фрагментом, который был создан ранее. Если просто добавлять новый фрагмент при каждом вызове onCreate, то произойдет своеобразная утечка фрагментов. Чтобы избежать этого, используются функции тегирования и нахождения, предоставляемые менеджером фрагментов.

Третий аргумент метода add — это уникальный тег (метка), присваиваемый фрагменту, когда этот фрагмент добавляется к активности. После создания тега метод findFragmentByTag менеджера фрагментов может использоваться для восстановления конкретного, отдельно взятого фрагмента, который добавлен с указанным тегом. В примере мы проверяем, существует ли уже фрагмент с заданным тегом, перед тем как создать новый экземпляр фрагмента. Если такого фрагмента нет, то он создается. Если фрагмент уже существует, то никаких действий не требуется. Так мы гарантируем, что каждую конкретную роль будет играть только один фрагмент, и боимся от утечек, связанных с фрагментами.

Тегирование и нахождение также могут использоваться и в других целях. Когда активности требуется сообщить о каких-нибудь изменениях состояния прикрепленного фрагмента, для этого активность обычно заранее тегирует этот фрагмент, а потом, когда он потребуется, использует FragmentManager для поиска тега и получения ссылки на него.

Транзакции фрагмента

Кроме применения тегирования, новый код также касается транзакций фрагментов. Еще раз дополним приложение, чтобы продемонстрировать ценность таких транзакций.

Прежде чем приниматься за транзакции, необходимо сделать небольшое отступление. Ранее мы отмечали, что в документации по разработке для Android рекомендуется, чтобы подклассы фрагмента не имели явных конструкторов. Итак, как же внешний объект передает состояние инициализации новому фрагменту? Класс `Fragment` поддерживает два метода, `setArguments` и `getArguments`, обеспечивающих такую возможность. Соответственно они позволяют внешнему вызывающему элементу — им может быть создатель фрагмента — сохранять пакет (`Bundle`) во фрагменте, а фрагменту — восстанавливать этот пакет когда-нибудь позднее.

Такая тонкая комбинация нового экземпляра фрагмента, пакета `Bundle` и вызова к `setArguments` работает во многом так же, как и конструктор. Поэтому целесообразно объединять эти элементы в статический фабричный метод, относящийся к объекту `Fragment`, вот так:

```
public static DateTime createInstance(Date time) {
    Bundle init = new Bundle();
    init.putString(
        DateTime.TAG_DATE_TIME,
        getDateTimeString(time));

    DateTime frag = new DateTime();
    frag.setArguments(init);
    return frag;
}

private static String getDateTimeString(Date time) {
    return new SimpleDateFormat("d MMM yyyy HH:mm:ss")
        .format(time);
}
```

Теперь этот статический фабричный метод можно использовать в методе `onCreate` класса `SimpleFragment` для создания нового экземпляра фрагмента, который будет корректно инициализирован переданным пакетом `Bundle`. Данный код практически идентичен предыдущей версии, за исключением того, что теперь используется статический фабричный метод, относящийся к `DateTime`, и ему передается аргумент:

```
@Override
public void onCreate(Bundle state) {
    super.onCreate(state);

    setContentView(R.layout.main);

    FragmentManager fragMgr = getFragmentManager();

    FragmentTransaction xact = fragMgr.beginTransaction();
    if (null == fragMgr.findFragmentByTag(FRAG1_TAG)) {
```

```

        xact.add(
            R.id.date_time,
            DateTime.newInstance(new Date()),
            FRAG1_TAG);
    }
    xact.commit();
}

```

Наконец, относящийся к фрагменту метод onCreate получает данные для инициализации из пакета, переданного в качестве аргумента, если только речь не идет о состоянии из предыдущей инкарнации метода:

```

@Override
public void onCreate(Bundle state) {
    super.onCreate(state);

    if (null == state) { state = getArguments(); }

    if (null != state) { time = state.getString(TAG_DATE_TIME); }

    if (null == time) { time = getDateTimeString(new Date()); }
}

```

Приложение, модифицированное до такой степени, все еще работает так же, как и исходный вариант. Но реализация значительно изменилась и стала гораздо более гибкой. В частности, у нас есть фрагмент, который можно инициализировать извне и использовать для демонстрации транзакций.

Идея транзакции фрагмента, как следует из названия, заключается в том, что все изменения происходят в результате единичного, «атомарного» действия. Чтобы это продемонстрировать, в последний раз дополним нашу программу-пример: добавим возможность создавать фрагменты в парах.

Вот новый макет:

```

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

    <Button
        android:id="@+id/new_fragments"
        android:layout_width="fill_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:textSize="24dp"
        android:text="@string/doi"
        />

    <FrameLayout
        android:id="@+id/date_time2"
        android:layout_width="fill_parent"

```

```

        android:layout_height="0dp"
        android:layout_weight="2"
        android:background="@color/blue"
    />

```

```

<FrameLayout
    android:id="@+id/date_time"
    android:layout_width="fill_parent"
    android:layout_height="0dp"
    android:layout_weight="2"
    android:background="@color/green"
/>

```

```

</LinearLayout>

```

Вот соответствующие дополнения к методу onCreate в SimpleFragment:

```

public void onCreate(Bundle state) {
    super.onCreate(state);

    setContentView(R.layout.main);

    ((Button) findViewById(R.id.new_fragments))
        .setOnClickListener(
            new Button.OnClickListener() {
                @Override
                public void onClick(View v) { update(); }
            });

    Date time = new Date();

    FragmentManager fragMgr = getFragmentManager();

    FragmentTransaction xact = fragMgr.beginTransaction();
    if (null == fragMgr.findFragmentByTag(FRAG1_TAG)) {
        xact.add(
            R.id.date_time,
            DateTime.newInstance(time),
            FRAG1_TAG);
    }

    if (null == fragMgr.findFragmentByTag(FRAG2_TAG)) {
        xact.add(
            R.id.date_time2,
            DateTime.newInstance(time),
            FRAG2_TAG);
    }

    xact.commit();
}

```

Наконец, изменяется и наше приложение-пример. Теперь при запуске оно выглядит как на рис. 7.2.

**Рис. 7.2.** Транзакции фрагмента

Оба фрагмента отображают одни и те же дату и время, так как обоим фрагментам сообщается одинаковое значение. Теперь можно прерываться на работу с другими приложениями, а потом возвращаться к нашей программе-примеру, можно поворачивать дисплей — наше демо не потеряет состояния. Программа довольно надежная. А теперь сделаем реализацию кнопки (пример 7.1).

Пример 7.1. Замена фрагмента

```
void update() {
    Date time = new Date();

    FragmentTransaction xact
        = getFragmentManager().beginTransaction();

    xact.replace(
        R.id.date_time,
        DateTime.newInstance(time),
        FRAG1_TAG);

    xact.replace(
        R.id.date_time2,
        DateTime.newInstance(time),
        FRAG2_TAG);

    xact.addToBackStack(null);
    xact.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN);

    xact.commit();
}
```

Этот метод использует именно атомарность транзакций фрагмента. Он во многом напоминает код инициализации фрагмента из метода `onCreate`, относящегося к `SimpleFragment`. Но вместо того, чтобы использовать транзакции для добавления новых экземпляров фрагмента, этот код замещает существующие фрагменты. Вызов `commit` в конце метода приводит к тому, что оба новых фрагмента становятся видимы одновременно. Время в верхнем и нижнем видах всегда будет идти синхронно.



Фрагмент, создаваемый в макете (при помощи тега `fragment` из языка XML), ни в коем случае нельзя заменять фрагментом, который создан динамически. Хотя на вид отличить первый вариант фрагмента от второго очень сложно, их жизненные циклы во многом несхожи. Вы вполне можете пользоваться в приложении фрагментами обеих разновидностей, но никогда не заменяйте фрагменты одной разновидности фрагментами другой. Например, если попытаться применить `setContentView` к макету, в котором «разметочный» фрагмент был заменен «программным», то будут возникать ошибки, которые сложно находить и устранять. Частым симптомом проблем такого рода является исключение `IllegalStateException` с сообщением `Fragment did not create a view` (Фрагмент не создал вида).

Итак, мы подходим к последней важной составляющей фрагмента — стеку переходов назад (`back stack`). Если последовательно запустить несколько активностей, то к ним можно будет вернуться в обратном порядке, воспользовавшись кнопкой «Назад». Такое поведение свойственно и для транзакций фрагмента.

Запустив это приложение, вы увидите на экране такую же картинку, как на рис. 7.2. Если нажать кнопку, расположенную в верхней части экрана, то голубой и зеленый фрагменты одновременно обновятся. Но еще лучше, если вы нажмете кнопку «Назад» (значок со стрелкой, указывающей влево, в нижнем левом углу дисплея), — вы увидите в обратном порядке все обновления, которые сделали, нажимая кнопку «Пуск». Например, если в обоих фрагментах отображается время 5 Apr 2011 12:49:32, а вы нажимаете кнопку «Пуск», то дисплей может обновиться так, что в обоих фрагментах — голубом и зеленом — будут показаны дата и время 5 Apr 2011 13:02:43. Если после этого нажать кнопку «Назад», то в обоих фрагментах опять будет показано время 5 Apr 2011 12:49:32. Целая транзакция — обновление обоих фрагментов — записывается в стек переходов назад как единое событие. Когда вы нажимаете кнопку «Назад», целая транзакция удаляется, открывая состояние, которое наступило в результате предыдущей транзакции.

Пакет поддержки

Один из важнейших аспектов, связанных с фрагментами, заключается в том, что, хотя они и появились в версии Android 3.0, а в более ранних версиях API отсутствуют, Google предоставляет пакет поддержки, чтобы функции, связанные с фрагментами, можно было использовать и на тех устройствах, где до сих пор работают более старые версии Android.

Пакет поддержки имеет ограничения. Приложение, использующее пакет поддержки, нуждается в нем, даже если работает в системе Android 3.0. Разрабатывая стратегию внедрения пакета поддержки, Google столкнулась с настоящей голово-

ломкой. Даже если бы удалось реализовать пакет совместимости так, чтобы приложение можно было прозрачно, без изменений портировать с Android 3.0 на Android 2.0, то проблема все равно бы возникала. Пакет поддержки приходилось бы включать как библиотеку в каждую использующую его программу. Если бы в пакете поддержки определялись классы, одноименные тем, что присутствуют в Android 3.0, то приложение, использующее такой пакет, определяло бы классы, которые вступали бы в конфликт со своими системными определениями. Во избежание этого потребовалось бы производить весьма хитрые манипуляции с загрузчиком классов.

Поэтому пакет поддержки работает на специальном базовом пакете, `android.support.v4`, в котором определяются характеристики, связанные с совместимостью. Чтобы программа, разработанная под Android 3.0, могла использовать пакет поддержки, в ее код потребуется внести изменения. Как минимум нужно будет внести следующие изменения.

- Скопируйте библиотеку пакета поддержки в ваш проект. В верхнем уровне этого проекта создайте каталог `lib` и скопируйте архив `android-support-v4.jar` из папки с Android SDK `extras/android/compatibility/v4/` в этот каталог `lib`.
- Добавьте пакет поддержки в путь сборки проекта. Если вы работаете с Eclipse, выберите эту библиотеку в диспетчере пакетов (возможно, чтобы увидеть ее, понадобится обновить проект: для этого нажмите **F5** или щелкните левой кнопкой мыши и нажмите **Refresh** (Обновить)). После того как библиотека выбрана, выполните **Build Path** ► **Add to Build Path** (Путь сборки ► Добавить в путь сборки)). Для быстрого выполнения двух этих шагов можно (в диспетчере пакетов в контексте вашего приложения) щелкнуть левой кнопкой мыши, затем выбрать **Android Tools** (Инструменты Android), а потом выполнить **Add Compatibility Library** (Добавить библиотеку совместимости).
- Измените целевую сборку проекта с Android 3.0 на Android 2.0.1 (**Properties** ► **Android** (Свойства ► Android)). Из-за изменения целевой версии сборки возникнет множество ошибок.
- Некоторые импорты, касающиеся `android.app`, придется обновить, чтобы они ссылались на `android.support.v4.app`. При условии, что в вашей программе не было ошибок до изменения ее целевой сборки, вы встретите только поврежденные импорты и обновите их базовый пакет.
- Все активности приложения, использующего фрагменты, должны быть обновлены так, чтобы вместо подкласса `Activity` применялся `FragmentActivity`.
- Все вызовы к `getFragmentManager` нужно заменить на вызовы к `getSupportFragmentManager`.
- Исправьте все оставшиеся ошибки и протестируйте программу.

Фрагменты и макет

В предыдущем разделе было рассказано, как сделать проект, рассчитанный на работу с фрагментами, чтобы он компилировался на версиях Android ниже Honeycomb.

Но в большинстве прикладных задач с этого все только начинается. Достаточно упомянуть хотя бы о том, что если приложение скомпилируется для Froyo, это отнюдь не означает, что оно будет хорошо выглядеть на устройстве с Froyo. На самом деле гарантировать создание красивого пользовательского интерфейса нельзя ни при каких обстоятельствах. Но в этом разделе мы обсудим некоторые проблемы и подскажем, как их можно решить.

Откровенно говоря, хорошего тут мало. Если приложение предполагается использовать и на широкоэкранном телевизоре с диагональю 1080 дюймов, который удобно смотреть с заднего сиденья семейного автомобиля, и на 12-дюймовом планшете, в обоих случаях в альбомной ориентации, а также на телефоне с книжной ориентацией, то такое приложение просто не может обойтись одним пользовательским интерфейсом. Единственный способ обеспечить отличное представление такого приложения на любом из упомянутых устройств — разрабатывать интерфейс с расчетом на каждое из этих устройств. Дополнительная настройка, необходимая для портирования пользовательского интерфейса, может ограничиться лишь парой корректировок графики и шрифта. С другой стороны, может понадобиться значительно изменить поток задач в пользовательском интерфейсе.

Простейший способ адаптировать пользовательский интерфейс к разнообразным устройствам — использовать *квалификатор конфигурации*. Это просто суффикс, добавляемый к имени каталога в дереве каталогов с ресурсами приложения. Таким образом, в приложении может содержаться несколько версий одного ресурса, предназначенных для использования в определенном контексте.

Рассмотрим, например, активность с экраном, описанным в едином ресурсе макета, `main.xml`. На этот макет ставится ссылка из кода, это делается при помощи константы `R.layout.main`. В исходниках проекта ресурс определяется в едином файле `.../res/layout/main.xml`.

При применении квалификаторов конфигурации единственная константа `R.layout.main` может иметь несколько определений, например одно в `.../res/layout-land/main.xml`, а другое в `.../res/layout-port/main.xml`. Система Android во время исполнения разрешает одну константу либо в книжную, либо в альбомную ориентацию, в соответствии с текущей ориентацией дисплея устройства.

Существуют квалификаторы конфигурации, описывающие ориентацию экрана, пиксельную плотность, соотношение сторон и абсолютный размер. Есть и другие квалификаторы, описывающие языковые и региональные настройки, режим привязки (*docking mode*), ночной или дневной вариант дисплея. Хотя маловероятно, что вам когда-либо понадобится такая подробная квалификация, в принципе, возможно, чтобы одна ресурсная константа имела тысячи вариантов — по одному на каждую из возможных комбинаций значений квалификаторов.

Квалификаторы конфигурации не слишком облегчают работу с множеством контекстов; они просто делают эту работу практически выполнимой. В инструменте разработки Eclipse обеспечивается минимальная поддержка квалификаторов. На втором экране мастера установки, где создается новый файл ресурсов (**File** ▶ **New** ▶ **Other** ▶ **Android XML Values File** (Файл ▶ Создать ▶ Прочее ▶ Файл Android с XML-значениями)); затем нажимается кнопка **Next** (Далее), после того как файл ресурса получит имя) перечисляются квалификаторы конфигурации и позволяет добавить так много (или так мало) квалификаторов, как вам нужно.

Квалификаторы конфигурации особенно интересны при работе с пользовательскими интерфейсами, так как обеспечивают естественную адаптацию при переходе из книжного в альбомный режим и обратно. Если читатель, говорящий на одном из европейских языков, работает с устройством в альбомной ориентации, то типичный пользовательский интерфейс фрагмента, естественно, располагается так: контекст оказывается в левой части экрана, а его детализация — в правой.

Для книжной ориентации такое расположение, разумеется, не подходит. Но совсем не сложно будет создать и книжный, и альбомный варианты макета, воспользовавшись квалификаторами конфигурации и изменив ориентацию макета `LinearLayout`, где содержится два фрагмента, с горизонтальной на вертикальную.

По-прежнему предполагается, что на экране должно быть достаточно места, чтобы было целесообразно делить один экран на несколько областей. Не забывайте, что фрагмент — это сущность, позволяющая эффективно использовать большие экраны, а не универсальный элемент для оптимизации любого дизайна пользовательского интерфейса. Хотя некоторые современные телефоны и имеют достаточно большие экраны, чтобы на них можно было работать с фрагментами, на многих устройствах фрагменты неприменимы. На таких устройствах вам понадобится плавно переходить от нового, фрагменто-ориентированного пользовательского интерфейса к оригинальному пользовательскому интерфейсу Android, напоминающему колоду карт.

Существует несколько способов выбора одного стиля пользовательского интерфейса из нескольких. Один из вариантов — поддерживать несколько версий приложения, а на рынке приложений Android различать их, используя в файле описания *рыночные фильтры* (market filters). Например, файл описания для версии, адаптированной для работы с небольшим экраном, может включать такие строки:

```
<supports-screens
    android:largeScreens="false"
    android:xlargeScreens="false" />
```

А файл описания для версии, ориентированной на работу с большим экраном, может выглядеть так:

```
<supports-screens
    android:smallScreens="false"
    android:normalScreens="false"
    android:largeScreens="true"
    android:xlargeScreens="true" />
```

Реализация такой идеи оправдана лишь в случаях, когда различать версии приложения требуется по каким-то причинам, не связанным с размером экрана. Если, например, одна из версий приложения ориентирована на использование в конкретной среде — скажем, приложение было написано для применения в автомобиле и к нему была добавлена функция отслеживания расхода бензина, — то такое приложение определено будет использоваться на экране с нестандартными размерами. Поэтому было бы удобно выбирать стиль пользовательского интерфейса для такого приложения уже на рынке.

Использование рыночных фильтров для ассоциирования определенного стиля пользовательского интерфейса с конкретными устройствами — порочная практика,

хотя иногда такой прием и срабатывает. Даже если бегло просмотреть документацию по атрибутам рыночных фильтров, применяемых для различения размеров экрана, становится понятно, что эти атрибуты находятся в стадии разработки. Например, в 13-й версии API атрибуты `...Screens`, использованные в предыдущих примерах, заменены атрибутом `android:requiresSmallestWidthDp`. К сожалению, на рынке пока не работает фильтрация по этому новому атрибуту.

Существует стандартная идиома для выбора во время исполнения между фрагменто-ориентированным интерфейсом и интерфейсом, напоминающим колоду карт. Она связана с очень хитрым использованием конфигурационных квалификаторов. Изучим фрагменто-ориентированный интерфейс. Например, приложение предназначено для обычного просмотра контактов. Вы щелкаете на имени контакта в левой части экрана, и подробная информация по этому фрагменту выводится в правой части экрана. Макет выглядит примерно так:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

    <ListView
        android:id="@+id/contacts"
        android:layout_width="fill_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
    />

    <FrameLayout
        android:id="@+id/contact_detail"
        android:layout_width="fill_parent"
        android:layout_height="0dp"
        android:layout_weight="2"
        android:background="@color/blue"
    />

</LinearLayout>
```

Мы уже видели, как конфигурационные квалификаторы могут использоваться для создания нескольких версий данного макета в едином приложении. В предыдущем примере мы добавили второй подобный макет, оптимизированный для отображения в книжной ориентации. Чтобы сообщить среде времени исполнения Android, какой макет в каком случае использовать, мы применили конфигурационные квалификаторы.

Фокус с возвратом к карточному стилю пользовательского интерфейса связан с созданием нового макета, в котором вообще нет фрагментов:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
```

```

        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
    >

    <ListView
        android:id="@+id/contacts"
        android:layout_width="fill_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
    />

</LinearLayout>

```

Нам остается просто изменить код так, чтобы перед созданием фрагмента программа проверяла, хватает ли на экране места для этого фрагмента. Если места для фрагмента не хватает, мы возвращаемся к пользовательскому интерфейсу в виде колоды карт:

```

public void onCreate(Bundle state) {
    super.onCreate(state);

    setContentView(R.layout.main);

    final boolean useFrag
        = null != findViewById(R.id.contact_detail);

    if (useFrag) { installFragment(); }

    contacts = (ListView) findViewById(R.id.contacts);
    setAdapter(contacts);
    contacts.setOnItemClickListener(
        new AdapterView.OnItemClickListener() {
            @Override public void onItemClick(
                AdapterView<?> parent,
                View view,
                int position,
                long id)
            {
                if (useFrag) { stackFragment(/*...*/); }
                else { stackActivity(/*...*/); }
            }
        });
}

```

Методы `stackFragment` и `stackActivity` реализуют соответственно фрагменто-ориентированный и карточный стили пользовательского интерфейса. Метод `stackFragment` использует менеджер транзакций для замены актуального фрагмента, как показано в примере 7.1. Метод `stackActivity`, в свою очередь, запускает новую активность стандартным способом, используя новое намерение (`Intent`). В данном случае будет относительно несложно абстрагировать поведение контактной информации в отдельный класс, который может использоваться классами обоих вариантов пользовательского интерфейса — `ContactDetailsActivity` и `ContactDetailsFragment`.

Есть и другой способ выбора между вариантами интерфейса во время исполнения. Для этого можно применить относящийся к активности метод `startActivity-FromFragment`. Этот метод перехватывает любую попытку запуска намерения со стороны фрагмента.

Для использования этой технологии в нашей учебной программе нужно применить немного иную стратегию. Левое подокно, списковый вид, обязательно будет фрагментом независимо от стиля пользовательского интерфейса. Этот фрагмент *всегда* будет пытаться запустить намерение для отображения деталей. Если приложение работает на настолько маленьком экране, что там можно задействовать лишь интерфейс в виде колоды карт, то данное намерение будет запускать активность для показа детальной информации. Если же, напротив, на экране окажется достаточно места, чтобы отобразить пользовательский интерфейс, состоящий из фрагментов, корневая активность просто перехватит вышеописанную попытку запуска и отобразит `ContactDetailsFragment`. Код будет очень похожим. Ниже показаны два макета: один в стиле колоды карт, другой, соответственно, с фрагментами. Первое определение будет содержаться лишь в каталоге `res/layout-small` (квалификатор контента — `small`). Второй макет будет находиться в каталогах с макетами, имеющими другие квалификаторы (`large` и т. д.).

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

    <fragment
        class="com.oreilly.demo.android.contactviewer.ContactsFragment"
        android:id="@+id/contacts"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        />

</LinearLayout>

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

    <fragment
        class="com.oreilly.demo.android.contactviewer.ContactsFragment"
        android:id="@+id/contacts"
        android:layout_width="0dp"
        android:layout_height="fill_parent"
        android:layout_weight="1"
        />
```

```

<FrameLayout
    android:id="@+id/contact_detail"
    android:layout_width="0dp"
    android:layout_height="fill_parent"
    android:layout_weight="2"
    android:background="@color/blue"
/>

```

```
</LinearLayout>
```

Вот код для корневой активности. Фрагмент `ContactsFragment` будет автоматически наполняться информацией из `R.layout.main.StartActivityFromFragment` решает, запускать ли фрагмент `DetailFragment`. Выбор зависит от того, существует ли в макете элемент, к которому можно прикрепить фрагмент. При отсутствии такого элемента запускается новая активность.

```

public class ContactViewer extends FragmentActivity {
    private static final String FRAG_TAG
        = ContactViewer.class.getCanonicalName() + ".fragment";

    private boolean useFrag;

    @Override
    public void onCreate(Bundle state) {
        super.onCreate(state);

        setContentView(R.layout.main);

        useFrag = null != findViewById(R.id.contact_detail);

        if (useFrag) { installDetailsFragment(); }
    }

    @Override
    public void startActivityFromFragment(
        Fragment fragment,
        Intent intent,
        int requestCode)
    {
        if (!useFrag) { startActivity(intent); }
        else if (fragment instanceof ContactsFragment) {
            launchDetailFragment(intent.getExtras());
        }
    }

    // Методы опущены...
}

```

Наконец, поговорим о `ContactFragment`. Если происходит щелчок на списковом виде, то он всегда переадресуется как намерение. Корневая активность может выбрать: перенаправить ли намерение либо представить запрос в виде нового фрагмента.

```
public class ContactsFragment extends ListFragment {

    @Override
    public View onCreateView(
        LayoutInflater inflater,
        ViewGroup container,
        Bundle b)
    {
        View view = super.onCreateView(inflater, container, b);

        installListAdapter(getActivity());

        return view;
    }

    @Override
    public void onListItemClick(ListView l, View v, int pos, long row){
        Cursor cursor = (Cursor) getListAdapter().getItem(pos);

        String id = cursor.getString(
            cursor.getColumnIndex(BaseColumns._ID));
        String name = cursor.getString(
            cursor.getColumnIndex(Contacts.DISPLAY_NAME));
        Intent intent = new Intent();
        intent.setClass(getActivity(), ContactDetailActivity.class);
        intent.putExtra(ContactDetails.TAG_ID, id);
        intent.putExtra(ContactDetails.TAG_CONTACT, name);
        startActivity(intent);
    }

    // Методы опущены...
}
```

8 Рисование двухмерной и трехмерной графики

Коллекция виджетов Android и инструментов для сборки их в более крупные элементы довольно удобна и функциональна, помогает решать самые разнообразные задачи. Но что делать, если ни один из предлагаемых виджетов не подходит для конкретного случая? Может быть, в приложении требуется изобразить игральные карты, фазы Луны или энергию, направляемую на основные двигатели ракетного корабля? В таких случаях нужно уметь создавать собственные виджеты самостоятельно.

В этой главе мы сделаем обзор графики и анимации в Android. Она рассчитана на программистов, обладающих базовым опытом работы с графикой. Здесь довольно подробно обсуждаются вопросы, связанные с поворотами дисплея. При чтении данной главы вам определенно придется обращаться к документации по Android, особенно потому, что фреймворк продолжает меняться. Например, со времен выхода Honeycomb основные нововведения в реализации связаны с улучшенными возможностями аппаратного ускорения. Кроме того, существенно изменились способы оптимизации отрисовки элементов в приложении. Результаты применения новых технологий обязательно впечатлят ваших пользователей.

Создание собственных виджетов

Выше мы уже упоминали, что *виджет* — это просто удобный термин для обозначения подклассов `android.view.View`, как правило представляющих собой узлы «листья» на дереве видов. Такой узел одновременно реализует и вид и контроллер. Но внутренние узлы дерева видов, хотя и могут включать в себя сложный код, обычно достаточно просты в обращении, с точки зрения пользователя. Пусть термин «виджет» и является неофициальным, он полезен при обсуждении основных рабочих компонентов пользовательского интерфейса, содержащих информацию и реализующих поведение, которые важны для пользователя.

Можно достичь многого и не создавая новых виджетов. `TextView`, `Button`, `DatePicker` — все это примеры виджетов, предоставляемых в инструментарии пользовательского интерфейса Android. В этой книге мы уже сделали несколько приложений, состоящих только из готовых виджетов, а также простых подклассов данных виджетов. Код этих приложений просто выстраивает деревья видов, komponуя их в коде или загружая из макетов, хранящихся в XML-файлах ресурсов.

Непростое приложение MicroJobs, которое мы рассмотрим в главе 9, имеет вид, содержащий список названий, соответствующих пунктам на карте. По мере того как на карту добавляются новые пункты, новые виджеты с названиями пунктов динамически добавляются в список. Даже в этом динамически изменяемом макете используются только готовые виджеты; новые виджеты здесь не создаются. Технологии MicroJobs, образно говоря, добавляют или удаляют рамки из такого дерева, как показано на рис. 6.3.

Более сложный виджет (такой, в который могут вкладываться новые виджеты) должен быть подклассом ViewGroup, который, в свою очередь, является подклассом View. Очень сложный виджет, используемый, например, как инструмент интерфейса и реализуемый в нескольких местах (и даже несколькими приложениями), может представлять собой целый пакет классов, лишь один из которых является потомком View.

Эта глава рассказывает, как создать собственный виджет, а для этого придется заглянуть «под капот» вида (view). Глава посвящена графике, а значит, компоненту View (Вид) из паттерна «Модель-вид-контроллер» (MVC). Как было указано выше, виджеты также содержат код контроллера, и такое решение является хорошим, поскольку в таком случае весь код, важный для поведения и его представления на экране, располагается в одном месте. В этой главе мы обсудим только реализацию вида. О реализации контроллера было рассказано в главе 6.

Поэтому, сосредоточившись на графической информации, мы сможем разбить задачи, поставленные в этой главе, на две важные части: нахождение пространства на экране и рисование в данном пространстве. Первая задача связана с *компоновкой*, или *построением макета* (layout). «Листовой» виджет может сообщать о том, что ему требуется пространство, определяя метод onMeasure, который будет вызываться фреймворком пользовательского интерфейса Android в нужное время. Вторая задача — отображение виджета как таковое — обрабатывается методом onDraw, относящимся к виджету.

Макет

Большая часть сложной работы в механизме построения макетов во фреймворке Android реализуется при помощи *контейнерных видов*. Контейнерный вид, как понятно из названия, содержит другие виды. Такие виды являются внутренними узлами в дереве видов и в подклассах ViewGroup. Инструментарий фреймворка предоставляет сложные контейнерные виды, предлагающие мощные и адаптируемые стратегии размещения информации на экране. Например, часто применяются контейнерные виды LinearLayout и RelativeLayout. Пользоваться ими обоими относительно просто, а вот правильно реализовать самостоятельно — очень сложно. Поскольку удобные и мощные контейнерные виды и так существуют, вам, возможно, никогда не придется самостоятельно реализовывать такой вид или его алгоритм компоновки, о котором здесь пойдет речь. Но понимание того, как все это работает (как фреймворк пользовательского интерфейса Android управляет процессом компоновки), поможет вам писать правильные и надежные виджеты.

В примере 8.1 показан, пожалуй, простейший рабочий виджет, который может написать каждый. При добавлении такого виджета в дерево видов какой-нибудь активности (Activity) он будет заполнять выделенное ему пространство голубым цветом. Не очень интересно, но прежде, чем перейти к созданию чего-нибудь более сложного, внимательно рассмотрим, как в этом простом примере решаются две основные задачи компоновки и отрисовки. Начнем с процесса компоновки; рисование будет описано ниже, в подразделе «Рисование с применением Canvas (холста)».

Пример 8.1. Простейший виджет

```
public class TrivialWidget extends View {

    public TrivialWidget(Context context) {
        super(context);
        setMinimumWidth(100);
        setMinimumHeight(20);
    }

    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        setMeasuredDimension(
            getSuggestedMinimumWidth(),
            getSuggestedMinimumHeight());
    }

    @Override
    protected void onDraw(Canvas canvas) {
        canvas.drawColor(Color.CYAN);
    }
}
```

Динамический макет требуется потому, что потребность виджета в свободном пространстве также изменяется динамически. Предположим, например, что виджет в программе с поддержкой **GPS-навигации отображает название города, к которому** вы в настоящий момент едете на машине. Когда вы направляетесь из Эли в Пост Миллз, виджет получает уведомление об изменении местоположения. Но когда программа собирается переписать название города, то обнаруживает, что на экране мало места и название нового города там не умещается. Программа должна послать экрану запрос на перерисовку, так, чтобы для названия города освободилось больше места, если это возможно.

Компоновка может быть удивительно сложной задачей, которую весьма нелегко выполнить правильно. Вероятно, не так сложно добиться, чтобы конкретный «листовой» виджет правильно выглядел на отдельно взятом устройстве. С другой стороны, очень сложно бывает сделать такой виджет, который должен будет упорядочивать свои дочерние элементы так, чтобы они правильно выстраивались на разнообразных устройствах, даже когда параметры экрана меняются.

Процесс компоновки начинается, когда метод `requestLayout` активируется в каком-нибудь виджете из дерева видов. Как правило, виджет вызывает метод `requestLayout` применительно к самому себе, когда ему требуется дополнительное

пространство. Правда, этот метод может вызываться из любой точки приложения, чтобы указать, что определенный вид на отображаемом в данный момент экране уже испытывает недостаток пространства.

Метод `requestLayout` заставляет фреймворк пользовательского интерфейса Android поставить событие в очередь событий пользовательского интерфейса. Когда каждое из этих событий будет обработано в порядке очереди, фреймворк предоставит всем контейнерным видам возможность запросить у каждого из его дочерних виджетов, сколько места нужно этому виджету для отрисовки. Этот процесс делится на два этапа: измерение дочерних видов и последующее упорядочение их в новых позициях. Первую фазу должны реализовывать все виды, вторая же важна только для реализаций контейнерных видов, которым потребуется управлять компоновкой их дочерних видов.

Измерение

Цель фазы измерений — предоставить каждому виду возможность динамически запросить столько пространства, сколько ему в идеале требуется для отрисовки. Фреймворк пользовательского интерфейса начинает этот процесс, вызывая метод `measure` у корневого виджета дерева видов. Начиная с этой точки, каждый контейнерный вид спрашивает свои дочерние виды о том, сколько пространства им нужно. Вызов передается всем видам-потомкам, начиная с самых глубоких, и, таким образом, каждый дочерний элемент получает возможность рассчитать свой размер до того, как это сделает родительский элемент. Родительский элемент рассчитывает собственный размер в зависимости от размеров входящих в него дочерних элементов и сообщает полученные данные своему родительскому элементу. И так далее по всему дереву.

Например, в разделе «Сборка графического интерфейса» главы 6 самый верхний `LinearLayout` опрашивает каждый из вложенных виджетов `LinearLayout` о том, какие размеры ему следует иметь. Эти виджеты, в свою очередь, опрашивают содержащиеся в них кнопки (`Button`) или виджеты `EditText` относительно их размеров. Каждый дочерний элемент сообщает об оптимальном для себя размере родительскому элементу. Затем родительские элементы складывают данные, полученные от дочерних элементов, плюс все отступы, которые передает им родительский элемент, и сообщают сумму самому верхнему `LinearLayout`.

Поскольку фреймворк должен гарантировать в ходе этого процесса определенные поведения для всех видов `View`, метод `measure` является финальным и не может быть переопределен. Вместо переопределения `measure` вызывает метод `onMeasure`, который виджеты могут переопределять, чтобы получать для себя необходимое пространство.

Аргументы метода `onMeasure` описывают пространство, которое готов выделить родительский элемент: указывают высоту и ширину. Эти величины выражаются в пикселах. Фреймворк предполагает, что ни один вид не может быть по размеру меньше 0 и больше 2^{30} пикселей, поэтому использует старшие двоичные разряды переданного параметра `int` для кодирования *режима спецификации измерений* (`measurement specification mode`). Ситуация такова, как если бы `onMeasure` вызывался с четырьмя аргументами: режимом спецификации ширины, шириной, режимом

спецификации высоты и высотой. Не пытайтесь сами выполнять смещение битов для разделения пар аргументов! Вместо этого следует использовать статические методы `MeasureSpec.getMode` и `MeasureSpec.getSize`.

Режимы спецификации описывают, как контейнерный вид требует от дочернего вида интерпретировать размеры, переданные ему. Таких режимов три:

- `MeasureSpec.EXACTLY` — контейнерный вид, выполняющий вызов, уже определил точный размер дочернего вида;
- `MeasureSpec.AT_MOST` — контейнерный вид, выполняющий вызов, задал максимальное значение для этого параметра, но дочерний вид может запросить и меньше места;
- `MeasureSpec.UNSPECIFIED` — контейнерный вид, выполняющий вызов, не задал дочернему виду никаких ограничений; дочерний вид может запросить любой требуемый размер.

Виджет всегда должен сам сообщить своему родительскому элементу в дереве видов, сколько ему (виджету) требуется места. Это делается путем вызова метода `setMeasuredDimensions`, задающего значения высоты и ширины. Родительский элемент может позже получить данные об этих свойствах при помощи методов `getMeasuredHeight` и `getMeasuredWidth`. Если ваша реализация переопределяет `onMeasure`, но не вызывает `setMeasuredDimensions`, то метод `measure` не завершится нормально, а выдаст исключение `IllegalStateException`.

Стандартная реализация `onMeasure`, наследуемая от `View`, вызывает `setMeasuredDimensions` с одним из двух значений для каждого направления. Если родительский узел задает режим `MeasureSpec.UNSPECIFIED`, то метод `setMeasuredDimensions`, относящийся к дочернему элементу, использует стандартный размер вида. В таком качестве применяется значение, получаемое или от `getSuggestedMinimumWidth`, или от `getSuggestedMinimumHeight`. Если родительский элемент использует один из двух других режимов спецификации, то стандартная реализация применяет размер, предложенный родительским элементом. Это очень целесообразная стратегия, и она позволяет типичной реализации виджета полностью прорабатывать весь этап измерений, просто задавая значения, возвращенные методами `getSuggestedMinimumWidth` и `getSuggestedMinimumHeight`.

На самом деле виджет может и не получить того количества пространства, которое запрашивает. Рассмотрим вид, ширина которого равна 100 пикселям и который имеет три дочерних элемента. Пожалуй, совершенно очевидно, как родительский элемент должен расположить свои дочерние элементы, если суммарная ширина в пикселях, запрошенная всеми дочерними элементами, составляет 100 или меньше. Но если, например, каждый из дочерних элементов запросит по 50 пикселей в ширину, родительский вид не сможет удовлетворить все три запроса.

Контейнерный вид полностью контролирует расположение своих дочерних элементов. В вышеописанных обстоятельствах он может «поступить справедливо» и выделить каждому дочернему элементу по 33 % доступной ширины. С тем же успехом он может отдать 50 пикселей самому левому виджету и по 50 — двум остальным. И наконец, он может отвести все 100 пикселей единственному виджету,

а двум остальным ничего не оставить. Но независимо от метода именно родительский элемент определяет размер и положение ограничивающей рамки для каждого дочернего элемента.

Другой вариант того, как контейнерный вид управляет пространством, отводимым каждому виджету, понятен из примера 8.1. Виджет, показанный в этом примере, является минималистичным и всегда запрашивает столько пространства, сколько ему требуется, независимо от того, сколько места ему предлагается (в отличие от стандартной реализации). Такой стратегией удобно пользоваться при работе с виджетами, которые будут добавляться к контейнерам из стандартного инструментария, в частности к контейнеру `LinearLayout`. Такие контейнеры реализуют свойство *притяжения* (*gravity*). Гравитация — это свойство, используемое в некоторых видах для указания выравнивания своих подэлементов. Когда вы начнете работать с одним из этих контейнеров, вас, возможно, удивит одно обстоятельство. Оказывается, что по умолчанию отрисовывается только первый из созданных вами виджетов! Чтобы исправить это, можно либо изменить свойство притяжения на `Gravity.FILL` при помощи метода `setGravity`, либо заставить виджеты «настаивать» на получении именно такого количества пространства, которое им требуется.

Необходимо также отметить, что контейнерный вид может несколько раз вызывать метод `measure` дочернего элемента за один этап измерений. При реализации `onMeasure` правильно сделанный контейнерный вид, пытающийся построить горизонтальный ряд виджетов, может, например, вызвать метод `measure` каждого дочернего виджета в режиме `MEASURE_SPEC.UNSPECIFIED` и шириной 0, чтобы определить, какой размер «предпочтет» тот или иной виджет. Собрав, таким образом, предпочтительные значения ширины для всех своих дочерних элементов, он может сравнить сумму этих значений с шириной, доступной в данной ситуации (эта ширина указывалась в вызове, посланном от родительского элемента к методу `measure`). Теперь родительский элемент снова может вызвать метод `measure` каждого дочернего виджета, но на этот раз — уже в режиме `MeasureSpec.AT_MOST` с указанием ширины, которая пропорционально лучше всего подходит каждому виджету с учетом имеющегося пространства. Поскольку `measure` можно вызывать несколько раз, реализация `onMeasure` должна быть идемпотентной и не менять состояния приложения.



Действие называется идемпотентным, если эффект от его однократного применения такой же, как и от многократного применения. Например, выражение $x = 3$ является идемпотентным, так как независимо от того, сколько раз вы его примените, x будет равен 3. С другой стороны, выражение $x = x + 1$ не является идемпотентным, так как значение x зависит от того, сколько раз выполняется выражение.

Понятно, что реализация метода `onMeasure` с контейнерным видом обычно довольно сложна. `ViewGroup`, суперкласс всех контейнерных видов, не дает стандартной реализации этого метода. В каждом контейнерном виде фреймворка пользовательских интерфейсов Android такая реализация — собственная. Если вы подумываете реализовать контейнерный вид, можно выбрать один из таких методов в качестве

стандартного и ориентироваться на него. Если же, напротив, вам придется выполнять измерения на ходу, то, скорее всего, вам нужно будет вызывать метод `measure` для каждого дочернего элемента, а также испробовать на практике вспомогательные методы `ViewGroup: measureChild`, `measureChildren` и `measureChildWithMargins`.

При завершении этапа измерений контейнерный вид, как и любой другой виджет, должен сообщить о том, сколько места ему требуется. Это делается путем вызова `setMeasuredDimensions`.

Упорядочение

После того как все контейнерные виды в дереве видов смогут «договориться» о размерах своих дочерних элементов, фреймворк приступает ко второму этапу компоновки, который заключается в упорядочении дочерних элементов. Опять же если вы не реализуете собственный контейнерный вид, то вам, вероятно, никогда не придется работать и с собственным кодом для упорядочения элементов. В этом пункте описывается базовый процесс упорядочения, чтобы вы могли лучше понять, как он, возможно, повлияет на ваши виджеты. Стандартный метод, реализованный в классе `View`, будет работать с типичными «листовыми» виджетами, как это показано в примере 8.1.

Поскольку относящийся к виду метод `onMeasure` можно вызывать несколько раз, фреймворк должен использовать иной метод, чтобы сигнализировать, что данный этап измерений завершен и что контейнерные виды должны окончательно зафиксировать положение своих дочерних элементов. Как и этап измерений, этап упорядочения реализуется при помощи двух методов. Фреймворк активирует финальный метод `layout` в верхней точке дерева видов. Метод `layout` обрабатывает все виды по общему принципу, а потом вызывает метод `onLayout`. Пользовательские виджеты переопределяют метод `onLayout`, чтобы реализовывать собственные поведения. Пользовательская реализация `onLayout` должна как минимум рассчитать размеры рабочего прямоугольника, который будет предоставляться для каждого дочернего элемента при отрисовке, и, в свою очередь, активировать метод `layout` для каждого дочернего элемента (ведь этот элемент сам по себе может быть родительским для других виджетов). Этот процесс может быть довольно сложен. Если вашему виджету требуется упорядочивать дочерние виды, попробуйте взять в качестве основы для него уже имеющийся контейнер, например `LinearLayout` или `RelativeLayout`.

Еще раз отметим, что виджет не обязательно получает столько пространства, сколько запрашивает. Он должен быть способен отрисовываться независимо от того, сколько именно пространства ему отводится. Если он пытается отрисовываться вне того пространства, которое ему выделил родительский элемент, то рисунок будет обрезаться прямоугольником отсечения (`clip rectangle`). О нем мы подробнее поговорим ниже в этой главе. Чтобы обеспечить филигранный контроль (то есть, например, чтобы виджет заполнял ровно столько пространства, сколько ему выделено), нужно либо реализовать `onLayout` и записать параметры отведенного пространства, либо следить за прямоугольником отсечения `Canvas`, аргументом `onDraw`.

Рисование с применением Canvas (холста)

Теперь, когда мы изучили, как виджеты получают экранное пространство, на котором отрисовываются, мы можем написать несколько виджетов, в которых осуществляется рисование.

Фреймворк пользовательского интерфейса Android обрабатывает операции отрисовки таким способом, который уже должен показаться вам знакомым — после того как мы поговорили об измерениях и упорядочении. Когда какой-либо компонент приложения определяет, что изображение, находящееся на экране в данный момент, устарело, поскольку то или иное состояние изменилось, этот компонент вызывает метод `invalidate` класса `View`. В результате такого вызова в общую очередь событий добавляется событие перерисовки (`redraw event`).

Когда рано или поздно программа переходит к обработке этого события, фреймворк вызывает метод `draw` в верхней точке дерева видов. На этот раз вызов распространяется в прямом порядке, то есть каждый вид сначала отрисовывается, а потом делает вызовы к своим дочерним видам. Это означает, что «листовые» виды рисуются после своих родительских видов, которые, в свою очередь, рисуются после своих родительских элементов. Получается, что виды, находящиеся выше по дереву, рисуются раньше, чем те, которые находятся ближе к корню дерева. Такой механизм иногда называется «алгоритм живописца».

Метод `draw` вызывает метод `onDraw`, переопределяемый каждым подклассом для реализации собственного варианта отображения. Когда вызван метод `onDraw` вашего виджета, этот виджет должен отобразиться в соответствии с актуальным состоянием приложения и вернуть управление. Кстати, оказывается, что ни `View.draw`, ни `ViewGroup.dispatchDraw` (отвечающие за обход вершин дерева) не являются финальными. Но их переопределение остается на ваш страх и риск!

Чтобы не приходилось чрезмерно много всего рисовать, фреймворк пользовательского интерфейса Android сохраняет определенную информацию о состоянии вида. Данная информация называется *прямоугольником отсечения*. Он является одной из ключевых концепций фреймворка и входит в состав информации о состоянии, которая передается в вызовах к методам графического отображения компонента. Расположение и размер этого прямоугольника можно получить и откорректировать при помощи соответствующих методов холста. Данный прямоугольник действует как трафарет, через который компонент выполняет все рисование: это означает, что компонент может рисовать только на тех участках холста, которые видны через прямоугольник отсечения. Правильно задавая размер, контуры и положение прикладываемого прямоугольника отсечения, фреймворк не позволяет компоненту ничего рисовать вне отведенных ему границ либо перерисовывать области, которые уже отрисованы правильно.

Еще один важный инструмент оптимизации появился в Android API 7, `Eclair`. Непрозрачному виду, то есть такому, который полностью заполняет занимаемую им прямоугольную область отсечения непрозрачными объектами, следует переопределять относящийся к виду метод `isOpaque`, чтобы он возвращал булево значение `true`. В таком случае вид словно подсказывает алгоритму отрисовки, что он не должен отрисовывать никакие виды, которые заслоняются виджетом. Даже в умеренно сложном дереве видов такой прием позволяет уменьшить количество отрисовок

конкретного пиксела с четырех-пяти до одной. Этот способ поможет, например, добиться плавной прокрутки спискового вида, который до этого притормаживал.

Прежде чем перейти к специфическим проблемам рисования, снова рассмотрим эту тему в контексте паттерна однопоточной архитектуры MVC, применяемого в Android. Здесь существует два важных правила.

- Код рисования должен содержаться в методе `onDraw`. При вызове `onDraw` виджет должен полностью отрисовываться, отражая состояние программы.
- При активации `onDraw` виджет должен отрисовываться максимально быстро. В ходе запроса к `onDraw` нет времени выполнять сложный запрос к базе данных или определять состояние какой-нибудь удаленной сетевой службы. Все состояние, которое вам необходимо отрисовать, должно быть кэшировано и готово к использованию прямо во время рисования. Долговременные задачи должны выполняться в отдельном потоке и использовать один из механизмов, описанных в подразделе «Продвинутые способы подключения: фокус и точность» раздела «Подключение контроллера» главы 6. Информация о состоянии модели, кэшируемая в виде, иногда называется *моделью вида*.

При рисовании во фреймворке пользовательского интерфейса Android применяются четыре основных класса. Если вы собираетесь реализовывать пользовательские виджеты и выполнять собственные операции отрисовки, вам нужно досконально познакомиться с этими классами.

- `Canvas` (подкласс `android.graphics.Canvas`) — у `Canvas` (с англ. — «холст»), как ни странно, нет адекватного аналога в реальном мире. Его можно считать сложным мольбертом, который способен ориентировать, сгибать и даже комкать бумагу, на которой вы рисуете, причем самыми интересными способами. Он сохраняет информацию о прямоугольнике отсечения, том трафарете, через который вы рисуете. Кроме того, по мере рисования холст может масштабировать появляющиеся на нем изображения, подобно фотоувеличителю. Он может производить и другие преобразования, для которых сложнее подобрать аналоги из реальной жизни: например, преобразовывать цвета и рисовать текст вдоль заданных линий и т. д.
- `Paint` (подкласс `android.graphics.Paint`) — это средство, при помощи которого вы рисуете. Оно отвечает за цвет, прозрачность и размер кисти при рисовании объектов на холсте. Кроме того, при рисовании текста данный инструмент отвечает за подбор гарнитуры, размера и оформления шрифта.
- `Bitmap` (подкласс `android.graphics.Bitmap`) — это «бумага», на которой вы рисуете. Она хранит пикселы нарисованного изображения.
- `Drawable` (почти подкласс `android.graphics.drawable.Drawable`) — этот подкласс описывает то, что вы хотите нарисовать: прямоугольник или изображение. Хотя не все, что вы рисуете, относится к `Drawable` (например, текст), многие объекты, особенно сравнительно сложные, — относятся.

Все рисование в примере 8.1 производится только при помощи `Canvas`, передаваемого в качестве параметра к `onDraw`. Чтобы получилось что-нибудь более интересное, нам как минимум понадобится `Paint`. Он обеспечивает контроль над цветом и прозрачностью (альфа-каналом) графики, которая рисуется с применением этого подкласса. При использовании вместе с методами рисования текста этот подкласс управляет гарнитурой, размерами и оформлением текста. `Paint` предлагает также

немало других возможностей. Некоторые из них описаны в разделе «Украшения» данной главы. Но пример 8.2 уже позволяет приступить к работе с `Paint`. Здесь задаются два из многих параметров элементов управления `Paint` (цвет и ширина линии). После этого рисуется жирная вертикальная линия, а за ней — серия горизонтальных линий. Альфа-значение (играющее ту же роль, что и четвертое значение в системе цветов RGB для Веба) снижается в каждой последующей зеленой линии, и она получается более прозрачной, чем предыдущая. Советуем подробнее ознакомиться с документацией по этому классу и изучить другие связанные с ним полезные атрибуты.

Пример 8.2. Использование `Paint`

```
@Override
protected void onDraw(Canvas canvas) {
    canvas.drawColor(Color.WHITE);

    Paint paint = new Paint();

    canvas.drawLine(33, 0, 33, 100, paint);

    paint.setColor(Color.RED);
    paint.setStrokeWidth(10);
    canvas.drawLine(56, 0, 56, 100, paint);

    paint.setColor(Color.GREEN);
    paint.setStrokeWidth(5);

    for (int y = 30, alpha = 255; alpha > 2; alpha >= 1, y += 10) {
        paint.setAlpha(alpha);
        canvas.drawLine(0, y, 100, y, paint);
    }
}
```

Картинка, создаваемая с помощью данного кода, показана на рис. 8.1.

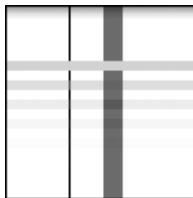


Рис. 8.1. Вывод в программе `Paint`

Теперь, добавив к нашему арсеналу `Paint`, мы сможем разобраться и с большинством других инструментов, необходимых для того, чтобы нарисовать хороший виджет. Например, код из примера 8.3 — это виджет, с которым мы работали в примере 6.7. Он все еще не так сложен, но обладает всеми компонентами полнофункционального виджета. Он работает с компоновкой, использует подсвечивание (показывающее, находится ли виджет в фокусе, то есть работает ли пользователь в данный момент именно с ним) и отражает состояние модели, к которой прикреплен. Виджет рисует серию точек, информация о которых сохраняется в закрытом

массиве. Каждая точка указывает собственные координаты по осям X и Y , а также собственные диаметр и цвет. Функция `onDraw` устанавливает цвет в `Paint` для каждой точки по отдельности и использует другие параметры для определения окружности, которая рисуется методом `drawCircle`, относящимся к холсту.

Пример 8.3. Виджет с точками

```
package com.oreilly.android.intro.view;

import android.content.Context;

import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Paint.Style;

import android.view.View;

import com.oreilly.android.intro.model.Dot;
import com.oreilly.android.intro.model.Dots;

public class DotView extends View {
    private final Dots dots;

    /**
     * @param context остальная часть приложения
     * @param dots точки, которые мы рисуем
     */
    public DotView(Context context, Dots dots) {
        super(context);
        this.dots = dots;
        setMinimumWidth(180);
        setMinimumHeight(200);
        setFocusable(true);
    }

    /** @see android.view.View#onMeasure(int, int) */
    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        setMeasuredDimension(
            getSuggestedMinimumWidth(),
            getSuggestedMinimumHeight());
    }

    /** @see android.view.View#onDraw(android.graphics.Canvas) */
    @Override protected void onDraw(Canvas canvas) {
        canvas.drawColor(Color.WHITE);

        Paint paint = new Paint();
        paint.setStyle(Style.STROKE);
        paint.setColor(hasFocus() ? Color.BLUE : Color.GRAY);
        canvas.drawRect(0, 0, getWidth() - 1, getHeight() - 1, paint);
    }
}
```

```

        paint.setStyle(Style.FILL);
        for (Dot dot : dots.getDots()) {
            paint.setColor(dot.getColor());
            canvas.drawCircle(
                dot.getX(),
                dot.getY(),
                dot.getDiameter(),
                paint);
        }
    }
}

```

Как и в случае с `Paint`, у нас теперь как раз достаточно места, чтобы приступить к исследованиям методов `Canvas`. Функции можно условно разделить на две группы, и обе эти группы стоит обсудить отдельно.

Рисование текста

Среди наиболее важных методов `Canvas` — те, которые используются для рисования текста. Хотя часть функций `Canvas` дублируется в других местах, это не касается функций, связанных с отображением текста. Чтобы поместить текст в виджет, нужно использовать `Canvas` (или сделать подкласс от другого виджета, применяющего его).

`Canvas` предоставляет несколько методов для отображения текста, которые позволяют вам с разной степенью гибкости работать над размещением каждого символа в тексте. Методы представлены попарно: один принимает строку `String`, а другой — массив `char[]`. В некоторых случаях применяются дополнительные вспомогательные методы. Например, простейший способ нарисовать текст — это передать координаты `x` и `y`, в которых начинается текст, и `Paint`, где указываются гарнитура, цвет шрифта и другие атрибуты (пример 8.4).

Пример 8.4. Пара методов для рисования текста

```

public void drawText(String text, float x, float y, Paint paint)
public void drawText(char[] text, int index, int count, float x,
                    float y, Paint paint)

```

В то время как первый метод передает текст через единственный параметр `String`, второй метод использует три параметра: массив `char`, индекс, указывающий первый символ из этого массива, с которого нужно начать рисовать текст, и общее количество символов текста, которые следует отобразить. Если вам требуется что-то более сложное, чем обычный горизонтальный текст, его можно расположить вдоль геометрического контура или даже разместить каждый отдельный символ там, где вы хотите. В примере 8.5 приводится метод `onDraw`, демонстрирующий применение трех различных методов отображения текста. Вывод показан на рис. 8.2.

Пример 8.5. Три способа рисования текста

```

@Override
protected void onDraw(Canvas canvas) {
    canvas.drawColor(Color.WHITE);

    Paint paint = new Paint();

```

```

paint.setColor(Color.RED);
canvas.drawText("Android", 25, 30, paint);

Path path = new Path();
path.addArc(new RectF(10, 50, 90, 200), 240, 90);
paint.setColor(Color.CYAN);
canvas.drawTextOnPath("Android", path, 0, 0, paint);

float[] pos = new float[] {
    20, 80,
    29, 83,
    36, 80,
    46, 83,
    52, 80,
    62, 83,
    68, 80
};
paint.setColor(Color.GREEN);
canvas.drawPosText("Android", pos, paint);
}

```



Рис. 8.2. Вывод трех вариантов рисования текста

Как видите, простейшая из пар — `drawText` — просто рисует текст из точки с заданными координатами. С другой стороны, в случае с `drawTextOnPath` текст можно рисовать вдоль любой линии `Path`. Линия, используемая в примере, — простая дуга. С тем же успехом это может быть прямая линия или кривая Безье.

Если и возможностей `drawTextOnPath` оказывается недостаточно, `Canvas` предлагает `drawPosText` — метод, позволяющий указать точное положение каждого символа в тексте. Обратите внимание — при задании положения символов попеременно записываются элементы массива `float`, передаваемого в качестве второго аргумента: $x_1, y_1, x_2, y_2, \dots$

Матричные преобразования

Вторая интересная группа методов `Canvas` — матричные преобразования `Matrix` и их вспомогательные методы `rotate`, `scale` и `skew`. Эти методы должен сразу узнать любой специалист, которому приходилось работать с трехмерной графикой в других

окружениях. Такие методы позволяют отображать отдельно взятый рисунок так, как если бы зритель двигался относительно рисуемых объектов.

Небольшое приложение для примера 8.6 демонстрирует возможности координатных преобразований, присущих Canvas.

Пример 8.6. Использование преобразований при работе с холстом

```
import android.app.Activity;

import android.content.Context;

import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Rect;

import android.os.Bundle;

import android.view.View;

import android.widget.LinearLayout;

public class TransformationalActivity extends Activity {

    private interface Transformation {
        void transform(Canvas canvas);
        String describe();
    }

    private static class TransformedViewWidget extends View { ❶
        private final Transformation transformation;

        public TransformedViewWidget(Context context, Transformation xform) {
            super(context);

            transformation = xform; ❷

            setMinimumWidth(160);
            setMinimumHeight(105);
        }

        @Override
        protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
            setMeasuredDimension(
                getSuggestedMinimumWidth(),
                getSuggestedMinimumHeight());
        }

        @Override
        protected void onDraw(Canvas canvas) { ❸
            canvas.drawColor(Color.WHITE);
```

```

    Paint paint = new Paint();

    canvas.save(); ❷
    transformation.transform(canvas); ❸

    paint.setTextSize(12);
    paint.setColor(Color.GREEN);
    canvas.drawText("Hello", 40, 55, paint);

    paint.setTextSize(16);
    paint.setColor(Color.RED);
    canvas.drawText("Android", 35, 65, paint);

    canvas.restore(); ❹

    paint.setColor(Color.BLACK);
    paint.setStyle(Paint.Style.STROKE);
    Rect r = canvas.getClipBounds();
    canvas.drawRect(r, paint);

    paint.setTextSize(10);
    paint.setColor(Color.BLUE);
    canvas.drawText(transformation.describe(), 5, 100, paint);
}

@Override
public void onCreate(Bundle savedInstanceState) { ❺
    super.onCreate(savedInstanceState);
    setContentView(R.layout.transformed);

    LinearLayout v1 = (LinearLayout) findViewById(R.id.v_left); ❻
    v1.addView(new TransformedViewWidget( ❼
        this,
        new Transformation() { ❶
            @Override public String describe() { return "identity"; }
            @Override public void transform(Canvas canvas) { }
        } ));
    v1.addView(new TransformedViewWidget( ❼
        this,
        new Transformation() { ❶
            @Override public String describe() { return "rotate(-30)"; }
            @Override public void transform(Canvas canvas) {
                canvas.rotate(-30.0F);
            } }));
    v1.addView(new TransformedViewWidget( ❼
        this,
        new Transformation() { ❶
            @Override public String describe() { return "scale(.5,.8)"; }
            @Override public void transform(Canvas canvas) {

```

```

        canvas.scale(0.5F, .8F);
    } }));
v1.addView(new TransformedViewWidget( 9
    this, 10
    new Transformation() {
        @Override public String describe() { return "skew(.1,.3)"; }
        @Override public void transform(Canvas canvas) {
            canvas.skew(0.1F, 0.3F);
        } }));

LinearLayout v2 = (LinearLayout) findViewById(R.id.v_right); 11
v2.addView(new TransformedViewWidget( 12
    this,
    new Transformation() { 10
        @Override public String describe() {
            return "translate(30,10)"; }
        @Override public void transform(Canvas canvas) {
            canvas.translate(30.0F, 10.0F);
        } }));
v2.addView(new TransformedViewWidget( 12
    this,
    new Transformation() { 10
        @Override public String describe() {
            return "translate(110,-20),rotate(85)";
        }
        @Override public void transform(Canvas canvas) {
            canvas.translate(110.0F, -20.0F);
            canvas.rotate(85.0F);
        } }));
v2.addView(new TransformedViewWidget( 12
    this,
    new Transformation() { 10
        @Override public String describe() {
            return "translate(-50,-20),scale(2,1.2)";
        }
        @Override public void transform(Canvas canvas) {
            canvas.translate(-50.0F, -20.0F);
            canvas.scale(2F, 1.2F);
        } }));
v2.addView(new TransformedViewWidget( 12
    this,
    new Transformation() { 10
        @Override public String describe() { return "complex"; }
        @Override public void transform(Canvas canvas) {
            canvas.translate(-100.0F, -100.0F);
            canvas.scale(2.5F, 2F);
            canvas.skew(0.1F, 0.3F);
        } }));
    }
}

```

Результат выполнения этого пространного кода показан на рис. 8.3.









 identity	 translate(30,10)
 rotate(-30)	 translate(110,-20),rotate(85)
 scale(5,.8)	 translate(-50,-20),scale(2,1.2)
 skew(1,.3)	 complex

Рис. 8.3. Преобразованные виды

Некоторые фрагменты кода отмечены цифрами.

- ❶ Это определение нового виджета `TransformedViewWidget`.
- ❷ Из второго аргумента конструктора код получает информацию о том, какое именно преобразование следует произвести.
- ❸ Это метод `onDraw` класса `TransformedViewWidget`.
- ❹ Код помещает актуальное состояние рисования в стек, используя `save` перед осуществлением каких-либо преобразований.
- ❺ Выполнение преобразования, переданного в качестве второго аргумента конструктора.
- ❻ Восстановление прежнего состояния, сохраненного на этапе 4, приготовление к рисованию ограничивающей рамки и метки.
- ❼ Это метод `onCreate`, относящийся к активности `Activity`.
- ❽ Создание контейнерного вида для левого столбца с виджетами.
- ❾ Здесь происходит инстанцирование `TransformedViewWidget`, добавляемых в левый столбец.
- ❿ Создание преобразования, входящего в список параметров, которые передаются конструктору `TransformedViewWidget`.
- ⓫ Создание контейнерного вида для виджетов, входящих в правый столбец.
- ⓬ Инстанцирование `TransformedViewWidget`, добавляемых в правый столбец.

Это небольшое приложение подсказывает нам ряд новых идей. Что касается видов и виджетов, данное приложение определяет единственный виджет, `TransformedViewWidget`, и создает восемь его экземпляров. Что касается компоновки, приложение создает два вида, которые называются `v1` и `v2`, получая их параметры из ресурсов. Затем программа добавляет по четыре экземпляра `TransformedViewWidget` к каждому виду `LinearLayout`. На этом примере видно, как в приложении можно комбинировать динамические виды и виды, основанные на ресурсах. Обратите внимание: создание контейнерных видов, а также создание новых виджетов происходит в методе `onCreate`, относящемся к активности.

Кроме того, данное приложение обеспечивает значительную гибкость виджета, поскольку решаемые задачи умело распределяются между самим виджетом и родительским видом. Рисование нескольких простых объектов определено в методе `onDraw` класса `TransformedViewWidget`:

- белый фон;
- слово `Hello` зеленым шрифтом (12-й кегль);
- слово `Android` красным шрифтом (16-й кегль);
- черная рамка;
- голубая метка.

В большинстве случаев фон вида задается при помощи метода `setBackground`, относящегося к `View`. В результате вид отрисовывает фон автоматически, как и требуется. Поскольку виджет в этом примере был создан для того, чтобы явно продемонстрировать все этапы, необходимые для полного отображения данного виджета (вскоре мы это увидим), мы приобретаем удобный дополнительный функционал, воспользовавшись встроенным фоном вида.

В центре всего этого метод `onDraw` осуществляет преобразование, указываемое вызывающей стороной. Приложение определяет собственный интерфейс, названный `Transformation`, а конструктор `TransformedViewWidget` принимает `Transformation` в качестве параметра. Рассмотрим детально, как именно задается преобразование.

Сначала изучим, как `onDraw` сохраняет собственный текст, также допуская при этом преобразование. В данном примере нам необходимо убедиться, что и рамка и метка рисуются поверх любых других элементов, отрисовываемых виджетом, даже если рамка или метка накладываются на эти элементы. Мы же не хотим, чтобы преобразование затронуло рамку или метку.

Удобно, что `Canvas` поддерживает внутренний стек, в который можно записать, а потом и восстановить матрицу преобразования, прямоугольник отсечения, а также любые другие элементы изменяемого состояния `Canvas`. Пользуясь преимуществами этого стека, `onDraw` вызывает `Canvas.save` для сохранения своего состояния перед преобразованием, а `Canvas.restore` — после, для восстановления сохраненного состояния.

Оставшаяся часть приложения контролирует преобразование, применяемое к каждому из восьми экземпляров `TransformedViewWidget`. Каждый новый экземпляр виджета создается с собственным анонимным экземпляром `Transformation`. К изображению в области, помеченной `identity` (дословно — «сущность»), никакого преобразования не применяется. Остальные семь областей помечены названиями тех преобразований, которые в них демонстрируются.

Базовые методы преобразования в Canvas — это `setMatrix` и `concatMatrix`. Два данных метода позволяют строить всевозможные преобразования. Методы, показанные в примере, — `translate`, `rotate`, `scale` и `skew` — являются вспомогательными и образуют на основе конкретных специфических матриц, с каждой из которых связаны собственные ограничения, актуальное состояние Canvas. Наконец, метод `getMatrix` дает возможность восстанавливать динамически созданную матрицу для последующего использования.

Хотя на первый взгляд это и неочевидно, такие функции преобразования могут быть исключительно полезны. Они позволяют изменить ракурс, под которым мы видим окно приложения, относительно трехмерного объекта! Не нужно особого воображения, чтобы понять, что сцена в квадрате `scale(.5,.8)` — та же, что и в квадрате `identity`, но первая сцена показана чуть издалека. Приложим еще немного воображения — и изображение в квадрате `skew(.1,.3)` может показаться неизменной картинкой, но на этот раз показанной сверху и чуть сбоку. Масштабирование или преобразование объекта может создать у пользователя впечатление, будто этот объект сдвинулся, а при наклоне и вращении — будто объект повернулся.

Если представить, что такие функции преобразования могут применяться к чему угодно, рисуемому на холсте (к линиям, тексту и даже изображениям), то их важность в приложениях становится еще более очевидной. Вид, демонстрирующий эскизы фотографий, можно достаточно просто реализовать как вид, уменьшающий любое изображение до 10 % от его актуального размера (хотя это и не оптимальный вариант). Приложение, отображающее панораму, которую вы видите слева, ведя машину по оживленной улице, может быть реализовано в том числе путем масштабирования и наклона небольшого количества изображений. Это особенно важно, поскольку преобразования могут быть очень быстрыми. Кроме того, преобразования хорошо приспособлены к работе с аппаратной оптимизацией.

Отрисовываемые объекты

Отрисовываемый объект (`Drawable`) — это объект, знающий, как отобразить себя на холсте Canvas. Поскольку во время отображения `Drawable` поддается полному контролю, даже самый сложный процесс отображения можно «упаковать» так, что пользоваться им не составит никакого труда.

В примерах 8.7 и 8.8 показаны изменения, которые необходимо сделать, чтобы реализовать пример, показанный на рис. 8.3, при помощи `Drawable`. Код, рисующий красный и зеленый текст, путем рефакторинга был преобразован в класс `HelloAndroidTextDrawable`, используемый методом `onDraw` при отображении (этот метод относится к виджету).

Пример 8.7. Использование `TextDrawable`

```
private static class HelloAndroidTextDrawable extends Drawable {
    private ColorFilter filter;
    private int opacity;

    public HelloAndroidTextDrawable() {}

    @Override
    public void draw(Canvas canvas) {
        Paint paint = new Paint();
```

```

        paint.setColorFilter(filter);
        paint.setAlpha(opacity);

        paint.setTextSize(12);
        paint.setColor(Color.GREEN);
        canvas.drawText("Hello", 40, 55, paint);

        paint.setTextSize(16);
        paint.setColor(Color.RED);
        canvas.drawText("Android", 35, 65, paint);
    }

    @Override
    public int getOpacity() { return PixelFormat.TRANSLUCENT; }

    @Override
    public void setAlpha(int alpha) { }

    @Override
    public void setColorFilter(ColorFilter cf) { }
}

```

Для использования новой реализации Drawable потребуется внести в метод `onDraw` из нашего примера всего несколько небольших изменений.

Пример 8.8. Использование виджета Drawable

```

package com.oreilly.android.intro.widget;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Rect;
import android.graphics.drawable.Drawable;
import android.view.View;

/** Виджет, отображающий отрисовываемый объект с преобразованием. */
public class TransformedViewWidget extends View {

    /** преобразование */
    public interface Transformation {
        /** @param canvas */
        void transform(Canvas canvas);
        /** @return text описание преобразования */
        String describe();
    }

    private final Transformation transformation;
    private final Drawable drawable;

    /**
     * отображение переданного отрисовываемого объекта в преобразованном виде
     */
}

```

```

    * @param context контекст приложения
    * @param draw объект, который требуется нарисовать при преобразовании
    * @param xform преобразование
    */
    public TransformedViewWidget(
        Context context,
        Drawable draw,
        Transformation xform)
    {
        super(context);

        drawable = draw;
        transformation = xform;

        setMinimumWidth(160);
        setMinimumHeight(135);
    }

    /** @see android.view.View#onMeasure(int, int) */
    @Override
    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        setMeasuredDimension(
            getSuggestedMinimumWidth(),
            getSuggestedMinimumHeight());
    }

    /** @see android.view.View#onDraw(android.graphics.Canvas) */
    @Override
    protected void onDraw(Canvas canvas) {
        canvas.drawColor(Color.WHITE);

        canvas.save();

        transformation.transform(canvas);
        drawable.draw(canvas);
        canvas.restore();

        Paint paint = new Paint();
        paint.setColor(Color.BLACK);
        paint.setStyle(Paint.Style.STROKE);
        Rect r = canvas.getClipBounds();
        canvas.drawRect(r, paint);

        paint.setTextSize(10);
        paint.setColor(Color.BLUE);
        canvas.drawText(
            transformation.describe(),
            5,
            getMeasuredHeight() - 5,
            paint);
    }
}

```

В какой-то мере этот код уже позволяет оценить огромный потенциал Drawable. Такая реализация TransformedViewWidget может преобразовать любой Drawable независимо от того, что именно предполагается рисовать. Код уже не имеет жесткой привязки к вращению и масштабированию нашего исходного жестко закодированного текста. Его можно использовать многократно для преобразования как текста, взятого из предыдущего примера, так и фотографии, снятой на фотоаппарат (рис. 8.4). Рисунок даже можно преобразовать в Drawable-анимацию.

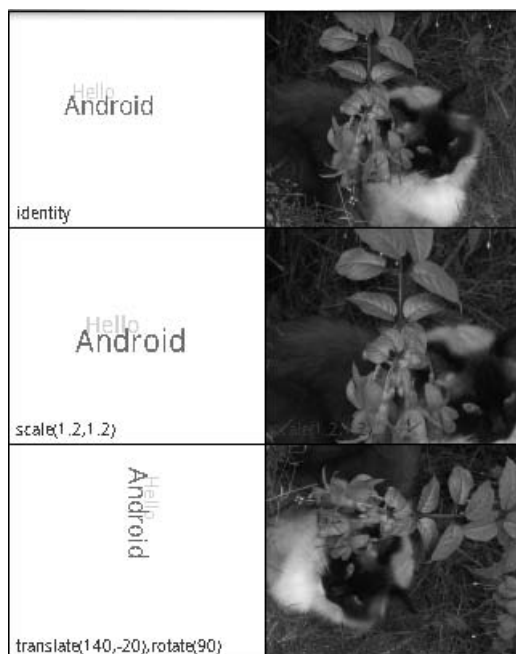


Рис. 8.4. Преобразованные виды с фотографиями

Drawable позволяют воплотить на практике такие сложные графические техники, как «девять прямоугольников»¹ или анимация. К тому же, поскольку Drawable полностью включают в себя процесс рендеринга, отрисовываемые объекты можно вкладывать друг в друга, чтобы разбивать сложные графические сущности на небольшие компоненты, пригодные для многократного использования.

Попробуем представить, как можно дополнить предыдущий пример, чтобы за минуту каждое из шести изображений на какое-то время «выцветало» и на его месте оставался просто белый квадрат. Разумеется, нужно просто изменить код из примера 8.8, добавив к нему эффект выцветания. Другой — гораздо более интересный — вариант реализации связан с написанием одного нового Drawable.

Конструктор этого нового отрисовываемого объекта (назовем такой объект FaderDrawable) будет принимать в качестве аргумента ссылку на целевой объект,

¹ По-английски называется nine-patch. Подробнее о технологии — http://habrahabr.ru/blogs/android_development/113623/#habracut. — *Примеч. пер.*

тот `Drawable`, который будет «выцветать» до белого квадрата. Кроме того, этот объект должен иметь какое-то представление о времени, например целое число — назовем его `t`, — приращение которого происходит по мере работы таймера. При каждом вызове метода `draw`, относящегося к объекту `FaderDrawable`, он сначала вызывает метод `draw` своей цели. Но потом он постепенно зарисовывает ту же целевую область белым цветом, используя значение `t` для определения прозрачности рисунка, как это показано в примере 8.2. Со временем `t` увеличивается, белый цвет становится все более насыщенным и целевой объект `Drawable` полностью исчезает из вида.

Такой гипотетический объект `FaderDrawable` демонстрирует несколько важных свойств отрисовываемых объектов. `FaderDrawable` по определению пригоден для многократного использования. Применяя его, можно «растворить» практически любой отрисовываемый объект. Кроме того, поскольку `FaderDrawable` дополняет `Drawable`, `FaderDrawable` можно использовать во всех тех местах, где мы бы применяли и его целевой объект, то есть `Drawable`, «выцветающий» до белого квадрата. Любой код, использующий `Drawable` в процессе отображения, может применять и `FaderDrawable` без изменений.

Разумеется, и сам `FaderDrawable` может быть заключен в другой объект. На самом деле можно достичь довольно сложных эффектов, построив цепь оболочек `Drawable`. В инструментарии Android предоставляются оболочки `Drawable`, поддерживающие такой способ работы, в том числе `ClipDrawable`, `RotateDrawable` и `ScaleDrawable`.

Может быть, на данном этапе вы уже мысленно переработали весь наш пользовательский интерфейс в виде совокупности отрисовываемых объектов. Но, хотя это и мощный инструмент, он не является панацеей. Есть несколько проблем, о которых не следует забывать, если вы собираетесь пользоваться `Drawable`.

Вы, наверное, заметили, что отрисовываемые объекты имеют немало общих функций с классом `View`: расположение, размеры, видимость и т. д. Не всегда бывает просто принять решение о том, когда `View` должен рисовать прямо на `Canvas`, когда — делегировать эту задачу своему подвиду, а когда — одному или нескольким отрисовываемым объектам. Есть даже класс `DrawableContainer`, позволяющий группировать несколько дочерних `Drawable` внутри родительского элемента. Можно строить деревья `Drawable`, аналогичные деревьям `View`, которые мы использовали до сих пор. Работая с фреймворком пользовательского интерфейса Android, мы просто признаем, что есть несколько способов сделать одно и то же.

Наиболее очевидная разница между двумя способами заключается в том, что отрисовываемые объекты не выполняют непосредственной реализации контроллера для обработки пользовательского ввода. Поэтому их нельзя считать виджетами в нашем понимании. Кроме того, `Drawables` не реализуют свойственный для `View` протокол измерения/компоновки. Как вы помните, такой протокол позволяет дочерним видам, находящимся внутри контейнерного вида, «договариваться» о компоновке компонентов в зависимости от изменения размеров вида. Когда отображаемому объекту требуется добавить, удалить или перегруппировать свои внутренние компоненты — это серьезный аргумент в пользу того, что в данном случае следует использовать полнофункциональный `View`, а не `Drawable`.

Вторая проблема, которую необходимо учитывать, состоит в том, что, поскольку объекты `Drawable` полностью заключают в себе процесс рисования, они не рисуются так, как объекты `String` или `Rect`. Например, не существует никаких методов `Canvas`, которые бы отображали `Drawable` в конкретных координатах. Возможно, вы задумаетесь, что лучше сделать, чтобы дважды отобразить одно и то же изображение: применить метод `View.onDraw`, который будет использовать два разных неизменяемых `Drawable`, или дважды задействовать один и тот же `Drawable`, переустановив его координаты после первого применения.

Но гораздо важнее, пожалуй, проблема более универсального характера. Причина, по которой воплощается сама идея использования `Drawable` по цепочке, заключается в том, что интерфейс `Drawable` не содержит никакой информации о внутренней реализации `Drawable`. Когда `Drawable` передается коду, код никоим образом не может узнать, что именно ему придется отобразить — простое изображение или сложную цепь эффектов — вращений, миганий, качаний. Разумеется, такая ситуация может быть как положительной, так и отрицательной.

Существенная часть процесса рисования происходит с сохранением состояния. Вы настраиваете `Paint`, области отсечения, `Canvas`, а потом рисуете. При совместной работе по цепочке `Drawable` нужно внимательно следить за тем, чтобы после изменения состояний `Drawable` между ними не возникало конфликтов. Проблема заключается в том, что при построении цепочки `Drawable` невозможно явно и сразу определить по типу объекта, какие конфликты могут возникнуть (ведь перед нами просто группа отрисовываемых объектов). Самое незначительное изменение может вызывать нежелательные последствия, которые плохо поддаются отладке.

Чтобы проиллюстрировать эту ситуацию, представим себе два класса-обертки `Drawable`, один из которых должен ужимать контент, а другой — поворачивать его на 90°. Если каждая операция реализуется путем присвоения матрице преобразований конкретного значения, то совместное использование этих классов может дать нежелательный эффект. Еще хуже ситуация, в которой *A* отлично оборачивает *B*, а *B* совсем не оборачивает *A*! Поэтому важно подробно описывать в документации, как именно работает данный `Drawable`.

Битовые карты

Битовая карта (`Bitmap`) — последний, четвертый основополагающий компонент для рисования. Кроме нее, в этот квартет входит то, что мы собираемся рисовать (`String`, `Rect` и т. д.), `Paint`, с помощью которого мы будем рисовать, и `Canvas`, на котором нужно рисовать. Сама битовая карта содержит биты. Как правило, вам не придется работать непосредственно с `Bitmap`, так как `Canvas`, сообщаемый в качестве аргумента к методу `onDraw`, уже несет в себе одну битовую карту.

Обычно `Bitmap` используется для кэширования рисунка, создавать который довольно долго, а изменять если и приходится, то нечасто. Представим себе, например, графический редактор, позволяющий пользователю рисовать на нескольких слоях одновременно. Слои действуют как прозрачные наложения, покрывающие основное изображение, и пользователь может по желанию включать и выключать те или иные

слои. Однако будет очень сложно заново рисовать тот или иной слой при каждом вызове `onDraw`. Быстрее было бы отображать весь рисунок при первом его появлении, со всеми видимыми слоями, а потом перерисовывать тот единственный слой, в который пользователь внес отображаемое изменение.

Реализация такого приложения может выглядеть как в примере 8.9.

Пример 8.9. Кэширование графики

```
private class CachingWidget extends View {
    private Bitmap cache;

    public CachingWidget(Context context) {
        super(context);
        setMinimumWidth(200);
        setMinimumHeight(200);
    }

    public void invalidateCache() {
        cache = null;
        invalidate();
    }

    @Override
    protected void onDraw(Canvas canvas) {
        if (null == cache) {
            cache = Bitmap.createBitmap(
                getMeasuredWidth(),
                getMeasuredHeight(),
                Bitmap.Config.ARGB_8888);

            drawCachedBitmap(new Canvas(cache));
        }

        canvas.drawBitmap(cache, 0, 0, new Paint());
    }

    // ...определение drawCachedBitmap
}
```

Как правило, этот виджет просто копирует кэшированную информацию `Bitmap`, `cache` в `Canvas`, передаваемый методу `onDraw`. Только если кэш будет помечен как устаревший путем вызова `invalidateCache`, будет вызываться `drawCachedBitmap` — та карта, которая и отобразит виджет. На практике `Bitmap` чаще всего используется для программного представления графического ресурса. `Resources.getDrawable` возвращает `BitmapDrawable`, если ресурс — это изображение.

Если совместить две эти идеи, оказывается, что при кэшировании изображения и заключении его в `Drawable` открывается еще одна интересная перспектива. Дело в том, что любая нарисованная информация допускает постобработку! Приложение, использующее все техники, представленные в этой главе, позволяет пользователю нарисовать в комнате мебель (создавая карту битов), а потом походить по этой комнате (при помощи матричных преобразований).



В версии Honeycomb архитектура рендеринга в Android претерпела существенные изменения. Эти изменения основаны на растущей мощности графических процессоров и задают новые правила, оптимизирующие способы отрисовки пользовательского интерфейса. Кэширование битовых карт с использованием нового графического конвейера может происходить даже медленнее, чем отрисовка их по мере необходимости. Прежде чем приступить к кэшированию битовых карт, попробуйте использовать `View.setLayerType`.

Украшения

Фреймворк пользовательского интерфейса Android — это не просто умный, хорошо подобранный инструментарий для написания графических пользовательских интерфейсов. Он еще и довольно красив. Разумеется, список инструментов, перечисленных в этой главе, — далеко не полный. Но они, возможно, помогут вам начать путь к созданию собственных шикарных приложений.



Некоторые техники, обсуждаемые в данном разделе, используют передовые возможности Android. Поэтому они оформлены хуже, чем классы, рассмотренные в начале этой главы: документация не такая подробная, некоторые функции явно находятся в процессе изменения, и вы даже можете сами натолкнуться на какие-то ошибки. Если возникнут проблемы, обратитесь в группу Google Android Developers¹. Это бесценный ресурс. Иногда на вопросы о том или ином функциональном аспекте отвечает тот самый человек, который его реализовывал.

Внимательно следите за датами выпуска решений, которые находите в Вебе. Некоторые функции стремительно изменяются. Код, который работал еще полгода назад, сейчас может не работать. Конечно, из этого следует и другой вывод: любое приложение, которое достаточно широко распространится, скорее всего, будет иметь на разных платформах различные реализации функций, рассматриваемых в этом разделе. Пользуясь этими техниками, вы можете укоротить срок службы вашего приложения и сузить число устройств, на которых данное приложение будет поддерживаться.

Оставшуюся часть раздела мы посвятим изучению единственного приложения, которое очень напоминает код, приведенный в примере 8.6: пара макетов `LinearLayout`, содержащих несколько экземпляров одного и того же виджета, каждый из которых демонстрирует свой графический эффект. В примере 8.10 приведены основные части виджета, причем код, рассмотренный выше, опущен для краткости. Виджет просто рисует несколько графических объектов и определяет интерфейс, посредством которого к процессу рендеринга могут применяться различные графические эффекты.

Пример 8.10. Виджет с эффектами

```
public class EffectsWidget extends View {

    /** эффект, который будет применяться к рисунку */
    public interface PaintEffect { void setEffect(Paint paint); }

    // ...
```

¹ <http://developer.android.com/index.html>.


```

        Shader.TileMode.REPEAT));
    } }));
lv.addView(new EffectsWidget(
    this,
    5,
    new EffectsWidget.PaintEffect() {
        @Override public void setEffect(Paint paint) {
            paint.setMaskFilter(
                new BlurMaskFilter(2, BlurMaskFilter.Blur.NORMAL));
        } }));

rv.addView(new EffectsWidget(
    this,
    2,
    new EffectsWidget.PaintEffect() {
        @Override public void setEffect(Paint paint) {
            paint.setShadowLayer(3, -8, 7, Color.GREEN);
        } }));
rv.addView(new EffectsWidget(
    this,
    4,
    new EffectsWidget.PaintEffect() {
        @Override public void setEffect(Paint paint) {
            paint.setShader(
                new LinearGradient(
                    0.0F,
                    40.0F,
                    15.0F,
                    40.0F,
                    Color.BLUE,
                    Color.GREEN,
                    Shader.TileMode.MIRROR));
        } }));
View w = new EffectsWidget(
    this,
    6,
    new EffectsWidget.PaintEffect() {
        @Override public void setEffect(Paint paint) { }
    });
rv.addView(w);
w.setBackgroundResource(R.drawable.throbber);

return (AnimationDrawable) w.getBackground();
}

```

На рис. 8.5 показано, что мы увидим после запуска этого кода. Как было указано выше, виджет 6 — анимированный. Скоро мы увидим, что фон виджета 6 начинает пульсировать красным.

Мы рассмотрим каждый эффект далее.

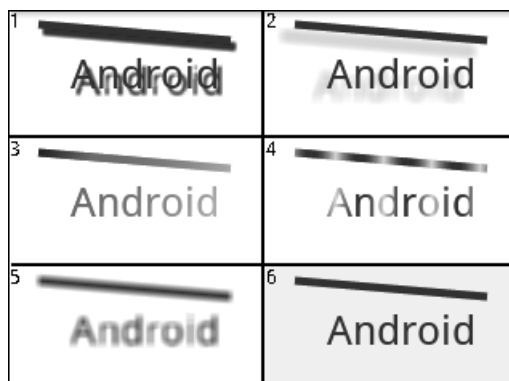


Рис. 8.5. Графические эффекты

Тени, градиенты, фильтры и аппаратное ускорение

PathEffect, MaskFilter, ColorFilter, Shader и ShadowLayer — атрибуты Paint. Все, что рисуется при помощи Paint, можно нарисовать, воспользовавшись одним или несколькими из этих преобразований. Пять верхних виджетов на рис. 8.5 — образцы нескольких таких эффектов.

На виджетах 1 и 2 показаны тени. В настоящее время тенями управляет метод `setShadowLayer`. Аргументы — радиус размывания и переносы x и y — определяют видимое расстояние и положение источника света, создающего тень относительно затененного объекта.

Во втором ряду виджетов показаны шейдеры (Shader). В инструментарии Android есть несколько готовых шейдеров. Один из них — `LinearGradient` — показан на виджетах 3 и 4. Градиент — это плавный переход между оттенками. Им можно пользоваться, например, для того, чтобы немного оживить фон страницы, не прибегая при этом к такому дорогому ресурсу, как битовые карты.

`LinearGradient` указывается с векторным значением, определяющим направление и скорость смены оттенков, массив цветов, через который будет происходить переход, и режим. Последний аргумент — режим — определяет, что должно произойти, когда одного полного перехода через градиент недостаточно, чтобы покрыть весь закрашенный объект. Например, в виджете 4 длина перехода составляет всего 15 пикселей, а сам рисунок шире 100 пикселей. Используя `Shader.TileMode.Mirror`, мы заставляем переход повторяться, меняя направление перехода на рисунке. В данном примере сначала создается градиент от голубого к зеленому длиной 15 пикселей, потом от зеленого к голубому — следующие 15 пикселей и далее по всему холсту.

Один из побочных эффектов повторной реализации фреймворка пользовательского интерфейса в вышедшем `Honeycomb` заключается в том, что некоторые сложные рисовальные эффекты оказались либо ограничены в использовании, либо вообще исчезли. Например, это коснулось и `drawTextOnPath`, и `drawTextPos`, упомянутых

выше. Метод `setShadowLayer` пока работает, но только с текстом. Если виджеты 1 и 2 отрисовываются с применением нового графического конвейера, использующего аппаратное ускорение, то текст получается с тенью, а линия над ним — нет.

Можно принудительно перевести устройство с `Noneusomb` в режим совместимости, где утраченные методы снова работают. Наиболее радикальный способ, позволяющий использовать устаревший конвейер отображения (задействующий программное, а не аппаратное ускорение), — добавить в описание приложения атрибут, показанный в примере 8.12.

Пример 8.12. Отключение аппаратного ускорения

```
<application
...
    android:hardwareAccelerated="false">
```

Кстати, эта настройка задается по умолчанию. Чтобы пользоваться аппаратным ускорением графики, появившимся в версии `Noneusomb` и применяющимся в более поздних версиях, нужно явно задать для этого атрибута в вашем приложении значение `true`. Если этого не сделать, то, несомненно, сотни приложений станут работать неправильно при установке их на устройстве, усовершенствованном до `Android v3` или выше. Итак, запомните: если вы сами не активизируете аппаратное ускорение графики для вашего приложения, такое ускорение работать не будет.

Можно обеспечить и более тонкий контроль над аппаратным ускорением. Чтобы это сделать, нужно не только включить атрибут `android:hardwareAccelerated` в элемент `application` файла описания приложения, но и применять этот атрибут к каждой отдельной активности. Если какая-то активность при использовании нового конвейера отображается неправильно, то для нее можно поставить этот атрибут в значении `false`, тогда как для остальных активностей он будет равен `true`.

Два следующих фрагмента кода, приведенных в примере 8.13, демонстрируют еще более тонкий контроль над аппаратным ускорением — соответственно на уровне окна и на уровне вида.

Пример 8.13. Тонкие способы отключения аппаратного ускорения

```
getWindow().setFlags(
    WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED,
    WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED);

myView.setLayerType(View.LAYER_TYPE_SOFTWARE, null);
```

Но не забывайте, что ваша цель — добиться правильной работы приложения при включенном аппаратном ускорении. Особенно в тех случаях, когда речь идет о новых устройствах, у вас просто не будет другой возможности провести необходимую оптимизацию. А именно оптимизация отображения при помощи аппаратного ускорения значительно повышает скорость и удобство работы с программой.

Анимация

В инструментарии пользовательского интерфейса `Android` есть еще средства для выполнения анимации. Анимации перехода (называемые в документации `Google`

tweened animations, анимациями с расчетом промежуточных кадров) представляют собой подклассы `android.view.animation.Animation`: `RotateAnimation`, `TranslateAnimation`, `ScaleAnimation` и т. д. Такие анимации используются при переходах между двумя парами видов. Анимация второго типа — подклассы от `android.graphics.drawable.AnimationDrawable`. `AnimationDrawable` — может размещаться на фоне любого виджета и позволяет создавать самые разнообразные эффекты. Наконец, имеется полнофункциональный класс на основе `SurfaceView`, который позволит вам полностью контролировать вашу собственную анимацию, создаваемую «по наитию».

Поскольку анимация обоих первых типов (анимация переходов и фоновая анимация) поддерживаются в классе `View`, это означает, что каждый из этих типов может использоваться практически с любым виджетом.

Анимация переходов

Анимация переходов начинается с вызова метода `startAnimation` класса `View` с экземпляром `Animation` в качестве параметра (или, разумеется, вашего собственного подкласса). После установки анимация проигрывается вплоть до завершения — у анимации перехода нет состояния паузы.

Центральный элемент анимации — метод `applyTransformation`. Он вызывается для генерации последовательных кадров анимации. В примере 8.14 показана реализация одного преобразования. Как видите, для анимации не генерируются цельные графические кадры. Вместо этого генерируются последовательные преобразования, которые будут применяться к анимируемому изображению. Как вы помните из пункта «Матричные преобразования» выше, такие преобразования используются для того, чтобы объект как будто двигался. Анимации перехода основаны именно на таком приеме.

Пример 8.14. Анимация перехода

```
@Override
protected void applyTransformation(float t, Transformation xf) {
    Matrix xform = xf.getMatrix();

    float z = ((dir > 0) ? 0.0f : -Z_MAX) - (dir * t * Z_MAX);
    camera.save();
    camera.rotateZ(t * 360);
    camera.translate(0.0f, 0.0f, z);
    camera.getMatrix(xform);
    camera.restore();

    xform.preTranslate(-xCenter, -yCenter);
    xform.postTranslate(xCenter, yCenter);
}
```

При применении данной конкретной реализации к объекту создается впечатление, как будто объект крутится в плоскости экрана (вызов метода `rotate`) и в то же время исчезает вдаль (вызов метода `translate`). Матрицу, которая будет применяться к целевому изображению, мы получаем от объекта `Transformation`, который был передан при вызове.

Данная реализация использует `camera`, экземпляр вспомогательного класса `Camera`. Класс `Camera` никак не связан с той камерой, которая установлена в телефоне, — это вспомогательный класс, который позволяет записывать состояние изображения. Здесь он используется для составления преобразований вращения и перехода в единой матрице, которая затем сохраняется в виде анимации перехода.

Первый параметр, сообщаемый `applyTransformation`, называется `t`. Фактически это номер кадра. Он передается как число с плавающей точкой в диапазоне от 0,0 до 1,0. Это число можно охарактеризовать и как процентный показатель того, насколько завершена анимация. В этом примере `t` используется для увеличения видимого расстояния до анимируемого объекта по оси *Z* (это линия, перпендикулярная плоскости экрана) и для задания доли от полного поворота, то есть для указания, насколько повернулось изображение. При увеличении `t` кажется, что анимируемое изображение вращается все дальше от нас против часовой стрелки, причем отдаляется от нас по оси *Z*.

Операции `preTranslate` и `postTranslate` необходимы для того, чтобы переходы изображения происходили вокруг центра этого изображения. По умолчанию матричные операции преобразуют объект, изменяя его относительно начала координат (верхнего левого угла). Если бы мы не обрамили наши преобразования в `preTranslate` и `postTranslate`, то казалось бы, что целевое изображение вращается вокруг своего левого верхнего угла. Фактически `preTranslate` перемещает начало координат к центру анимации, которая должна получиться после операции перехода, а `postTranslate` приводит к восстановлению координат после завершения перехода.

Если подумать о том, что именно должна делать анимация перехода, то становится понятно, что она фактически должна как будто совмещать две анимации: анимация с предыдущего кадра должна затухать, а анимация со следующего — нарастать. В примере 8.14 именно так применяется последняя переменная, которую мы еще не объяснили, — `dir`. Ее значение может быть равно 1 или -1. Она определяет, когда анимированное изображение как будто сжимается, а когда — постепенно увеличивается, выходя на передний план. Нам нужно просто найти способ, чтобы объединить процессы сжатия и разрастания в одну анимацию.

Для этого применяется уже знакомый вам паттерн со слушателем `Listener`. Класс `Animation` определяет слушатель, называемый `Animation.AnimationListener`. Любой экземпляр `Animation`, обладающий ненулевым слушателем, вызывает этот слушатель один раз при запуске, один раз при остановке и по разу для каждой итерации между началом и завершением этого процесса. Слушатель, замечающий, что анимация сжатия завершается, должен порождать новую анимацию, при которой изображение разрастается. Тогда вы достигнете требуемого эффекта. В примере 8.15 показана оставшаяся часть реализации этой анимации.

Пример 8.15. Состав анимации перехода

```
public void runAnimation() {
    animateOnce(new AccelerateInterpolator(), this);
}
```

```
@Override
public void onAnimationEnd(Animation animation) {
    root.post(new Runnable() {
```

```

        public void run() {
            curView.setVisibility(View.GONE);
            nextView.setVisibility(View.VISIBLE);
            nextView.requestFocus();
            new RotationTransitionAnimation(-1, root, nextView, null)
                .animateOnce(new DecelerateInterpolator(), null);
        }
    });
}

void animateOnce(
    Interpolator interpolator,
    Animation.AnimationListener listener)
{
    setDuration(700);
    setInterpolator(interpolator);
    setAnimationListener(listener);
    root.startAnimation(this);
}

```

Переход начинается с метода `runAnimation`. Переопределенный метод `onAnimationEnd`, относящийся к `AnimationListener`, порождает вторую половину процесса. Данный метод вызывается, когда целевое изображение кажется слишком отдаленным. Тогда он скрывает изображение, анимируемое в настоящий момент (`curView`), и заменяет его новым видимым изображением — `nextView`. Затем он создает новую анимацию, которая проигрывается в обратном порядке и заставляет новое изображение вращаться и увеличиваться, выступая на передний план.

Класс `Interpolator` — это пример замечательного внимания к деталям. Значения для `t`, сообщаемые `applyTransformation`, могут распределяться во времени и нелинейно. В данной реализации анимация, близкая к завершению, ускоряется, а потом, когда на передний план выступает новое изображение, снова замедляется. Такой эффект достигается благодаря применению двух интерполяторов: `AccelerateInterpolator` для первой части анимации и `DecelerateInterpolator` — для второй части. Без интерполятора разница между последовательными значениями `t`, передаваемыми `applyTransformation`, была бы постоянной. В таком случае казалось бы, что анимация протекает с постоянной скоростью. `AccelerateInterpolator` преобразует эти равномерно распределенные значения `t` в такие значения, которые располагаются сравнительно близко друг к другу в начале анимации и гораздо дальше друг от друга — в конце анимации. Благодаря этому кажется, что анимация ускоряется. `DecelerateInterpolator` создает прямо противоположный эффект. Кроме того, в Android для особых случаев предоставляются еще два интерполятора: `CycleInterpolator` и `LinearInterpolator`.

Функция объединения анимационных процессов встроена в инструментарий при помощи класса со слегка смущающим названием `AnimationSet`. В этом классе предоставляется удобный механизм составления списка (*не* множества). Этот список упорядоченный, и отдельно взятая анимация может входить в него несколько раз. Таким образом, список составляется из анимаций, которые должны проигрываться в определенном порядке. Кроме того, инструментарий предлагает несколько стандартных вариантов перехода: `AlphaAnimation`, `RotateAnimation`,

ScaleAnimation и TranslateAnimation. Разумеется, такие анимации перехода не обязательно должны быть симметричными, как в предыдущем примере. Новое изображение может постепенно проступать, оставаясь прозрачным (эффект alpha-fade in), в то время как старое изображение будет постепенно сжиматься, уходя в угол. Или новое изображение может постепенно подниматься снизу экрана, пока старое медленно растворяется. Возможности практически безграничны.

Фоновая анимация

Покадровая анимация (в документации Google — *frame-by-frame animation*) очень проста. Она представляет собой набор кадров, которые проигрываются в определенном порядке с заданными интервалами. Анимация такого рода реализуется в виде подклассов от AnimationDrawable.

Объекты AnimationDrawable, будучи подклассами Drawable, могут использоваться во всех тех контекстах, что и любые другие Drawable. Но анимирующий их механизм сам не является частью Drawable. Для осуществления анимации экземпляр AnimationDrawable пользуется внешним механизмом сервисов — реализацией интерфейса Drawable.Callback.

Класс View реализует этот интерфейс и может использоваться для анимирования AnimationDrawable. К сожалению, в таком случае анимация будет применяться *только к тому* объекту Drawable, который задан в качестве фона View.

Хорошая новость заключается в том, что этого, вполне возможно, будет достаточно. Фоновая анимация имеет доступ ко всему холсту виджета. Все рисуемое этой фоновой анимацией будет казаться нарисованным под содержимым, создаваемым методом View.onDraw. Поэтому такой способ плохо подходит для реализации полнофункциональных независимых графических объектов, свободно перемещающихся по экрану (спрайтов). И все же, умело пользуясь классом DrawableContainer (позволяющим одновременно анимировать несколько различных рисунков) и тем, что фон в любой момент можно изменить, вы сможете достаточно многого достичь и без реализации собственного анимационного фреймворка.

AnimationDrawable на фоне вида — вот и все, что вам потребуется, чтобы сделать практически что угодно: и указать, что в данный момент происходит какая-то долговременная операция (например, нарисовать крылатые конвертики, летящие от телефона к сотовой вышке), и просто заставить фон кнопки на экране пульсировать.

Пример с пульсирующей кнопкой в виджете 6 — наглядный, и при этом он очень прост в реализации. В примерах 8.16 и 8.17 показан весь необходимый код. Анимация определяется как ресурс, а код применяет анимацию к кнопке. Можно задать Drawable в качестве фона при помощи setBackgroundDrawable или setBackgroundResource.

Пример 8.16. Покадровая анимация (ресурс)

```
<animation-list
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
    <item android:drawable="@drawable/throbber_f0" android:duration="160" />
    <item android:drawable="@drawable/throbber_f1" android:duration="140" />
    <item android:drawable="@drawable/throbber_f2" android:duration="130" />
    <item android:drawable="@drawable/throbber_f3" android:duration="100" />
    <item android:drawable="@drawable/throbber_f4" android:duration="130" />
```

```

<item android:drawable="@drawable/throbber_f5" android:duration="140" />
<item android:drawable="@drawable/throbber_f6" android:duration="160" />
</animation-list>

```

В примере 8.17 показан код, запускающий анимацию. Метод `onClick` в основном виде начинает анимацию перехода и переключает виды, которые меняются местами после каждого щелчка. Кроме того, этот метод переключает и фоновую анимацию: когда фон не виден, он не анимируется.



В ранних версиях Android не существовало способа запустить фоновую анимацию из метода `Activity onCreate`. Чтобы решить такую задачу, приходилось прибегать к разным уловкам. Этот баг был исправлен в API 6 (Muffin). Достаточно написать `((AnimationBackground) view).getBackground().start()` — и все отлично работает.

Пример 8.17. Покадровая анимация (код)

```

@Override
public void onCreate(Bundle savedInstanceState) {

    // ...код опущен

    // устанавливаем слушатель щелчков для работы с анимацией
    final View root = findViewById(R.id.main);
    findViewById(R.id.main).setOnClickListener(
        new OnClickListener() {
            @Override public void onClick(View v) {
                new RotationTransitionAnimation(1, root, cur, next)
                    .runAnimation();
                // смена видов
                View t = cur;
                cur = next;
                next = t;
                toggleThrobber();
            }
        });
}

// ...код опущен

void toggleThrobber() {
    if (null != throbber) {
        if (efxView.equals(cur)) { throbber.start(); }
        else { throbber.stop(); }
    }
}

```

Здесь стоит отметить, что, если вам приходилось работать с другими фреймворками пользовательских интерфейсов, вы, возможно, привыкли рисовать фон вида в двух первых строках метода `onDraw` (или его эквивалента). Если сделать так, то рисование пойдет поверх анимации! Вообще, стоит приучить себя использовать `setBackground` для управления фоном `View` независимо от того, применяется ли в качестве фона однотонный цвет, градиент, изображение или анимация.

Загружать `DrawableAnimation` из ресурса — гибкое решение. Можно указать список отрисовываемых ресурсов (любых изображений на ваш выбор), которые составляют анимацию. Если анимация должна быть динамической, то `AnimationDrawable` — простейший инструмент для создания динамического отрисовываемого объекта, который можно анимировать на фоне `View`.

Анимация с применением `Surface View`

Для полнофункциональной анимации нам потребуется вид `SurfaceView`. Он представляет узел в дереве видов — а значит, и пространство на дисплее, — на котором может рисовать совершенно любой процесс. После компоновки и задания размеров `SurfaceView` этот узел начинает реагировать на щелчки и получать обновления, как любой другой виджет. Но он не рисует, а просто резервирует место на экране, не позволяя любым другим виджетам воздействовать на пиксели, которые оказались внутри отсеченного им прямоугольника.

Для рисования на `SurfaceView` требуется реализовать интерфейс `SurfaceHolder.Callback`. Два метода — `surfaceCreated` и `surfaceDestroyed` — информируют средство реализации о том, что поверхность изображения (`drawing surface`) соответственно доступна или недоступна для рисования. Аргументом для обоих вызовов является экземпляр третьего класса — `SurfaceHolder`. В интервале между двумя этими вызовами функция отрисовки может вызывать методы `lockCanvas` и `unlockCanvasAndPost`, относящиеся к `SurfaceView`, и редактировать пиксели в этом виде.

Все это может показаться сложным, даже по сравнению с изоэкренной анимацией, рассмотренной выше. Да, признаем, это действительно сложно. Как обычно, конкурентное исполнение повышает вероятность возникновения неприятных ошибок, которые к тому же сложно находить. Пользователь `SurfaceView` должен быть уверен не только в том, что доступ к любому состоянию, совместно используемому несколькими потоками, правильно синхронизирован, но и в том, что доступ к `SurfaceView`, `Surface` или `Canvas` будет осуществляться исключительно в интервале между вызовами `surfaceCreated` и `surfaceDestroyed`. Разумеется, загружать `DrawableAnimation` из ресурса — гибкое решение.

Если вы собираетесь пользоваться анимацией с `SurfaceView`, то, наверное, планируете работать и с графикой `OpenGL`. Как вы позже увидите, в `SurfaceView` доступно соответствующее расширение для `SurfaceView`. Правда, оно проявится в довольно неожиданном месте.

Графика `OpenGL`

Платформа Android поддерживает графику `OpenGL` примерно так же, как шляпа-цилиндр помогает вытаскивать кроликов. Хотя `OpenGL` — бесспорно, одна из самых захватывающих технологий в Android, она играет в этой операционной системе довольно маргинальную роль. Но не может не радовать, что после выхода `Noneuscomb` `OpenGL` была полностью интегрирована в графику Android.. В то время как, согласно документации, ранние версии Android поддерживали `OpenGL 1.0` и `1.1`, `Noneuscomb` не просто поддерживает `OpenGL 2.0`, но и использует эту технологию в качестве основы для отображения объектов `View`. В сущности, `OpenGL`

является предметно-ориентированным языком, интегрированным в Java. Те, кто уже давно занимается разработкой игровых пользовательских интерфейсов, смогут значительно быстрее усвоить разработку программ Android с OpenGL, чем хороший программист Java, даже эксперт по пользовательским интерфейсам, написанным на Java.

Перед тем как перейти к обсуждению самой графической библиотеки OpenGL, давайте ненадолго обратимся к вопросу о том, как именно OpenGL рисует пиксели, которые отображаются на экране. До сих пор в данной главе рассматривался сложный фреймворк View, используемый для организации и представления объектов на экране. OpenGL — это язык, на котором приложение описывает целую сцену. Такая сцена будет отображаться движком, который находится не просто за пределами виртуальной машины Java, но и вообще может работать на другом процессоре (имеется в виду графический процессор или GPU). Скоординировать на экране отображение двух видов, обрабатываемых разными процессорами, довольно не просто.

Класса SurfaceView, рассмотренного выше, для этого почти достаточно. Его задача — создавать поверхность, на которой может рисовать иной поток, кроме графического потока пользовательского интерфейса. Нам очень пригодился бы инструмент, представляющий собой расширение SurfaceView, который бы чуть лучше поддерживал многопоточность, а также обеспечивал поддержку OpenGL.

Оказывается, что именно такой инструмент уже есть. Все демонстрационные приложения, входящие в состав дистрибутива Android SDK и использующие при работе анимацию на основе OpenGL, работают с применением вспомогательного класса GLSurfaceView. Поскольку демонстрационные приложения, написанные самими создателями Android, **используют этот класс, мы советуем рассмотреть возможность его применения и в ваших программах.**

GLSurfaceView определяет интерфейс GLSurfaceView.Renderer, который радикально упрощает всю работу, связанную с OpenGL и GLSurfaceView. На самом же деле весь этот функционал исключительно сложен. GLSurfaceView вызывает метод отображения getConfigSpec для получения информации о конфигурации OpenGL. Еще два метода — sizeChanged и surfaceCreated — вызываются GLSurfaceView для того, чтобы сообщить средству отображения (рендереру) соответственно, что его размер изменился или что нужно подготовиться к рисованию. Наконец drawFrame, центральная часть интерфейса, вызывается для отображения нового кадра OpenGL.

В примере 8.18 показаны важные методы из реализации средства отображения OpenGL.

Пример 8.18. Покадровая анимация с применением OpenGL

// ...определенное состояние, заданное в конструкторе

```
@Override
public void surfaceCreated(GL10 gl) {
    // настройка поверхности
    gl.glDisable(GL10.GL_DITHER);
```

```

gl.glHint(
    GL10.GL_PERSPECTIVE_CORRECTION_HINT,
    GL10.GL_FASTEST);

gl.glClearColor(0.2f, 0.1f, 0.8f, 0.1f);
gl.glShadeModel(GL10.GL_SMOOTH);
gl.glEnable(GL10.GL_DEPTH_TEST);

// инициализация шахматной доски
initImage(gl);
}

@Override
public void drawFrame(GL10 gl) {
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glLoadIdentity();

    GLU.gluLookAt(gl, 0, 0, -5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

    // применение шахматной доски к фигуре
    gl.glActiveTexture(GL10.GL_TEXTURE0);

    gl.glTexEnvx(
        GL10.GL_TEXTURE_ENV,
        GL10.GL_TEXTURE_ENV_MODE,
        GL10.GL_MODULATE);
    gl.glTexParameterx(
        GL10.GL_TEXTURE_2D,
        GL10.GL_TEXTURE_WRAP_S,
        GL10.GL_REPEAT);
    gl.glTexParameterx(
        GL10.GL_TEXTURE_2D,
        GL10.GL_TEXTURE_WRAP_T,
        GL10.GL_REPEAT);

    // анимация
    int t = (int) (SystemClock.uptimeMillis() % (10 * 1000L));
    gl.glTranslatef(6.0f - (0.0013f * t), 0, 0);

    // рисование
    gl.glFrontFace(GL10.GL_CCW);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuf);
    gl.glEnable(GL10.GL_TEXTURE_2D);
    gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, textureBuf);
    gl.glDrawElements(
        GL10.GL_TRIANGLE_STRIP,

```

```

        5,
        GL10.GL_UNSIGNED_SHORT, indexBuf);
    }

    private void initImage(GL10 gl) {
        int[] textures = new int[1];
        gl.glGenTextures(1, textures, 0);
        gl.glBindTexture(GL10.GL_TEXTURE_2D, textures[0]);

        gl.glTexParameterf(
            GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_MIN_FILTER,
            GL10.GL_NEAREST);
        gl.glTexParameterf(
            GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_MAG_FILTER,
            GL10.GL_LINEAR);
        gl.glTexParameterf(
            GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_WRAP_S,
            GL10.GL_CLAMP_TO_EDGE);
        gl.glTexParameterf(
            GL10.GL_TEXTURE_2D,
            GL10.GL_TEXTURE_WRAP_T,
            GL10.GL_CLAMP_TO_EDGE);
        gl.glTexEnvf(
            GL10.GL_TEXTURE_ENV,
            GL10.GL_TEXTURE_ENV_MODE,
            GL10.GL_REPLACE);

        InputStream in
            = context.getResources().openRawResource(R.drawable.cb);
        Bitmap image;
        try { image = BitmapFactory.decodeStream(in); }
        finally {
            try { in.close(); } catch(IOException e) { }
        }

        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, image, 0);

        image.recycle();
    }

```

Метод `surfaceCreated` готовит сцену. Он задает несколько атрибутов OpenGL, которые должны инициализироваться только в том случае, когда виджет получает новую рисовальную поверхность. Кроме того, этот метод вызывает `initImage`, который считывает ресурс точечного рисунка (`bitmap resource`) и сохраняет его в виде двумерной текстуры. Когда наконец вызывается `drawFrame`, все уже готово для рисования. А именно: текстура применена к плоской фигуре, вершины которой задаются конструктором в `vertexBuf`, фаза анимации выбрана, и сцена перерисовывается. На рис. 8.6 показан рабочий пример с OpenGL.

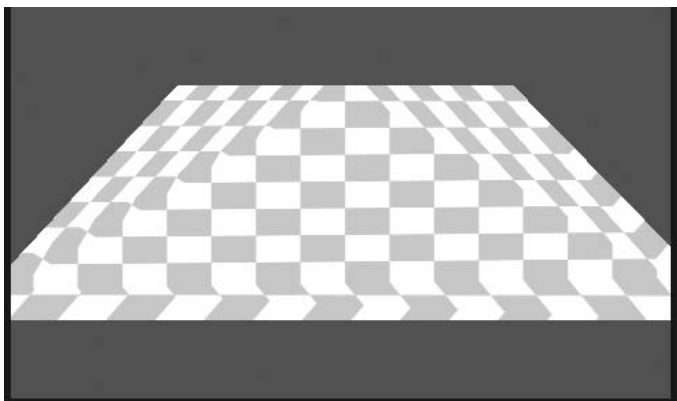


Рис. 8.6. Рисунок с применением OpenGL

Как всегда, не забывайте, что ваш вид встраивается в активность, которая имеет жизненный цикл! Если вы используете OpenGL, обязательно отключайте длительные анимационные процессы, когда активность не видна.

9

Обращение с данными и их долговременное хранение

Чтобы выполнять все то множество функций, которые предлагаются в современных мобильных телефонах, — отслеживать контакты, события и задачи, — мобильная операционная система и ее приложения должны быть особенно хороши в области хранения больших объемов данных и отслеживания их изменений. Обычно все эти данные структурированы в виде строк и столбцов, как в таблице или очень простой базе данных. Наряду с обычными требованиями по хранению данных, предъявляемыми к приложению, в жизненном цикле приложения **Android** необходимо обеспечить быстрое и надежное долговременное сохранение данных. Только так программа может существовать в условиях постоянной энергозависимости, свойственной мобильной среде, где сплошь и рядом неожиданно разряжаются батареи и операционная система **Android** может вдруг решить, что программу необходимо выгрузить из памяти.

Android предоставляет легкую, но мощную реляционную базу данных **SQLite** для долговременного хранения данных. Более того, как было рассказано в главе 3, существующие в системе **Android** поставщики содержимого позволяют приложениям предоставлять свои данные другим приложениям.

В этой главе предлагается простое руководство по работе с **SQL**, которое поможет освоить тему долговременного хранения данных в **Android** с помощью **SQLite**. Кроме того, мы покажем вам интересное приложение — **MJAndroid**, — позволяющее получить реальное впечатление о том, как управлять базой данных **Android**. В главе 15 мы на том же примере продемонстрируем, как в **Android** работает картографический **API**. В главе 12 будет показана реализация поставщика содержимого.

Обзор реляционной базы данных

Реляционная база данных предоставляет эффективную, структурированную и универсальную систему для управления информацией, требующей долговременного хранения. При применении базы данных приложения используют структурированные

запросы для преобразования информации в долговременные двухмерные матрицы, называемые *таблицами* (а в первых теоретических статьях на эту тему — *отношениями*). Разработчики пишут запросы на высокоуровневом языке SQL (Standard Query Language, язык структурированных запросов). SQL — это общепринятый язык для работы с системами управления реляционными базами данных (СУРБД, или RDMBS). Этот популярный инструмент для управления данными активно используется уже с конца 1970-х годов. SQL стал применяться в промышленных масштабах, когда его приняли Национальный институт стандартов и технологии (в 1986 году) и организация ISO (в 1987 году). Он используется повсюду — от терабайтовых систем Oracle и SQL Server до, как вы увидите, мобильного телефона, где при помощи этого языка хранится электронная почта.

Таблицы баз данных — как раз то, что надо для хранения наборов данных, в которых встречается много экземпляров идентичных сущностей. С такими наборами постоянно приходится сталкиваться при разработке программ. Например, в телефонной книге содержится множество контактов, и во всех этих контактах потенциально записывается однотипная информация (то есть адрес, телефонный номер и т. д.). Каждая строка данных в таблице хранит информацию об отдельном человеке, причем в каждом столбце содержится специфический атрибут этого человека: имена в одном столбце, адреса в другом, номера домашних телефонов — в третьем. Если определенный человек связан с несколькими компонентами (например, у него несколько адресов), реляционные базы данных умеют обрабатывать и такую информацию, но в данной главе мы не будем подходить к проблеме настолько детально.

SQLite

В качестве движка базы данных Android использует SQLite. Это самодостаточная транзакционная база данных, для работы которой не требуется отдельного серверного процесса. SQLite используется и во многих средах и приложениях, а разработкой SQLite активно занимается обширное свободное сообщество. В отличие от баз данных, ориентированных на работу с ПК (они еще называются корпоративными базами данных) и предоставляющих массу функций, которые обеспечивают отказоустойчивость и конкурентный доступ к данным, в SQLite последовательно упраздняются все функции, кроме тех, что абсолютно необходимы. Это делается для того, чтобы база данных занимала как можно меньше места в памяти. Например, во многих системах баз данных используется статическая типизация, но в SQLite информация о типе базы данных не сохраняется. Вместо этого задача хранения информации о типах делегируется высокоуровневым языкам, например Java, которые отображают структуры базы данных на высокоуровневые типы.

SQLite — это не проект Google, хотя Google и участвовала в ее создании. Разработкой SQLite занимается международная команда специалистов, которые из всех сил стремятся улучшить функционал и надежность этой технологии (движка). Надежность — основная составляющая SQLite. Больше половины кода в этом проекте связано с тестированием библиотеки. Библиотека справляется с устранением разнообразных системных ошибок, в частности дефицита памяти, ошибок,

связанных с дисками, перебоев с питанием. База данных ни в коем случае не должна оказаться в невосстанавливаемом состоянии. На мобильном телефоне такая ситуация привела бы к серьезной ошибке, поскольку в базе данных часто сохраняется критически важная информация. Правда, приятно отметить, что SQLite не-легко вывести из строя — в противном случае, если батарея неожиданно откажет, мобильный телефон превратится в абсолютно ненужное устройство.

Обширная и подробная документация по проекту SQLite расположена по адресу <http://www.sqlite.org/docs.html>.

Язык SQL

Для написания программ в Android обычно требуются базовые навыки программирования на языке SQL, хотя для решения наиболее распространенных задач, связанных с данными, предоставляются более высокоуровневые классы. Эту главу можно считать введением в SQL. Хотя SQL и не является темой нашей книги, мы достаточно подробно расскажем вам об Android-ориентированной SQL, и вы сможете обеспечивать долговременное хранение данных в самых разнообразных приложениях для Android. Более подробное описание языка SQLite приводится по адресу <http://www.sqlite.org/lang.html>. Мы расскажем о языке SQLite на примере простых команд SQL и по ходу повествования покажем, как пользоваться командой `sqlite3` и как именно запросы изменяют таблицы, к которым они применяются. Кроме того, вам может пригодиться руководство W3Schools на эту тему: http://www.w3schools.com/sql/sql_intro.asp.

При работе с SQLite база данных — это обычный файл в файловой системе Android, который может находиться во внутренней памяти телефона или на карте памяти. Но, как вы увидите, базы данных большинства приложений располагаются в каталоге `/data/data/com.example.yourAppPackage/databases`. Можно запустить команду `ls` в этом каталоге в оболочке `adb` — так составляется список баз данных, которые Android создал за вас.

База данных занимается долговременным хранением данных. Это означает, что она обновляет файл SQLite способом, который указывается в каждом предложении SQL, выдаваемом программой. Далее по тексту мы описываем, как различные команды SQLite используются в утилите `sqlite3`, предназначенной для работы с командной строкой. Позже мы покажем, как достичь аналогичных эффектов при помощи API Android. Хотя функция работы с SQL из командной строки не предусмотрена в приложении, которое мы создаем, она определенно пригодится вам для отладки приложений по мере их разработки. Вы увидите, что написание кода для базы данных в Android — это, как правило, итеративный процесс написания кода Java для управления таблицами, а потом — взятия созданных данных из ячеек таблицы с помощью командной строки.

Команды определения данных в SQL

Все предложения языка SQL можно разделить на две крупные категории: используемые для создания и изменения таблиц (то есть хранилищ с данными)

и применяемые для создания, считывания, обновления и удаления данных в этих таблицах. В этом подразделе рассмотрим первую группу: команды описания данных.

- **CREATE TABLE** — работа с SQL начинается с создания таблицы для хранения данных. Команда **CREATE TABLE** создает новую таблицу в базе данных SQLite. Команда указывает имя таблицы, которое должно быть уникальным среди всех таблиц базы данных, и различные *столбцы (columns)* этой таблицы для хранения данных. Каждый столбец имеет уникальное имя в данной таблице, а также тип (в SQL определяются разные типы, например дата или текстовая строка). В столбце также могут указываться и другие атрибуты, например такие: должны ли значения быть уникальными, задается ли по умолчанию стандартное значение, если строка вставляется без указания значения, может ли в столбце присутствовать значение NULL.

Таблица базы данных чем-то напоминает таблицу Excel. Продолжая пример с базой телефонных контактов, предположим, что в каждой строке таблицы содержится информация об одном контакте. В столбцах таблицы располагаются различные фрагменты информации, которую вы собираете о каждом контакте: имя, фамилия, день рождения и т. д. Несколько примеров, которые мы приведем в этой главе, помогут вам приступить к работе с нашей базой данных, содержащей задачи.



Таблицы, создаваемые при помощи предложений SQL **CREATE TABLE**, и атрибуты, которые в них содержатся, называются схемой базы данных.

- **DROP TABLE** — удаляет таблицу, которая была добавлена командой **CREATE TABLE**. Данная команда принимает имя таблицы, которую требуется удалить. После того как эта команда выполнится, любые данные, содержащиеся в удаленной таким образом таблице, восстановить будет невозможно.

Вот пример кода на языке SQL, который создает, а потом удаляет простую таблицу для хранения контактов:

```
CREATE TABLE contacts (  
    first_name TEXT,  
    last_name TEXT,  
    phone_number TEXT,  
    height_in_meters REAL);
```

```
DROP TABLE contacts;
```

При вводе команд с помощью `sqlite3`, каждая команда должна завершаться символом `;`.

После того как таблица будет создана, схему базы данных можно изменять (например, вам может потребоваться добавить столбец или изменить значение столбца, заданное по умолчанию). Для внесения таких изменений нужно использовать команду **ALTER TABLE**.

Типы SQLite

Выше говорилось, что необходимо указывать тип для каждого из столбцов, которые вы создаете в описываемых таблицах. В SQLite поддерживаются следующие типы данных.

- TEXT — текстовая строка, сохраняемая в кодировке базы данных (UTF-8, UTF-16BE или UTF-16LE). Этот тип данных — самый распространенный.
- REAL — значение с плавающей точкой, сохраняемое как восьмибайтное число с плавающей точкой стандарта IEEE.
- BLOB — любые двоичные данные, сохраненные именно в том виде, в каком они были введены. Тип данных BLOB можно использовать для сохранения любых данных переменной длины, например исполняемых файлов или сохраненных на устройстве изображений из Интернета. Обычно данные типа BLOB сильно нагружают мобильную базу данных, и, как правило, их следует избегать. В главе 13 мы покажем альтернативную схему сохранения изображений, взятых из Интернета.
- INTEGER — это целое число со знаком, сохраненное в 1, 2, 3, 4, 6 или 8 байтах, в зависимости от порядка величины.

Подробная информация по типам SQLite предоставляется по адресу <http://www.sqlite.org/datatype3.html>.

Ограничения, определяемые для базы данных

Ограничения, определяемые для базы данных (database constraints), — это особые атрибуты, присваиваемые для ее таблиц. Некоторые ограничения являются информационно-ориентированными, например требуют, чтобы все значения в столбце были уникальными. Таков, например, будет столбец с номерами страховых полисов. Другие ограничения характеризуются более функциональными свойствами. Основу межтабличных связей составляют реляционные ограничения PRIMARY KEY и FOREIGN KEY.

В большинстве таблиц должен присутствовать специальный столбец, который уникально идентифицирует каждую отдельно взятую строку. В SQL такой столбец называется PRIMARY KEY. Как правило, он используется только как идентификатор каждой строки. В отличие, например, от номера страхового полиса он не имеет никакого дополнительного значения. Поэтому нет нужды задавать значения для данного столбца. Вместо этого можно позволить SQLite присваивать приращиваемые целочисленные значения добавляемым в таблицу новым строкам. В других базах данных для достижения аналогичного результата обычно требуется специально отметить такой столбец как автоинкрементный. В SQLite также предлагается явное ограничение AUTOINCREMENT, но первичные ключи (PRIMARY KEY) являются автоинкрементными по умолчанию. Приращиваемые значения в этом столбце функционально напоминают указатели на неявные объекты (opaque pointers), присутствующие в таких высокоуровневых языках, как Java или C: другие таблицы баз данных и код высокоуровневого языка могут использовать данный столбец для ссылки на конкретную строку.

Когда строки баз данных имеют уникальные первичные ключи, можно начать задумываться о зависимостях между таблицами. Например, таблица, используемая для хранения данных о сотрудниках, может определить столбец с целочисленными значениями под названием `employer_id`. В этом столбце будут содержаться значения первичных ключей строк другой таблицы, `employers` (эта таблица будет содержать данные о работодателях). Если выполнить запрос и выбрать одну строку или более из таблицы `employers`, то можно собрать идентификационные номера работодателей и просмотреть сотрудников в таблице `employees` по столбцу `employer_id`. Так программа может находить сотрудников, занятых у определенного работодателя. Две рассматриваемые таблицы (мы сократили их до нескольких столбцов, важных для данного конкретного примера) могут выглядеть так:

```
CREATE TABLE employers (  
    _id INTEGER PRIMARY KEY,  
    company_name TEXT);
```

```
CREATE TABLE employees (  
    name TEXT,  
    annual_salary REAL NOT NULL CHECK (annual_salary > 0),  
    employer_id REFERENCES employers(_id));
```

Идея таблицы, ссылающейся на первичный ключ другой таблицы, формально поддерживается в SQL как ограничение `FOREIGN KEY`, гарантирующее валидность межтабличных ссылок. Данное ограничение сообщает базе данных, что целочисленные значения в столбце с ограничением по внешнему ключу (`foreign key constraint`) должны ссылаться на валидные первичные ключи строк базы данных, относящихся к другой таблице. Следовательно, если вставить в таблицу `employees` такую строку, чей `employer_id` отсутствует в таблице `employers`, то во многих разновидностях SQL вы получите сообщение о нарушении ограничения. Это поможет вам избежать появления «висящих» ссылок. Такой механизм также называют принудительным внедрением внешних ключей (`enforcement of foreign keys`). Однако в **SQLite** ограничение по внешнему ключу является необязательным, и в Android оно отключено. Нельзя полагаться на ограничение по внешнему ключу для отлавливания недействительных ссылок на внешние ключи. Поэтому, создавая схемы баз данных, использующие внешние ключи, нужно действовать особенно тщательно.

Существует еще несколько ограничений, эффекты от применения которых менее масштабны.

- **UNIQUE** — такое ограничение требует, чтобы значение конкретного столбца отличалось от всех остальных значений данного столбца во всех имеющихся строках. Действует при каждой вставке новой строки или обновлении существующей. Любая операция вставки или обновления, при которой происходит попытка занести в таблицу значение-дубль, нарушает ограничение SQLite.
- **NOT NULL** — не допускает, чтобы какое-либо значение в данном столбце было равно `NULL`. Обратите внимание: первичный ключ одновременно удовлетворяет условиям **UNIQUE** и **NOT NULL**.
- **CHECK** — принимает булево выражение и требует, чтобы данное выражение возвращало `true` для любого значения, вставляемого в столбец. Пример — атрибут `CHECK (annual_salary > 0)`, показанный выше в таблице `employees`.

Команды манипуляции данными в SQL

После того как будет закончено определение таблиц при помощи команд определения данных, можно вставлять ваши данные в базу данных и направлять к ней запросы. Следующие команды манипуляции данными применяются в предложениях SQL особенно часто.

○ **SELECT** — это предложение является основным инструментом для выполнения запросов к базе данных. Результатом выполнения этого запроса являются ноль или более строк с данными, где в каждой строке имеется фиксированное количество столбцов. Можно сказать, что предложение **SELECT** создает новую таблицу, состоящую только из тех строк и столбцов, которые вы выберете в данном предложении. Предложение **SELECT** — самая сложная команда в языке SQL, она обеспечивает разнообразные способы построения отношений между данными в одной или нескольких таблицах баз данных. API Android поддерживает для команды **SELECT** языка SQL следующие условия:

- **FROM** — указывает таблицы, из которых будут извлекаться данные для выполнения запроса;
- **WHERE** — обозначает условия, которым должны удовлетворять выбранные строки таблицы, чтобы их можно было вернуть в запросе;
- **GROUP BY** — упорядочивает результаты по группам в соответствии с именем столбца;
- **HAVING** — далее ограничивает результаты, сравнивая полученные группы с выражениями. Можно удалять из запроса такие группы, в которых не набирается минимального требуемого количества элементов;
- **ORDER BY** — задает порядок сортировки результатов запроса, указывая имя столбца, по которому будет происходить сортировка, и функцию (например, **ASC** для возрастания, **DSC** для убывания), которая будет сортировать строки по элементам, содержащимся в указанном столбце;
- **LIMIT** — ограничивает количество строк в запросе до указанного значения (например, пять строк).

Вот несколько примеров предложений **SELECT**:

```
SELECT * FROM contacts;
```

```
SELECT first_name, height_in_meters  
FROM contacts  
WHERE last_name = "Smith";
```

```
SELECT employees.name, employers.name  
FROM employees, employers  
WHERE employee.employer_id = employer._id  
ORDER BY employer.company_name ASC;
```

Первое предложение получает все строки из таблицы `contacts`, поскольку для фильтрации результатов не применяется условие **WHERE**. Возвращаются все столбцы (обозначаются звездочкой `*`) полученных строк. Второе предложение

получает данные об именах и росте членов семьи Смит. Последнее предложение выводит на печать список сотрудников и их работодателей, отсортированный по названию компании.

Подробнее о команде SELECT — по адресу http://www.sqlite.org/lang_select.html.

- INSERT — данное предложение добавляет новую строку в указанную таблицу базы данных вместе с множеством указанных значений. Причем значения для каждого столбца имеют подходящий для данного столбца тип SQLite (например, значение 5 подходит для столбца с целочисленными значениями `integer`). При вставке может быть указан список столбцов, которые будут затронуты вставкой. Причем этот список может содержать меньше столбцов, чем всего имеется в таблице. Если не задать значений для некоторых из столбцов, SQLite укажет для каждого такого столбца стандартное значение (по умолчанию) — опять же если вы задали такое значение в предложении CREATE TABLE. Если вы не указали стандартного значения, SQLite использует в таком качестве значение NULL.

Вот несколько примеров предложений INSERT:

```
INSERT INTO contacts(first_name)
VALUES("Thomas");
```

```
INSERT INTO employers VALUES(1, "Acme Balloons");
INSERT INTO employees VALUES("Wile E. Coyote", 100000.000, 1);
```

Первая команда добавляет в список контактов запись о каком-то человеке, которого зовут Томас, а вот фамилия, телефонный номер и рост Томаса неизвестны (NULL). Второе предложение добавляет в список работодателей Acme Balloons, а третья — указывает, что Уайл И. Койот является сотрудником этой компании.

Подробнее о команде INSERT — по адресу http://www.sqlite.org/lang_insert.html.

- UPDATE — данное предложение изменяет несколько строк в заданной таблице, записывая в них новые значения. При каждом присваивании указывается имя таблицы и функция, которая должна предоставить новое значение для конкретного столбца. Как и в случае с SELECT, можно указать условие WHERE, идентифицирующее строки, которые должны быть обновлены, когда задействуется команда UPDATE. Как и в случае с INSERT, можно указать список столбцов, которые должны быть изменены при выполнении команды. Список столбцов работает так же, как и с INSERT. Очень важное значение имеет условие WHERE: если оно не находит совпадений ни с одной из строк, то команда UPDATE не окажет никакого эффекта. Но если опустить условие, то утверждение затронет все строки таблицы.

Вот несколько примеров предложений UPDATE:

```
UPDATE contacts
SET height_in_meters = 10, last_name = "Jones"
```

```
UPDATE employees
```

```
SET annual_salary = 200000.00
WHERE employer_id = (
    SELECT _id
    FROM employers
    WHERE company_name = "Acme Balloons");
```

Первое предложение заявляет, что все ваши друзья — великаны по фамилии Джонс. Вторая — это более сложный запрос. Она значительно поднимает зарплату всем сотрудникам организации Acme Balloons.

Подробнее о команде UPDATE — по адресу http://www.sqlite.org/lang_update.html.

Дополнительные концепции, связанные с базой данных

Итак, вы уже знаете достаточно много простых команд SQL, чтобы приступить к работе с базами данных в Android. Но по мере того, как создаваемые вами приложения будут становиться все сложнее, вам, вероятно, придется воспользоваться следующими конструктами SQL, которые мы не будем детально рассматривать в нашей книге.

- Внутреннее соединение (inner join) — выбирает данные из двух и более таблиц, в которых данные связаны друг с другом по внешнему ключу. Запросы такого типа полезны при сборке объектов, которые были распределены по двум и более таблицам. В приведенном выше примере с работодателем/сотрудником мы наблюдали внутреннее соединение. Как мы уже указывали, поскольку Android не требует обязательного использования внешних ключей, здесь могут начаться проблемы. А именно — если выяснится, что ключ для соединения не является валидной межтабличной ссылкой, то есть если столбец с внешним ключом на самом деле указывает на несуществующий первичный ключ строки из другой таблицы.
- Составной запрос — SQLite поддерживает сложные манипуляции с базами данных, так как позволяет использовать комбинации предложений. В одном из примеров с обновлениями, показанном выше, мы видели составной запрос, где команда SELECT была включена в UPDATE.
- Триггеры — позволяют разработчику писать предложения SQL, которые будут получать обратный вызов, если состояние базы данных изменится тем или иным образом.

Чтобы подробно изучить эти темы, рекомендуем вам обратиться к соответствующим источникам.

Транзакции базы данных

Транзакции базы данных позволяют делать последовательности предложений SQL атомарными: либо все предложения выполняются успешно, либо ни одно из них не оказывает воздействия на базу данных. Это свойство может быть важным,

например если в вашем приложении возникнет нештатная ситуация (допустим, крах системы). Транзакция гарантирует, что если устройство отключается прямо в ходе выполнения определенной последовательности операций, то ни одна из уже выполненных операций не отразится на базе данных. В контексте баз данных принято говорить, что транзакции SQLite обеспечивают пресловутые свойства ACID: <http://ru.wikipedia.org/wiki/ACID>.

При работе с SQLite каждая операция, вносящая изменения в базу данных, выполняется в собственной транзакции. Это означает, что разработчик может быть уверен: все вставляемые значения будут записаны в базу данных лишь в том случае, если выполнится все предложение целиком. Кроме того, можно явно начинать и завершать транзакцию так, чтобы она объединяла несколько предложений. При каждой конкретной транзакции SQLite не изменяет базу данных до тех пор, пока все предложения этой транзакции не завершатся успешно.

Учитывая энергозависимость, свойственную для мобильной среды Android, рекомендуем не только обеспечивать стабильность информации, необходимую для приложения, но и смело использовать транзакции — это поможет повысить отказоустойчивость вашего приложения.

Пример работы с базой данных с использованием sqlite3

Теперь, когда вы знаете основы SQL и понимаете, как этот язык соотносится с SQLite, рассмотрим простую базу данных для хранения метаданных видео. Мы будем применять команду `sqlite3` для работы с командной строкой, а также отладочную оболочку Android, которую можно запустить командой `adb`. Пользуясь командной строкой, можно просматривать изменения базы данных по мере их возникновения. Кроме того, мы покажем несколько простых примеров того, как работать с этим полезным инструментом отладки базы данных. Более подробная информация о `sqlite3` приводится по адресу <http://www.sqlite.org/sqlite.html>. Обратите внимание на то, что в первый раз этот пример лучше запустить на эмуляторе Android, поскольку для запуска его на реальном устройстве вам потребуются административные (привилегированные) права доступа.

Для начала инициализируем базу данных:

```
$ adb shell
# cd /data/data/
# mkdir com.oreilly.demo.pa.ch10.sql
# cd com.oreilly.demo.pa.ch10.sql
# mkdir databases
# cd databases
#
# sqlite3 simple_video.db
SQLite version 3.6.22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```



Обратите внимание: разработчикам не приходится создавать эти каталоги вручную, как мы поступили в данном примере, поскольку система **Android сама создает их во время установки приложения**. Этап создания каталогов необходим только в данном конкретном примере, так как у нас еще нет приложения, в котором система могла бы автоматически создавать такие каталоги.

Командная строка `sqlite3` принимает команды двух разновидностей: стандартные команды **SQL** и однословные команды, начинающиеся с точки (`.`). Ниже во вводном сообщении показана первая (и, пожалуй, наиболее важная) из этих команд: `.help`. Попробуйте ее и увидите, какие команды вам доступны.

```
sqlite> .help
.bail ON|OFF          Stop after hitting an error. Default OFF
.databases            List names and files of attached databases
.dump ?TABLE? ...    Dump the database in a SQL text format
.echo ON|OFF         Turn command echo on or off
.exit                Exit this program
.explain ON|OFF       Turn output mode suitable for EXPLAIN on or off.
.header(s) ON|OFF    Turn display of headers on or off
.help                Show this message
.import FILE TABLE  Import data from FILE into TABLE
.indices TABLE       Show names of all indices on TABLE
.load FILE ?ENTRY?   Load an extension library
.mode MODE ?TABLE?   Set output mode where MODE is one of:
                    csv      Comma-separated values
                    column    Left-aligned columns. (See .width)
                    html     HTML <table> code
                    insert    SQL insert statements for TABLE
                    line      One value per line
                    list       Values delimited by .separator string
                    tabs       Tab-separated values
                    tcl        TCL list elements
.nullvalue STRING     Print STRING in place of NULL values
.output FILENAME      Send output to FILENAME
.output stdout        Send output to the screen
.prompt MAIN CONTINUE Replace the standard prompts
.quit                Exit this program
.read FILENAME        Execute SQL in FILENAME
.schema ?TABLE?       Show the CREATE statements
.separator STRING     Change separator used by output mode and .import
.show                Show the current values for various settings
.tables ?PATTERN?    List names of tables matching a LIKE pattern
.timeout MS           Try opening locked tables for MS milliseconds
.timer ON|OFF         Turn the CPU timer measurement on or off
.width NUM NUM ...    Set column widths for "column" mode
```

В этом списке есть и еще одна важная команда — `.exit`. Запомните ее! Это выход из этого длинного списка. Для выхода можно также нажать сочетание клавиш **Ctrl+D**.

Кроме того, важно запомнить, что каждая команда SQL должна завершаться точкой с запятой. Если вы увидите что-то подобное:

```
sqlite> ls
...>
```

то, значит, SQLite полагает, что вы собираетесь вводить предложение SQL, и ожидает символа ; в конце предложения. Обратите также внимание, что команды, начинающиеся с точки (.), не должны завершаться точкой с запятой.



Мы воспользовались ls в качестве примера команды, которую пользователь может ввести по рассеянности, забыв, что работает с sqlite3. На самом деле ls не является командой sqlite3. Если поставить после ls точку с запятой, sqlite пожалуется на ошибку, а потом вы сможете вводить правильные команды, начинающиеся с точки, или предложения sql.

Пока нам не особенно интересно большинство команд, начинающихся с точки, поскольку сама база данных еще пуста. Давайте введем какие-нибудь данные:

```
sqlite> create table video (
...> _id integer primary key,
...> title text,
...> description text,
...> url text);
```

В этих строках создается новая таблица под названием video. Типы столбцов — integer и text. Таблица содержит первичный ключ _id. Имя для данного конкретного столбца выбрано не случайно. Android требует использовать именно это имя, чтобы таблица могла работать с системой курсоров самой ОС Android.

Чтобы просмотреть новоиспеченные таблицы, воспользуемся «точечной» командой .table:

```
sqlite> .table
video
sqlite>
```

Рассмотрим несколько различных запросов, которые иллюстрируют изученные выше концепции SQL, а также описывают приложение, основанное на этих таблицах. Вставим в наши новые таблицы какие-нибудь данные, чтобы запросы могли вернуть результат:

```
INSERT INTO video (_id, title, url)
VALUES(1, "Epic Fail Car",
"http://www.youtube.com/watch?v=0lynapTnYVkeGE");
INSERT INTO video (_id, title, url)
VALUES(2, "Epic Fail Bicycle",
"http://www.youtube.com/watch?v=7n7apTnYVkeGE");
INSERT INTO video (_id, title, url)
VALUES(3, "Epic Fail Wagon",
"http://www.youtube.com/watch?v=m0iGn2c47LA");
INSERT INTO video (_id, title, url)
VALUES(4, "Epic Fail Sidewalk",
```

```
"http://www.youtube.com/watch?v=m0iGn2cNcNo");
INSERT INTO video (_id, title, url)
VALUES(5, "Epic Fail Motorcycle",
"http://www.youtube.com/watch?v=7n7apBB8qkeGE");
```

Аккуратно закрывайте кавычки. Если вы введете одиночную кавычку, то `sqlite3` постоянно будет напоминать вам об этом, пока вы не закроете кавычки.

В данном примере мы не вводим значения для всех столбцов таблицы. Содержимое скобок после фразы `INTO` в предложении — это список столбцов, в которые предложение будет записывать данные. В скобках после фразы `VALUES` содержатся сами значения в том же порядке.

Теперь предположим, что вы хотите найти все названия видео, в которых присутствует последовательность `cycle`. Воспользуемся запросом `SELECT`:

```
sqlite> SELECT title FROM video WHERE title LIKE "%cycle%";
Epic Fail Bicycle
Epic Fail Motorcycle
```

`sqlite3` выводит строки по одной. В данном примере мы набрали заглавными буквами зарезервированные ключевые слова `SQL`, чтобы синтаксис был яснее. Так делать не обязательно. Регистр может быть верхним, нижним или смешанным.

В примере также показано использование сопоставления с образцом, доступное в `SQL`. Ключевое слово `LIKE` в комбинации с подстановочным (джокерным) символом `%` позволяет сопоставлять фрагменты строк.

Предположим, нам нужно выстроить список всех видеофайлов вместе с их URL, которые должны быть отсортированы в алфавитном порядке по заголовкам:

```
sqlite> SELECT title, url FROM video ORDER BY title DESC;
Epic Fail Wagon|http://www.youtube.com/watch?v=m0iGn2c47LA
Epic Fail Sidewalk|http://www.youtube.com/watch?v=m0iGn2cNcNo
Epic Fail Motorcycle|http://www.youtube.com/watch?v=7n7apBB8qkeGE
Epic Fail Car|http://www.youtube.com/watch?v=01ynapTnYVkeGE
Epic Fail Bicycle|http://www.youtube.com/watch?v=7n7apTnYVkeGE
```

Как видите, в `sqlite3` для разделения значений, относящихся к различным столбцам, применяется вертикальная черта (`|`).

Мы пока не ввели описания наших видеофайлов. Давайте добавим хотя бы одно:

```
sqlite> UPDATE video SET description="Crash!" WHERE title LIKE "%Car";
sqlite> UPDATE video SET description="Trip!" WHERE title LIKE "%Sidewalk%";
sqlite> SELECT title, description FROM video WHERE NOT description IS NULL;
Epic Fail Car|Crash!
Epic Fail Sidewalk|Trip!
```

Наконец, давайте удалим запись — воспользуемся для этого ее ID:

```
sqlite> DELETE FROM video WHERE _id = 1;
sqlite> SELECT _id, description FROM videos;
2|Epic Fail Bicycle
3|Epic Fail Wagon
4|Epic Fail Sidewalk
5|Epic Fail Motorcycle
```

SQL и модель построения архитектуры вокруг базы данных в приложениях Android

Теперь, обладая базовыми навыками программирования на языке SQL, можно подумать и о том, как применить их при написании приложения для Android. Наша цель — создавать надежные приложения, базирующиеся на паттерне MVC («Модель-вид-контроллер»). Этот же паттерн лежит в основе качественно сделанных программ с пользовательскими интерфейсами, причем принцип его организации хорошо подходит для Android. Статья «Википедии» о паттерне MVC — <http://ru.wikipedia.org/wiki/Model-View-Controller>.

Одно из фундаментальных различий между приложениями для мобильных телефонов и персональных компьютеров заключается в том, как в них организуется долговременное хранение данных. Традиционные приложения для ПК — текстовые редакторы, текстовые процессоры, рисовальные программы, программы для составления презентаций и т. д. — часто используют такой вариант паттерна MVC, который ориентирован на работу с документами. Эти программы открывают документ, считывают его в память компьютера и в памяти превращают его в объекты, образующие модель данных. Такие программы создают представления модели данных, обрабатывают пользовательский ввод при помощи контроллера, а затем модифицируют модель данных (рис. 9.1). Основное следствие такой архитектуры заключается в том, что вы специально загружаете и сохраняете документы, чтобы модель данных сохранялась между отдельными запусками программы. Мы уже видели, как в Android работают компоненты пользовательского интерфейса. Теперь исследуем существующие в Android API для управления базами данных и подготовимся к внедрению такой модели данных, которая будет работать по-новому.

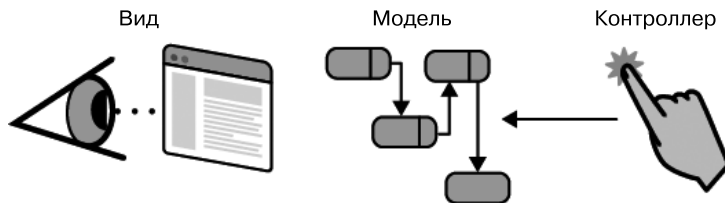


Рис. 9.1. Приложения, ориентированные на работу с документами и реализующие модель данных с объектами, которые находятся в памяти

Для надежной работы Android система иначе комбинирует модели данных и элементы пользовательского интерфейса. Приложения работают на мобильных устройствах, которые располагают ограниченным объемом памяти. Кроме того, батареи таких устройств могут разрядиться когда угодно, и, как правило, это происходит некстати. Помимо того, для небольших мобильных устройств важное значение имеет сокращение количества интерактивных взаимодействий, предла-

гаемых пользователю. Например, если программа будет напоминать пользователю о необходимости сохранить документ в тот самый момент, когда он пытается ответить на телефонный звонок, — это, конечно же, будет неудобно. В Android вообще отсутствует сама концепция документа. Пользователь всегда должен иметь под рукой нужные данные и быть уверенным, что его данные сохранятся в полном порядке.

Чтобы пользователю было легко поэтапно сохранять и использовать данные приложения, компонент за компонентом, и чтобы при этом данные всегда оставались в долговременной (энергонезависимой) памяти без необходимости специально сохранять всю модель данных, Android поддерживает организацию информации, ориентированную на работу с базой данных. Эта поддержка реализуется в классах баз данных, видов и активностей (рис. 9.2). Мы расскажем, как использовать классы баз данных Android для реализации модели такого рода.

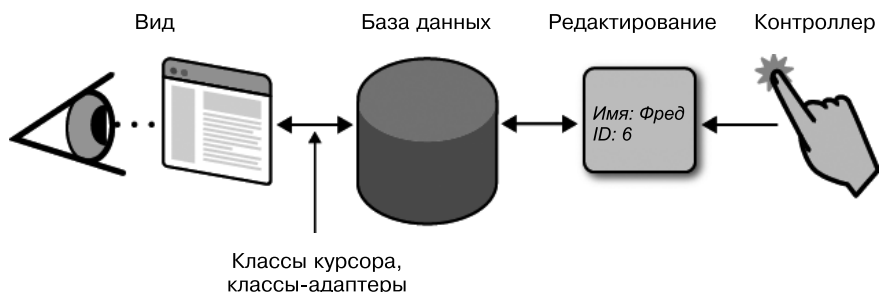


Рис. 9.2. В Android поддерживается модель, где информация сосредоточена в базе данных

Классы базы данных в Android

В этом разделе делается введение в классы Java, обеспечивающие доступ к рассмотренным выше в главе функциям SQLite. При их применении используется только что описанная модель, ориентированная на работу с данными.

- SQLiteDatabase — это интерфейс Android на языке Java, предназначенный для взаимодействия с реляционной базой данных. Он поддерживает достаточно насыщенную реализацию SQL, предоставляющую вам практически все элементы, которые могут понадобиться в мобильном приложении, в том числе возможность работы с курсором.
- Cursor — это контейнер для результатов запроса к базе данных. Курсор поддерживает систему наблюдения, построенную в стиле MVC. Курсоры похожи на результаты запроса из технологии JDBC и представляют собой возвращаемые значения, получаемые от запросов, направленных к базе данных Android. Курсор может представлять много объектов, и для этого ему не требуется экземпляр каждого из этих объектов. Работая с курсором, вы можете перейти в начало списка результатов запроса и при необходимости получать доступ

к любой интересующей вас строке (одна строка за раз). Для доступа к данным курсора вызываются методы, названия которых строятся по принципу `Cursor.getAs*(int columnNumber)` (например, `Cursor.getString()`). Значения, которые будет возвращать курсор, зависят от текущего индекса курсора, который можно при необходимости увеличивать, вызывая `Cursor.moveToNext()`, или уменьшать, вызывая `Cursor.moveToPrevious()`. Текущий индекс курсора можно считать указателем на результирующий объект.

Курсоры — это основополагающая часть модели Android MVC, которую мы подробно рассмотрим в главе 12.

- `SQLiteOpenHelper` — этот класс обеспечивает каркас жизненного цикла для создания и обновления базы данных приложения. Данный класс довольно полезен при решении такой важной задачи, как перенос информации из старой версии приложения в более новую, где, возможно, изменится организация таблиц базы данных.
- `SQLiteQueryBuilder` — этот класс предоставляет высокоуровневую абстракцию для создания запросов `SQLite`, которые затем будут использоваться в приложениях Android. Применяя этот класс, можно упростить написание запросов, так как он избавляет вас от необходимости возиться с самим синтаксисом SQL.

Разработка базы данных для приложений Android

В данном разделе мы рассмотрим часть кода, который подробно описывается в главе 12 и занимается долговременным хранением метаданных, аннотирующих видеофайлы. К этим метаданным относятся заголовок видео, его описание и URL видеофайла. Данный код находится внутри поставщика содержимого Android, который, по нашему мнению, как раз подходит для размещения кода базы данных. В главе 12 будет подробно рассказано, как написать поставщик содержимого. Приведенный ниже код помогает проиллюстрировать, как в Android создается и используется база данных `SQLite`. В этом приложении будет применяться практически такая же база данных, как и в примере, где мы учились работать с инструментом командной строки `sqlite3`. Но на этот раз мы напишем код, в котором для управления данными используется API Android.

Базовая структура класса `SimpleVideoDbHelper`

В нашем примере в файле `SimpleFinchVideoContentProvider.java` заключена вся логика SQL, необходимая для работы с базой данных `simple_video` в Android. Приложения, которым требуется доступ к данным, сохраненным в долговременной памяти этой базы данных, взаимодействуют с поставщиком содержимого и курсором, предоставляемым этим поставщиком. Данный механизм будет объяснен в главе 12.

Клиенты полностью изолированы от детальной информации о том, как именно сохраняются данные. Это хорошая практика программирования, и ее следует придерживаться при написании всех ваших приложений для Android, где будут использоваться базы данных.

В данный момент, поскольку нас интересует именно работа с базами данных в Android, достаточно знать, что SimpleVideoDbHelper — это модель базы данных в поставщике содержимого. Все свойства, специфичные для конкретной базы данных (ее имя, названия ее столбцов и описания ее таблиц), закладываются в этом классе. Разумеется, если мы имеем дело с большой и сложной базой данных, то такой вспомогательный класс также может быть гораздо сложнее и состоять из нескольких компонентов.

Класс SimpleVideoDbHelper наследуется от абстрактного класса SQLiteOpenHelper, следовательно, он должен переопределять методы onCreate и onUpgrade. Метод onCreate вызывается автоматически при первом запуске приложения. Его задача — создать базу данных. Когда появится новая версия приложения, базу данных также, возможно, потребуется обновить, например добавить таблицы, столбцы или вообще изменить схему базы данных. Когда возникает такая необходимость, решением этой задачи занимается метод onUpgrade. Этот метод вызывается всякий раз, когда DATABASE_VERSION в вызове к конструктору отличается от версии, сохраненной в базе данных. Создавая новую версию базы данных, необходимо увеличить номер версии:

```
public static final String VIDEO_TABLE_NAME = "video";

public static final String DATABASE_NAME = SIMPLE_VIDEO + ".db";
private static int DATABASE_VERSION = 2;

public static final int ID_COLUMN = 0;
public static final int TITLE_COLUMN = 1;
public static final int DESCRIPTION_COLUMN = 2;
public static final int TIMESTAMP_COLUMN = 3;
public static final int QUERY_TEXT_COLUMN = 4;
public static final int MEDIA_ID_COLUMN = 5;

private static class SimpleVideoDbHelper extends SQLiteOpenHelper {
    private SimpleVideoDbHelper(Context context, String name,
                                SQLiteDatabase.CursorFactory factory)
    {
        super(context, name, factory, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) {
        createTable(sqLiteDatabase);
    }

    private void createTable(SQLiteDatabase sqLiteDatabase) {
        String qs = "CREATE TABLE " + VIDEO_TABLE_NAME + " (" +
```

```

        FinchVideo.SimpleVideos._ID +
        " INTEGER PRIMARY KEY AUTOINCREMENT, " +
        FinchVideo.SimpleVideos.TITLE_NAME + " TEXT, " +
        FinchVideo.SimpleVideos.DESCRPTION_NAME + " TEXT, " +
        FinchVideo.SimpleVideos.URI_NAME + " TEXT");";
        SQLiteDatabase.execSQL(qs);
    }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase,
        int oldv, int newv)
    {
        sqLiteDatabase.execSQL("DROP TABLE IF EXISTS " +
            VIDEO_TABLE_NAME + ";");
        createTable(sqLiteDatabase);
    }
}

```

Ниже перечислены основные элементы, связанные с кодом SimpleVideoDbHelper.

Константы

В классе SimpleVideoDbHelper определяются две важные константы, а также следующие элементы.

- DATABASE_NAME — в этой константе содержится имя файла базы данных (в нашем случае — simple_video.db). Именно так и называется конкретный файл базы данных SQLite. Как мы уже упоминали, путь к этому файлу выглядит следующим образом: /data/data/com.oreilly.demo.pa.finchvideo/databases/simple_video.db. Система Android сама создаст за вас файл базы данных.
- DATABASE_VERSION — в этой константе определяется версия базы данных, которую вы выбираете произвольно, а затем приращиваете всякий раз, когда изменяете схему базы данных. Если версия базы данных на машине имеет меньший номер, чем DATABASE_VERSION, то система запускает метод onUpgrade, обновляющий базу данных до актуального уровня.
- VIDEO_TABLE_NAME — это имя единственной таблицы, входящей в нашу простейшую базу данных.
- *_NAME — это имена столбцов базы данных. Как было указано выше, необходимо определить столбец с именем _id и использовать его в качестве основного ключа для любой таблицы, к которой вы собираетесь получить доступ при помощи курсора.

Конструктор

Конструктор базы данных, присутствующий в этом поставщике, SimpleVideoDbHelper, использует функцию super для вызова своего родительского конструктора. Родительский конструктор выполняет всю основную работу по созданию объекта базы данных.

- onCreate — когда приложение Android пытается записать данные в несуществующую базу данных или считать данные из несуществующей базы, фреймворк выполняет метод onCreate. Метод onCreate в классе YouTubeDbHelper демонстрирует один из способов создания базы данных. Если для инициализации базы данных требуется существенный объем кода на SQL, то, возможно, будет лучше держать код в файле ресурсов strings.xml. Так можно значительно улучшить читаемость кода Java. Но в таком случае при изменении кода разработчику также придется одновременно иметь дело с двумя разными файлами, чтобы видеть, что на самом деле происходит. Разумеется, если в программе реализована простая база данных, то проще будет писать код SQL прямо в коде Java, как мы и поступили в SimpleVideoDbHelper. Или же, если вы используете построитель запросов, можно вообще обойтись без SQL.



Если вы собираетесь загружать данные SQL из строкового ресурса String, то строку нужно особым образом изменить, а в документации Android об этом моменте упомянуто лишь вскользь. Необходимо экранировать в строке ресурса все одиночные и двойные кавычки обратным слешем (то есть заменить " на \" и ' на \') или заключить всю строку в одиночные или двойные кавычки. Кроме того, следует отключить форматирование строки, воспользовавшись атрибутом formatted="false".

Например:

```
<string name="sql_query" formatted="false">  
    SELECT * FROM videos WHERE name LIKE \"%cycle%\"  
</string>
```

Метод onCreate может и не создавать базы данных. Он передается новоиспеченной совершенно пустой базе данных и должен полностью ее инициализировать. В SimpleVideoDbHelper это довольно простая задача, решаемая при помощи вызова createTable.

- onUpgrade — метод onUpdate для SimpleVideoDbHelper очень прост: он удаляет базу данных. Если поставщик содержимого пытается воспользоваться этой базой данных позже, Android вызывает метод onCreate, так как требуемая база данных не существует. Хотя такой лобовой подход вполне годится для нашего элементарного случая, когда поставщик содержимого является простым кэшем сетевой информации, подобный метод никак не подходит для базы данных со списком контактов! Ваши клиенты определенно будут недовольны, если им придется повторно вбивать всю информацию после каждого обновления прошивки телефона. Поэтому наш метод onUpgrade не особенно применим на практике. Вообще, методу onUpgrade придется распознавать все предыдущие версии базы данных, которые ранее использовались данным приложением, и располагать специальной стратегией преобразования этой информации в наиболее новый формат. В более крупном приложении, скорее всего, будет несколько скриптов обновления, по одному для каждой версии, которая могла устареть к настоящему

времени. Затем приложение последовательно запустит каждый скрипт обновления, чтобы вся база данных стала актуальной.

- `createTable` — эту функцию мы создали, чтобы инкапсулировать в нее весь код SQL, создающий нашу таблицу.

API базы данных на примере MJAndroid

В данном разделе мы покажем более сложное приложение под названием MJAndroid. Эта программа демонстрирует, как использовать небольшую базу данных в гипотетическом приложении, предназначенном для поиска вакансий. Здесь мы исследуем аспекты программы, которые связаны с долговременным хранением данных. В главе 15 мы рассмотрим, как в этом приложении интегрированы картографические функции, позволяющие отобразить на карте результаты поиска вакансий. Сначала расскажем об этом приложении чуть подробнее.

Android и социальные сети

Одно из наиболее грандиозных ожиданий в мире мобильных телефонов Android связано с тем, что на них могут работать приложения, предоставляющие пользователям новые возможности общения в социальных сетях. Эти ожидания отражают текущую реальность и тот этап, который сейчас переживает в своем развитии Интернет. Первое поколение интернет-приложений предназначалось для того, чтобы обеспечивать пользователям доступ к информации, и многие из таких приложений были довольно популярны. Вторая волна интернет-приложений была призвана обеспечить пользователям связь друг с другом. Такие программы, как Facebook, YouTube и т. п., предоставляют возможности найти друзей по интересу. В таких приложениях сами пользователи являются авторами всего или практически всего контента, который есть на сайте. Система Android обладает достаточным потенциалом, чтобы переосмыслить эту концепцию и добавить в нее новую грань — мобильность. Ожидается, что для пользователей мобильных устройств будет написано целое новое поколение приложений: социальные сети, с которыми можно будет работать, просто идя по улице, приложения, которым будет известно, где именно находится пользователь, приложения, которые позволят без труда делиться сложным контентом (например, изображениями, видео и т. д.). MJAndroid — конкретный пример, демонстрирующий, какое место Android собирается занять в этой растущей нише.

С помощью приложения MJAndroid MicroJobs пользователь может найти временную работу в том районе, где живет, позволяющую работать по несколько часов в день и иметь дополнительный заработок. Предполагается, что работодатели, ищущие людей на подработку, вводят на сайт информацию о доступных вакансиях, их описание, часы работы и предлагаемые ставки зарплаты. Вся эта информация должна находиться в расположенной в Вебе базе данных, доступ к которой возможен с мобильных телефонов Android. Те, кто ищет подработку на несколько часов в день, могут входить в эту базу данных при помощи программы MicroJobs, искать работу поблизости от того места, где живут, общаться с друзьями на темы, связан-

ные с работой и потенциальными работодателями, а также звонить непосредственно самому работодателю, если вакансии их заинтересуют. В нашем примере мы не будем создавать специального онлайн-сервиса. У нас на телефоне уже есть специальные искусственные данные, предназначенные для тестирования. Наше приложение имеет несколько функций, дополняющих его основное предназначение определенными чертами, уникальными для мобильных устройств.

- Картография — в среде мобильных телефонов Android поддерживаются динамические интерактивные карты, и мы максимально воспользуемся теми возможностями, которые с ними связаны. В разделе «MapView и MapActivity» главы 15 вы увидите, что, написав совсем немного кода, можно будет отображать на телефоне динамические карты окрестностей, получать обновления информации о местоположении от внутренней GPS-системы, чтобы карта автоматически прокручивалась по мере того, как мы передвигаемся. Мы сможем прокручивать карту в двух направлениях, по вертикали и горизонтали, и даже переключаться в режим просмотра со спутника.
- Поиск друзей и предстоящих событий — опять же в главе 15 мы рассмотрим графический слой, накладываемый на карту (оверлей), который покажет, где поблизости от нашего местоположения предлагается работа. Чтобы подробнее ознакомиться с вакансией, нужно будет просто прикоснуться к соответствующему символу на карте. Мы получим доступ к приложению Android, которое занято управлением контактами, и увидим информацию об адресах наших друзей (а также узнаем их телефонные номера и номера служб мгновенных сообщений). Кроме того, мы сможем обращаться к базе данных MicroJobs, чтобы почитать дополнительное описание предлагаемых вакансий.
- Мгновенные сообщения — когда мы найдем поблизости друзей, с которыми захотим поболтать, мы сможем связаться с ними по службе мгновенных сообщений, в данном случае при помощи SMS.
- Разговор с друзьями или работодателями — если переговоры по ICQ окажутся слишком медленными и затруднительными, то мы сможем позвонить по сотовому друзьям или же работодателю, разместившему вакансию.
- Просмотр Интернета — у большинства работодателей найдется специальный сайт с дополнительной информацией о работе. Мы сможем выбрать работодателя из списка или прямо на карте и быстро заглянуть на сайт. А там уже можно, например, внимательно рассмотреть, в каком месте предлагается работать.

Это интересное приложение вполне можно разрабатывать и дальше, пока оно не достигнет размеров полномасштабного сервиса. Но мы хотим показать, как просто создать и скомбинировать эти многообещающие возможности в нашем собственном приложении. Как и весь код из этой книги, код данного приложения можно скачать на сайте, и мы настоятельно рекомендуем вам это сделать. Если код будет у вас под рукой, в него можно будет в любой момент заглянуть и уточнить какие-нибудь детали. Кроме того, будет просто вырезать фрагменты кода и вставлять их в ваше приложение по мере чтения книги. Пока мы будем использовать MJAndroid как практически «боевой» пример, который поможет нам лучше разбираться в API Android, предназначенном для работы с базой данных.

На рис. 9.3 показано, как выглядит на экране приложение MJAndroid, когда вы его только запускаете. Это карта окрестностей, на которую наложено несколько кнопок и указателей.

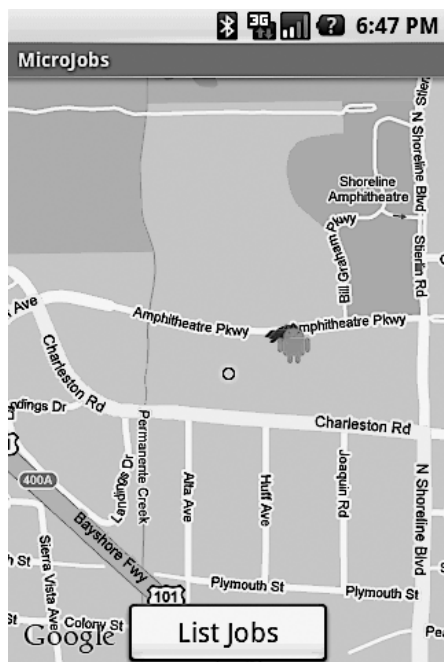


Рис. 9.3. Исходный скриншот приложения MJAndroid

Исходный каталог (src)

Имя пакета с MJAndroid — `com.microjobsinc.mjandroid`. Eclipse моделирует аналогичную структуру каталогов, как и для любого проекта на языке Java, и показывает вам весь проект, когда вы открываете каталоги `src`. Кроме этих каталогов с пакетами, есть еще один каталог с таким же именем, как и пакет, в котором находятся все файлы Java для данного проекта. В нем есть следующие файлы.

- `MicroJobs.java` — основной файл с исходным кодом приложения. Эта активность запускается первой и отображает карту, являющуюся центральным элементом приложения, и по мере необходимости вызывает другие активности и сервисы для реализации различных функций пользовательского интерфейса.
- `MicroJobsDatabase.java` — вспомогательный класс базы данных, обеспечивающий легкий доступ к локальной базе данных MJAndroid. Именно здесь при помощи SQLite сохраняется вся информация о работодателе, пользователе и самой работе.
- `AddJob.java` и `EditJob.java` — относятся к работе с базой данных MJAndroid. Эти файлы обеспечивают отображение таких экранов, где пользователь может до-

бавлять новую информацию в базу данных или редактировать имеющуюся информацию.

- `MicroJobsDetail.java` — активность, отображающая подробную информацию о конкретной вакансии.
- `MicroJobsEmpDetail.java` — активность, показывающая подробную информацию о работодателе, в том числе его имя, адрес, репутацию, адрес электронной почты, телефонный номер и т. д.
- `MicroJobsList.java` — активность, отображающая список работ (в отличие от картографического вида в файле `MicroJobs.java`). Здесь мы видим обычный список работодателей и вакансий. Пользователь может отсортировать список по любому полю, а также просмотреть подробную информацию о вакансии или работодателе — для этого нужно просто прикоснуться к конкретной записи на экране.

Загрузка и запуск приложения

Запуск `MJAndroid` из SDK осложняется тем, что в данном приложении используется `MapView`. Для работы с `MapView` Android требуется специальный ключ к картографическому API, и такой ключ жестко привязывается к той конкретной машине, на которой вы разрабатываете приложение. В подразделе «Подписывание приложения» раздела «Подписывание приложения» главы 4 мы говорили о требованиях, предъявляемых к подписыванию и запуску вашей программы. И поскольку работа данного приложения зависит от картографического API, в этом примере нам нужно настроить ключ к API, чтобы программа работала правильно. Для запуска `MJAndroid` просто запустите в Eclipse проект для этой главы — раньше мы так уже поступали.

Запросы к базе данных и считывание информации из базы данных

Существует много способов считывания информации из базы данных SQL, но все они сводятся к базовой последовательности операций.

1. Создание предложения SQL, описывающего данные, которые вам требуется получить.
2. Применение этого предложения к базе данных.
3. Отображение результирующих данных SQL на такие структуры данных, которые будут понятны языку, с которым вы работаете.

Этот процесс может быть очень сложен при применении специальных программ объектно-реляционного отображения или относительно прост, но трудоемок при написании запросов внутри вашего приложения. Инструменты для объектно-реляционного отображения (object relational mapping, ORM, <http://ru.wikipedia.org/wiki/ORM>) защищают код от сложностей, связанных с программированием баз данных и отображением (ассоциированием) объектов. Эти проблемы не исчезают, а просто

не отвлекают вас от работы. Вы можете достичь большей отказоустойчивости кода в том, что касается изменений базы данных, но для этого придется произвести сложную настройку объектно-реляционного отображения, а затем заниматься поддержкой этой системы. В настоящее время объектно-реляционное отображение в программах Android, как правило, не применяется.

Простой подход с написанием запросов в вашем приложении хорош лишь для очень небольших проектов, которые не будут особенно меняться с течением времени. Приложения, прямо в коде которых содержится код базы данных, характеризуются дополнительной неустойчивостью, поскольку, когда схема базы данных изменяется, весь код, который ссылается на эту базу данных, должен быть проверен и, возможно, переписан.

Обычный компромиссный подход заключается в том, что вся логика базы данных извлекается в множество объектов, единственная цель которых — преобразовывать запросы приложения в запросы к базе данных и отправлять полученные результаты обратно в приложение. Именно такой подход мы применили в приложении MJAndroid. Весь код базы данных содержится в одном классе `MicroJobsDatabase`, который также дополняет `SQLiteOpenHelper`. Но с применением поставщика содержимого `SimpleFinchVideoContentProvider` база данных остается достаточно простой и нам не приходится прибегать к использованию внешних строк.

Android позволяет модифицировать курсоры, если они не применяются с поставщиками содержимого. Мы обязательно воспользуемся этой возможностью, чтобы еще сильнее уменьшить зависимости кода и скрыть всю информацию о каждой конкретной операции базы данных в нашем модифицированном курсоре. Интерфейс для вызывающей стороны в методе `getJobs` класса `MicroJobsDatabase` впервые появляется в том коде, который мы приведем ниже. Задача метода заключается в том, чтобы вернуть `JobsCursor`, в котором перечислены вакансии из базы данных. Один параметр, передаваемый методу `getJobs`, позволяет пользователю выбрать, как отсортировать полученные вакансии — по столбцу `title` или по столбцу `employer_name`:

```
public class MicroJobsDatabase extends SQLiteOpenHelper {
    ...
    /** Возвратить отсортированный JobsCursor
     * @param sortBy критерии сортировки
     */
    public JobsCursor getJobs(JobsCursor.SortBy sortBy) { ❶
        String sql = JobsCursor.QUERY + sortBy.toString(); ❷
        SQLiteDatabase d = getReadableDatabase(); ❸
        JobsCursor c = (JobsCursor) d.rawQueryWithFactory( ❹
            new JobsCursor.Factory(),
            sql,
            null,
            null);
        c.moveToFirst(); ❺
        return c; ❻
    }
    ...
    public static class JobsCursor extends SQLiteCursor{ ❼
        public static enum SortBy{ ❽
```

```

        title,
        employer_name
    }
    private static final String QUERY =
        "SELECT jobs._id, title, employer_name, latitude,
            longitude, status "+
        "FROM jobs, employers "+
        "WHERE jobs.employer_id = employers._id "+
        "ORDER BY ";
    private JobsCursor(SQLiteDatabase db, SQLiteCursorDriver driver,
        String editTable, SQLiteQuery query) { ❹
        super(db, driver, editTable, query);
    }
    private static class Factory implements
        SQLiteDatabase.CursorFactory{ ❺
        @Override
        public Cursor newCursor(SQLiteDatabase db,
            SQLiteCursorDriver driver, String editTable,
            SQLiteQuery query) { ❻
            return new JobsCursor(db, driver, editTable, query); ❼
        }
    }
    public long getColJobsId(){ ❸
        return getLong(getColumnIndexOrThrow("jobs._id"));
    }
    public String getColTitle(){
        return getString(getColumnIndexOrThrow("title"));
    }
    public String getColEmployerName(){
        return getString(getColumnIndexOrThrow("employer_name"));
    }
    public long getColLatitude(){
        return getLong(getColumnIndexOrThrow("latitude"));
    }
    public long getColLongitude(){
        return getLong(getColumnIndexOrThrow("longitude"));
    }
    public long getColStatus(){
        return getLong(getColumnIndexOrThrow("status"));
    }
}

```

Далее даются пояснения к коду.

- ❶ Функция, формирующая запрос в зависимости от того, какую сортировку столбцов заказал пользователь (то есть в зависимости от параметра `sortBy`). Эта же функция возвращает результаты в виде курсора.
- ❷ Создание строки запроса. Большинство строк являются статическими (переменная `QUERY`), но эта строка зависит от выбора столбца, по которому происходит сортировка. Хотя `QUERY` и является закрытым полем, обрамляющий класс по-прежнему имеет доступ к ней. Это объясняется тем, что и метод `getJobs`, и класс

JobsCursor находятся внутри класса MicroJobsDatabase, который и предоставляет доступ к закрытым элементам JobsCursor для метода getJobs.

Чтобы получить текст для сортировочного столбца, мы просто применяем toString к перечислимому параметру, переданному вызывающей стороной. Мы могли бы определить ассоциативный массив, который позволил бы нам более свободно именовать переменные, но приведенное выше решение проще. Кроме того, названия столбцов достаточно красиво всплывают на экране — это делается благодаря автозавершению, предусмотренному в нашей интегрированной среде разработки.

- 3 Получает описатель базы данных.
- 4 Создает курсор JobsCursor при помощи метода rawQueryWithFactory класса SQLiteDatabase. Он принимает фабричный метод, которым Android воспользуется, чтобы создать курсор именно такого типа, какой нам нужен. Если бы мы воспользовались более простым методом rawQuery, то получили бы стандартный Cursor, у которого нет особых свойств JobsCursor.
- 5 Для удобства вызывающей стороны переходим к первой строке в возвращенном результате. Таким образом, курсор возвращается уже готовым к использованию. Часто программист забывает вызвать moveToFirst, а потом рвет на себе волосы, пытаясь понять, почему объект Cursor выдает сплошные исключения.
- 6 Курсор — это возвращаемое значение.
- 7 Класс, создающий курсор, возвращаемый методом getJobs.
- 8 Простой способ предоставить альтернативные критерии сортировки: имена столбцов сохраняются в enum. Этот тип используется в элементе 2.
- 9 Конструктор для модифицированного курсора. Последний аргумент — это запрос, передаваемый вызывающей стороной.
- 10 Класс фабрики для создания курсора, вложен в класс JobsCursor.
- 11 Создается курсор из запроса, переданного вызывающей стороной.
- 12 Возвращает курсор к внешнему классу JobsCursor.
- 13 Вспомогательные функции, извлекающие конкретные столбцы из строки курсора. Например, getColTitle возвращает значение столбца title в строке, на которую в данный момент ссылается курсор. Таким образом, реализация базы данных отделяется от вызывающего кода и код становится проще читать.



Хотя создание подклассов курсора — хороший прием, позволяющий использовать базу данных в рамках одного приложения, он не сработает с API поставщиков содержимого, так как в Android не предусмотрен способ совместного использования подклассов курсора несколькими процессами. Кроме того, программа MJAndroid — это искусственный пример, который всего лишь демонстрирует, как работать с базой данных. В главе 13 мы покажем рабочее приложение с более надежной архитектурой.

Далее приведен пример использования базы данных. Код получает курсор, информация в котором отсортирована по заголовку. Это делается путем вызова getJobs. Потом происходит итерация процесса для каждой вакансии:

```
MicroJobsDatabase db = new MicroJobsDatabase(this); ❶
JobsCursor cursor = db.getJobs(JobsCursor.SortBy.title); ❷

for (int rowNum = 0; rowNum < cursor.getCount(); rowNum++) { ❸
    cursor.moveToPosition(rowNum);
    doSomethingWith(cursor.getColTitle()); ❹
}
```

Рассмотрим пояснения к коду.

- ❶ Создание объекта `MicroJobsDatabase`. Аргумент `this` представляет контекст, как это было описано выше.
- ❷ Создание курсора `JobsCursor`, использующего перечисление `SortBy`, рассмотренное выше.
- ❸ Использование стандартных методов `Cursor` для итерации через курсор.
- ❹ Все еще в рамках цикла вызывается один из специальных методов, предоставляемый `JobsCursor` для того, чтобы «что-нибудь сделать» на выбор пользователя. Метод будет вызван для каждого значения столбца `title`.

Использование метода `query`

В то время как в приложениях, выполняющих нетривиальные операции с базами данных, полезно изолировать предложения SQL, как было показано выше, для приложений, которые рассчитаны на простые операции с базами данных (например, для нашего `SimpleFinchVideoContentProvider`), не менее полезно использовать метод `SQLiteDatabase.query`. Рассмотрим следующий пример, относящийся к `video`:

```
videoCursor = mDb.query(VIDEO_TABLE_NAME, projection,
    where, whereArgs,
    null, null, sortOrder);
```

Как и в случае с `SQLiteDatabase.rawQueryWithFactory`, показанным выше, объект `Cursor` является возвращаемым значением метода `query`. Здесь мы присваиваем данный курсор переменной `videoCursor`, определенной выше.

Метод `query` применяет команду `SELECT` к таблице с заданным именем. В нашем случае имя таблицы хранится в константе `VIDEO_TABLE_NAME`. Данный метод принимает два параметра. Во-первых, это проекция, где перечисляются столбцы, которые только и должны отображаться в запросе, — значения других столбцов не появятся и в результатах курсора. Многие приложения отлично работают, просто передавая `null` вместо проекции, и в таком случае в результирующем курсоре отобразятся все столбцы. Далее, аргумент `where` содержит условие SQL `where` без ключевого слова `WHERE`. Кроме того, аргумент `where` может включать в себя ряд строк `'?'`, которые будут заменяться значениями `whereArgs`. При обсуждении метода `execSQL` мы более подробно поговорим о том, как связываются эти два значения.

Изменение базы данных

Курсоры Android отлично подходят для считывания информации из базы данных, но в классе `android.database.Cursor` нет методов для создания, обновления

или удаления данных. В классе `SQLiteDatabase` предоставляется два базовых API, которые можно использовать как для считывания, так и для записи.

- Набор из четырех методов, называемых `insert`, `query`, `update` и `delete`.
- Более общий метод `execSQL`, принимающий каждое отдельное предложение SQL, которое не занимается возвратом данных, и применяющий это предложение к базе данных.

Рекомендуем пользоваться первыми четырьмя методами, когда требуется производить одноименные им операции (вставку, запрос, обновление и удаление). Мы покажем вам оба способа использования операций в MJAndroid.

Вставка информации в базу данных

Предложение SQL `INSERT` используется всякий раз, когда вы хотите вставить информацию в базу данных SQL. Предложение `INSERT` соответствует операции «создать» в методологии CRUD.

В приложении MJAndroid пользователь может добавлять вакансии в список, щелкая на элементе меню **Add Job** (Добавить вакансию) при просмотре списка вакансий. Затем пользователь может заполнить форму, в которой указывает работодателя, название вакансии и ее описание. После того как пользователь нажмет в форме кнопку **Add Job** (Добавить вакансию), выполнится следующая строка кода:

```
db.addJob(employer.id, txtTitle.getText().toString(),
txtDescription.getText().toString());
```

Этот код вызывает функцию `addJob`, передавая **ID работодателя, название вакансии и описание работы**. Функция `addJob` выполняет всю работу по записи вакансии в базу данных.

Использование метода `insert`. В следующем примере демонстрируется использование метода `insert`:

```
/**
 * Добавление в базу данных новой вакансии.
 * Вакансия будет считаться открытой.
 * @param employer_id работодатель, разместивший вакансию
 * @param title название вакансии
 * @param description описание работы
 */
public void addJob(long employer_id, String title, String description) {
    ContentValues map = new ContentValues(); ❶
    map.put("employer_id", employer_id);
    map.put("title", title);
    map.put("description", description);
    try{
        getWritableDatabase().insert("jobs", null, map); ❷
    } catch (SQLException e) {
        Log.e("Error writing new job", e.toString());
    }
}
```

Пояснения к коду следующие.

- ❶ Объект `ContentValues` отображает имена столбцов на значения столбцов. Внутри системы он реализуется как `HashMap<String, Object>`. Однако в отличие от обычных `HashMap` `ContentValues` обладает строгой типизацией. Можно указать тип данных для каждого значения, сохраненного в контейнере `ContentValues`. Когда вы получаете значения обратно, `ContentValues` автоматически преобразует значения в требуемый тип, если это возможно.
- ❷ Второй параметр метода `insert` — `nullColumnHack`. Он используется только в качестве значения по умолчанию, когда третий параметр (ассоциативный контейнер, `map`) равен `null`, следовательно, в отсутствие значения по умолчанию строка будет совершенно пуста.

Использование метода `execSQL`. Данное решение является более низкоуровневым, чем решение с методом `insert`. Здесь создается **SQL, и этот код отправляется для исполнения** в библиотеку. Хотя можно жестко закодировать все предложения, в том числе данные, передаваемые пользователем, мы продемонстрируем предпочтительный метод, при котором задействуются параметры связывания (`bind parameters`).

Параметр связывания обозначается знаком `?`, который занимает место в предложении **SQL**, обычно там, где должен стоять параметр, переданный пользователем, например значение в условии `WHERE`. После создания предложения **SQL с параметрами** связывания его можно использовать многократно, задавая точное значение параметров связывания всякий раз перед тем, как выполнить это предложение:

```
/**
 * Добавление в базу данных новой вакансии.
 * Вакансия будет считаться открытой.
 * @param employer_id работодатель, разместивший вакансию
 * @param title          название вакансии
 * @param description описание работы
 */
public void addJob(long employer_id, String title, String description){
    String sql =
        "INSERT INTO jobs " +
        "(_id, employer_id, title, description, start_time, end_time, " +
        "status) " +
        "VALUES " +
        "(NULL, ?, ?, ?, 0, 0, 3)";
    Object[] bindArgs = new Object[]{employer_id, title, description};
    try{
        getWritableDatabase().execSQL(sql, bindArgs);
    } catch (SQLException e) {
        Log.e("Error writing new job", e.toString());
    }
}
```

Рассмотрим пояснения к коду.

- ❶ Построение шаблона **SQL-запроса** под названием `sql`, содержащего пригодные для связывания параметры, которые будут заполняться пользовательскими данными. Параметры, пригодные для связывания, обозначаются в строке знаком `?`.

Далее мы строим массив объектов под названием `bindArgs`, в котором содержится по объекту на каждый элемент нашего **SQL-шаблона**. В шаблоне три вопросительных знака, поэтому в массиве объектов должно быть три элемента.

- ❷ Выполнение команды SQL путем передачи строки шаблона SQL и аргументов связывания к `execSQL`.

Применение шаблона SQL и аргументов связывания — более предпочтительный способ по сравнению с построением предложения SQL со всеми параметрами в виде `String` или `StringBuilder`. При использовании шаблона с параметрами вы защищаете свое приложение от внедрения (инъекции) SQL-кода. Такие атаки происходят, когда пользователь-злоумышленник пытается ввести информацию в форму и эта информация намеренно предназначена для изменения базы данных таким способом, которого не предусматривал разработчик. Как правило, для этого взломщики вводят синтаксические символы, чтобы преждевременно завершить команду SQL, выполняемую в настоящий момент, и сразу за ними вводят в поле формы новые команды SQL. Кроме того, подход «шаблон-плюс-параметры» защищает вас и от рядовых ошибок, например от попадания неверных символов в параметры. К тому же такой подход помогает держать код более чистым и не вводить длинные последовательности прикрепляемых вручную строк, автоматически заменяя вопросительные знаки.

Обновление данных, уже имеющих в базе

Пользователь приложения `MicroJobs` может отредактировать вакансию, щелкнув на ней в списке и выбрав в меню элемент `Edit Job` (Изменить вакансию). Затем пользователь может изменять строки, описывающие работодателя, вакансию и подробности о ней. Это делается в форме `editJob`. После того как пользователь нажмет в форме кнопку `Update` (Обновить), выполнится следующая строка кода:

```
db.editJob((long)job_id, employer.id, txtTitle.getText().toString(),
    txtDescription.getText().toString());
```

Этот код вызывает функцию `editJob`, передавая ID вакансии и три элемента, которые пользователь может изменить: **ID работодателя, название вакансии и описание работы**. Функция `editJob` выполняет всю работу по изменению информации о вакансии в базе данных.

Использование метода `update`. В следующем примере демонстрируется использование метода `update`:

```
/**
 * Обновление информации о вакансии в базе данных.
 * @param job_id      идентификатор имеющейся вакансии
 * @param employer_id работодатель, предлагающий вакансию
 * @param title       название вакансии
 * @param description описание работы
 */
public void editJob(long job_id, long employer_id, String title,
    String description)
{
    ContentValues map = new ContentValues();
    map.put("employer_id", employer_id);
```

```

map.put("title", title);
map.put("description", description);
String[] whereArgs = new String[]{Long.toString(job_id)};
try{
    getWritableDatabase().update("jobs", map, "_id=?", whereArgs); ❶
} catch (SQLException e) {
    Log.e("Error writing new job", e.toString());
}
}

```

Пояснение к коду следующее.

- ❶ Первый параметр `update` — это имя таблицы, которой мы собираемся манипулировать. Второй параметр — отображение названий столбцов на новые значения. Третий — небольшой фрагмент кода на SQL. В данном случае это SQL-шаблон с одним параметром. Параметр отмечен вопросительным знаком и заполняется содержимым вашего четвертого аргумента.

Использование метода `execSQL`. В следующем примере демонстрируется использование метода `execSQL`:

```

/**
 * Обновление вакансии в базе данных.
 * @param job_id идентификатор имеющейся вакансии
 * @param employer_id работодатель, предлагающий вакансию
 * @param title название вакансии
 * @param description описание работы
 */
public void editJob(long job_id, long employer_id, String title, String description)
{
    String sql =
        "UPDATE jobs " +
        "SET employer_id = ?, "+
        " title = ? "+
        " description = ? "+
        "WHERE _id = ? ";
    Object[] bindArgs = new Object[]{employer_id, title, description,
                                     job_id};
    try{
        getWritableDatabase().execSQL(sql, bindArgs);
    } catch (SQLException e) {
        Log.e("Error writing new job", e.toString());
    }
}

```

Данная функция — наиболее простая, которую можно использовать в этом примере. При чтении книги ее легко понять, но для реального приложения такая функция все же не подойдет. Дело в том, что в реальном приложении придется проверять строки ввода на присутствие недопустимых символов, проверять, существует ли вакансия, которую вы собираетесь обновить, уточнять валидность значения `employer_id` перед тем, как его использовать, лучше организовать выявление ошибок и т. д. Кроме того, по всей видимости, в любом приложении, которое совместно применяется многочисленными пользователями, потребуется их аутентификация.

Удаление информации из базы данных

Программа MicroJobs позволяет пользователям не только создавать и изменять записи о вакансиях, но и удалять их. В основном интерфейсе приложения пользователь нажимает кнопку **List Jobs** (Составить список вакансий), чтобы вывести такой список, а потом нажимает конкретную вакансию, чтобы просмотреть о ней подробную информацию. На этом уровне пользователь может выбрать команду **Delete this job** (Удалить вакансию). При нажатии кнопки **Delete** (Удалить) в файле `MicroJobsDetail.java` выполняется следующая строка кода:

```
db.deleteJob(job_id);
```

Код вызывает метод `deleteJob` класса `MicroJobsDatabase`, сообщая ID вакансии, которую необходимо удалить. Код напоминает функции, уже рассмотренные нами выше, ему также не хватает некоторых практически важных черт.

Использование метода `delete`. В следующем примере демонстрируется применение метода `delete`:

```
/**
 * Удаление вакансии из базы данных.
 * @param job_id идентификатор вакансии, которую следует удалить
 */
public void deleteJob(long job_id) {
    String[] whereArgs = new String[]{Long.toString(job_id)};
    try{
        getWritableDatabase().delete("jobs", "_id=?", whereArgs);
    } catch (SQLException e) {
        Log.e("Error deleteing job", e.toString());
    }
}
```

Использование метода `execSQL`. В следующем примере демонстрируется применение метода `execSQL`:

```
/**
 * Удаление вакансии из базы данных.
 * @param job_id идентификатор вакансии, которую следует удалить
 */
public void deleteJob(long job_id) {
    String sql = String.format(
        "DELETE FROM jobs " +
        "WHERE _id = '%d' ",
        job_id);
    try{
        getWritableDatabase().execSQL(sql);
    } catch (SQLException e) {
        Log.e("Error deleteing job", e.toString());
    }
}
```

ЧАСТЬ III

Скелет приложения Android

Глава 10. Каркас работоспособного приложения

Глава 11. Создание пользовательского интерфейса

Глава 12. Использование поставщиков содержимого

Глава 13. Поставщики содержимого как фасад для веб-сервисов
RESTful

В первых двух частях книги были описаны основные архитектурные проблемы, с которыми приходится сталкиваться в приложениях Android. **Скелетное приложение**, рассмотренное в части III, демонстрирует описанные выше подходы на практике. Этот код вы можете использовать как отправную точку для ваших собственных приложений.

10 Каркас работоспособного приложения

В этой и следующей главах будет сделано введение в каркасное, или скелетное, приложение, которое служит примером практического использования многих подходов к проектированию и реализации программ, описанных в книге. Особенно широко здесь отражен материал главы 3, где мы говорили о компонентах приложения.

Каркасное приложение, рассмотренное в данной главе, вы можете использовать в качестве отправной точки для написания собственных приложений. Мы рекомендуем создавать программы именно на основе такого каркаса, а не с чистого листа и не с мелких примеров. Дело в том, что в небольшом примере могут быть не реализованы все аспекты объекта `Activity` и жизненного цикла процесса.

Подход, которым мы пользуемся в этой главе, поможет нам визуализировать и понять жизненный цикл компонента, прежде чем он нам действительно понадобится. Подгонка жизненного цикла под приложение, которое было написано без понимания жизненного цикла как такового либо в заблуждении, что обработка жизненного цикла вообще не понадобится, — верный способ написать такую программу Android, которая будет неожиданно выходить из строя. При этом точно воспроизвести условия, спровоцировавшие отказ, будет очень нелегко, а в самой программе будут сохраняться стойкие ошибки, которые могут оставаться скрытыми очень долго, несмотря на любые попытки их искоренить. Иными словами, тяжело в учении — легко в бою.

Хотя эта глава и не рассказывает непосредственно о пользовательских интерфейсах, не забывайте, что классы для пользовательских интерфейсов Android разрабатывались с учетом и тех ограничений, которые накладывает сама архитектура Android, и тех **возможностей, которыми эта операционная система располагает**. Реализации пользовательского интерфейса и обработка жизненного цикла неразрывно связаны. Правильная обработка жизненных циклов приложения, процесса, содержащего приложение, объектов `Activity`, содержащих пользовательский интерфейс приложения, и объектов `Fragment`, которые могут иметься в экземпляре `Activity`, — вот что нужно для обеспечения хорошего пользовательского взаимодействия.

Чтобы сохранить на компьютере код каркасного приложения, о котором мы будем говорить, его нужно скачать в архиве по ссылке Examples (Примеры) на сайте книги — <https://github.com/bmeike/ProgrammingAndroid2Examples.git>. Находящийся здесь код может содержать больше функций, кроме того, в нем могут быть исправлены ошибки.

Визуализация жизненных циклов

Выше в книге и, возможно, в документации по разработке для Android вам уже встречались схематические изображения различных аспектов жизненного цикла компонентов. Там же мы читали о том, как строится жизненный цикл. Но вот в чем проблема, связанная с этими описаниями: жизненные циклы компонентов динамичны, а схема — это неподвижная картинка. Более того, переходы жизненных циклов компонентов и процессов зависят от управления памятью. Когда память начинает заканчиваться, в жизненных циклах компонентов начинают происходить операции, призванные восстановить память. Распределение памяти, сборка мусора и способ, которым в Android обеспечивается восстановление памяти для охвата процессов, — феномены, по сути своей не настолько детерминированные, как выполнение блока кода. Эти феномены зависят от конфигурации. Здесь, выстраивая и выполняя код, мы увидим, как именно протекают жизненные циклы приложения, и поэкспериментируем с ними прямо в действующей программе.

Визуализация жизненного цикла активности

Мы нагляднее представим вам жизненный цикл компонента Activity, запустив специально созданную программу и изучив, как работают методы жизненного цикла Activity в виде LogCat программы Eclipse. Следующий код — это листинг подкласса Activity. В этом подклассе реализованы методы жизненного цикла, а в каждом методе присутствуют вызовы регистрации. Выноски в коде аннотируют подробное описание обработки жизненного цикла — это описание начинается в пункте «Методы жизненного цикла класса Activity» далее. Рассмотрим этот листинг и изучим, какая информация будет регистрироваться:

```
package com.oreilly.demo.pa.ch10.finchlifecycle;
```

```
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
```

```
public class FinchLifecycle extends Activity {
```

```
    // Создание строк для регистрации
    private final String TAG = this.getClass().getSimpleName();
    private final String RESTORE = ", can restore state";
```

```
    // Строка "fortytwo" используется в качестве примера состояния
```

```
private final String state = "fortytwo";

@Override
public void onCreate(Bundle savedInstanceState) { ❶
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    String answer = null;
    // savedInstanceState не может равняться null
    if (null != savedInstanceState) {
        answer = savedInstanceState.getString("answer");
    }
    Log.i(TAG, "onCreate"
        + (null == savedInstanceState ? "" : (RESTORE + " " + answer)));
}

@Override
protected void onRestart() { ❷
    super.onRestart();
    // Уведомление о том, что будет запущена активность
    Log.i(TAG, "onRestart");
}

@Override
protected void onStart() { ❸
    super.onStart();
    // Уведомление о том, что активность запускается
    Log.i(TAG, "onStart");
}

@Override
protected void onResume() { ❹
    super.onResume();
    // Уведомление о том, что активность
    // будет взаимодействовать с пользователем
    Log.i(TAG, "onResume");
}

protected void onPause() { ❺
    super.onPause();
    // Уведомление о том, что активность прекращает
    // взаимодействовать с пользователем
    Log.i(TAG, "onPause" + (isFinishing() ? " Finishing" : ""));
}

@Override
protected void onStop() { ❻
    super.onStop();
    // Уведомление о том, что активность больше не видима
```

```

        Log.i(TAG, "onStop");
    }

    @Override
    protected void onDestroy() { ❷

        super.onDestroy();
        // Уведомление о том, что активность будет удалена
        Log.i(TAG,
            "onDestroy "
            // регистрация любых изменений конфигурации,
            // если таковые возникнут
            + Integer.toString(getChangingConfigurations(), 16));
    }

    //////////////////////////////////////
    // вызывается во время жизненного цикла, когда состояние экземпляра
    // следует сохранить или восстановить
    //////////////////////////////////////

    @Override
    protected void onSaveInstanceState(Bundle outState) { ❸
        // сохранение состояния экземпляра
        outState.putString("answer", state);
        super.onSaveInstanceState(outState);
        Log.i(TAG, "onSaveInstanceState");
    }

    @Override
    public Object onRetainNonConfigurationInstance() { ❹

        Log.i(TAG, "onRetainNonConfigurationInstance");
        return new Integer(getTaskId());
    }

    @Override
    protected void onRestoreInstanceState(Bundle savedInstanceState) { ❺
        super.onRestoreInstanceState(savedInstanceState);
        // Восстановление состояния; нам известно,
        // что savedInstanceState не равно null
        String answer = null != savedInstanceState ?
            savedInstanceState.getString("answer") : "";
        Object oldTaskObject = getLastNonConfigurationInstance();
        if (null != oldTaskObject) {
            int oldtask = ((Integer) oldTaskObject).intValue();
            int currentTask = getTaskId();
            // Задача не должна изменяться при изменениях конфигурации
            assert oldtask == currentTask;
        }
        Log.i(TAG, "onRestoreInstanceState"

```

```

        + (null == savedInstanceState ? "" : RESTORE) + " " + answer);
    }

    //////////////////////////////////////////////////
    // Это мелкие методы жизненного цикла, которые вам,
    // возможно, не понадобятся
    //////////////////////////////////////////////////

    @Override
    protected void onCreate(Bundle savedInstanceState) { ❾

        super.onCreate(savedInstanceState);
        String answer = null;
        // savedInstanceState может быть равно null
        if (null != savedInstanceState) {
            answer = savedInstanceState.getString("answer");
        }
        Log.i(TAG, "onPostCreate"
            + (null == savedInstanceState ? "" : (RESTORE + " " + answer)));
    }

    @Override
    protected void onResume() { ❿

        super.onResume();
        Log.i(TAG, "onPostResume");
    }

    @Override
    protected void onPause() { ⓫

        super.onPause();
        Log.i(TAG, "onPostPause");
    }

    @Override
    protected void onStop() { ⓬

        super.onStop();
        Log.i(TAG, "onPostStop");
    }

    @Override
    protected void onDestroy() { ⓭

        super.onDestroy();
        Log.i(TAG, "onPostDestroy");
    }
}

```

Когда вы готовы запустить приложение, сначала отобразите вид **LogCat**, выполнив команду **Window ▸ Show View ▸ Other** (Окно ▸ Отобразить вид ▸ Прочие), и откройте каталог **Android** в диалоговом окне **Show View** (Отобразить вид). Затем выберите **LogCat** (рис. 10.1).

Теперь запустите приложение в эмуляторе или на физическом устройстве. Поскольку пример из этой главы построен с применением интерфейсов **Fragment API** и **Android API** версии 11, соответствующих версии операционной системы **Android 3.0 Honeycomb**, а также с применением класса **Fragment** из пакета поддержки **Android**, вы можете запускать пример при помощи любой из этих баз кода.

В виде **LogCat** в программе **Eclipse** начнет появляться регистрационная информация. Чтобы отобразить только ту регистрационную информацию, которая

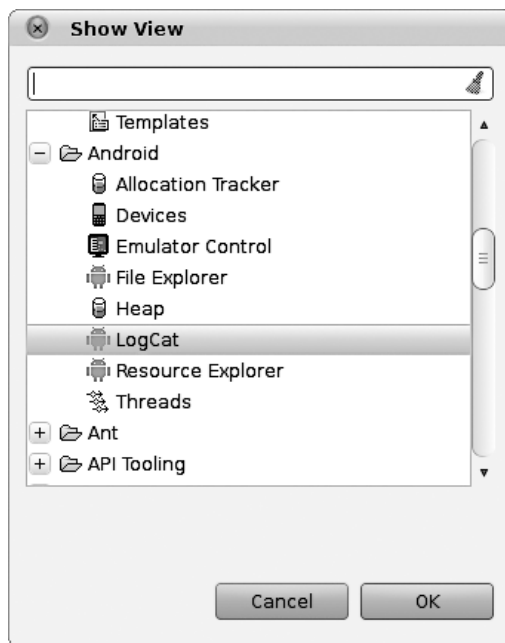


Рис. 10.1. Выбор LogCat из показанного списка

относится к коду, приведенному в предыдущем листинге, общий объем информации можно отфильтровать. Нажмите зеленый символ «+» на панели инструментов в окне регистрации. Откроется диалоговое окно для определения фильтра регистрационной информации (рис. 10.2).

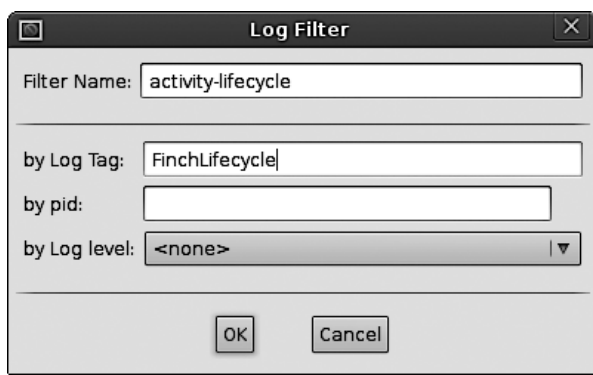


Рис. 10.2. Создание фильтра, который позволяет отобразить только те данные из журнала регистрации, которые отмечены как FinchLifecycle

В данном случае мы собираемся отфильтровать журнал регистрации на основании метки, которой мы воспользовались в классе `FinchLifecycle`. Эта метка на-

зывается так же, как и класс: `FinchLifecycle`. Фильтр мы назовем `activity-lifecycle` (жизненный цикл активности), как показано на рис. 10.2.

Теперь при запуске программы вы увидите только ту регистрационную информацию, которая относится к методам жизненного цикла активности. Эта информация будет содержаться на вкладке `activity-lifecycle` в виде LogCat. Если вы хотите видеть всю регистрационную информацию, то на вкладке Log (Журнал) будет отображаться журнал без фильтрации.

Если запустить программу в эмуляторе Android 3.0, вы увидите примерно то же, что и на рис. 10.3.



Рис. 10.3. Код примера этой главы, запущенный в эмуляторе Android 3.0



Здесь мы будем пользоваться именно Android 3.0, так как в этой главе речь пойдет о жизненных циклах и о классе `Fragment`. Если вы хотите запустить пример на устройстве или эмуляторе, имеющем более раннюю версию, чем Android 3.0, то можно воспользоваться примером, адаптированным для такой версии. Для подобного переноса применяется пакет совместимости Android, позволяющий задействовать `Fragment` и другие классы из Android API уровня 11 вплоть до API уровня 4, который соответствует Android 1.6.

Первое, что вы увидите на вкладке `activity-lifecycle` в виде LogCat, — несколько сообщений из журнала регистрации (рис. 10.4).

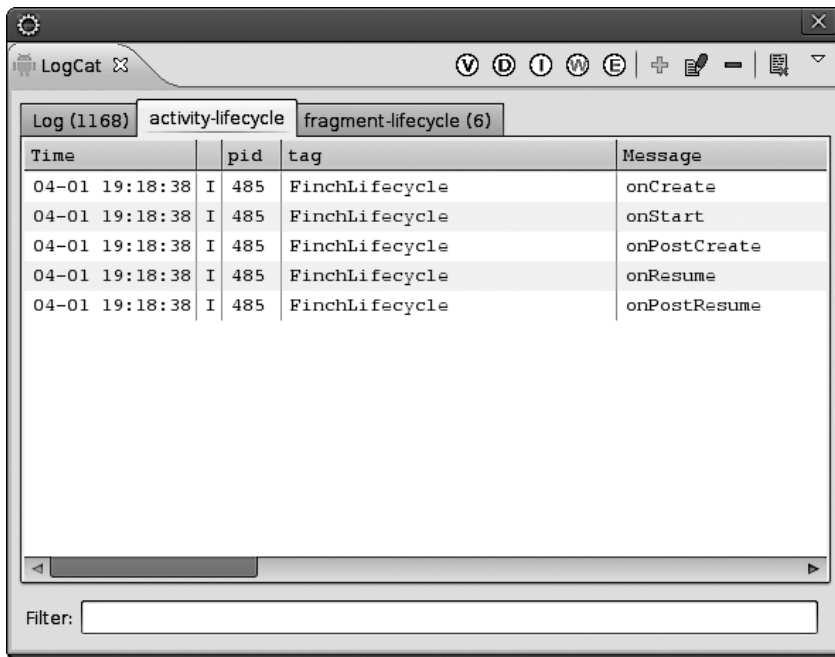


Рис. 10.4. Информация из регистрационного журнала, где показаны новый процесс и восстановленное состояние активности

Чтобы сгенерировать интересную нам регистрационную информацию, можно запустить несколько приложений, пару раз перейти из одного в другое, пользуясь для возврата в приложение Finch переключателем приложений или инструментом Launcher. После того как вы запустите достаточно много приложений и вернетесь обратно в Finch, вы увидите, что идентификатор процесса (PID) изменился, но само приложение, как кажется, осталось в том же состоянии, в каком вы его покинули. Это объясняется тем, что состояние данной активности, как и все остальные компоненты приложения, было восстановлено из сохраненного состояния. Регистрационная информация, приведенная на рис. 10.5, показывает именно такое изменение.



Если вы не обнаружите никакого вывода в виде LogCat, переключитесь в перспективу DDMS (в меню Window (Окно)) и щелкните на том устройстве или эмуляторе, которым вы пользуетесь в виде Devices (Устройства).

Запуская другие приложения, потребляющие память, вы одновременно задействуете некоторые стратегии, которые применяются в Android для освобождения памяти. Разумеется, поскольку программы Android **работают на виртуальной машине**, напоминающей виртуальную машину Java, то первым делом начинается сборка мусора. В ходе ее освобождается память, занятая неиспользуемыми экземплярами объектов, на которые отсутствуют ссылки. Android задействует и еще одну

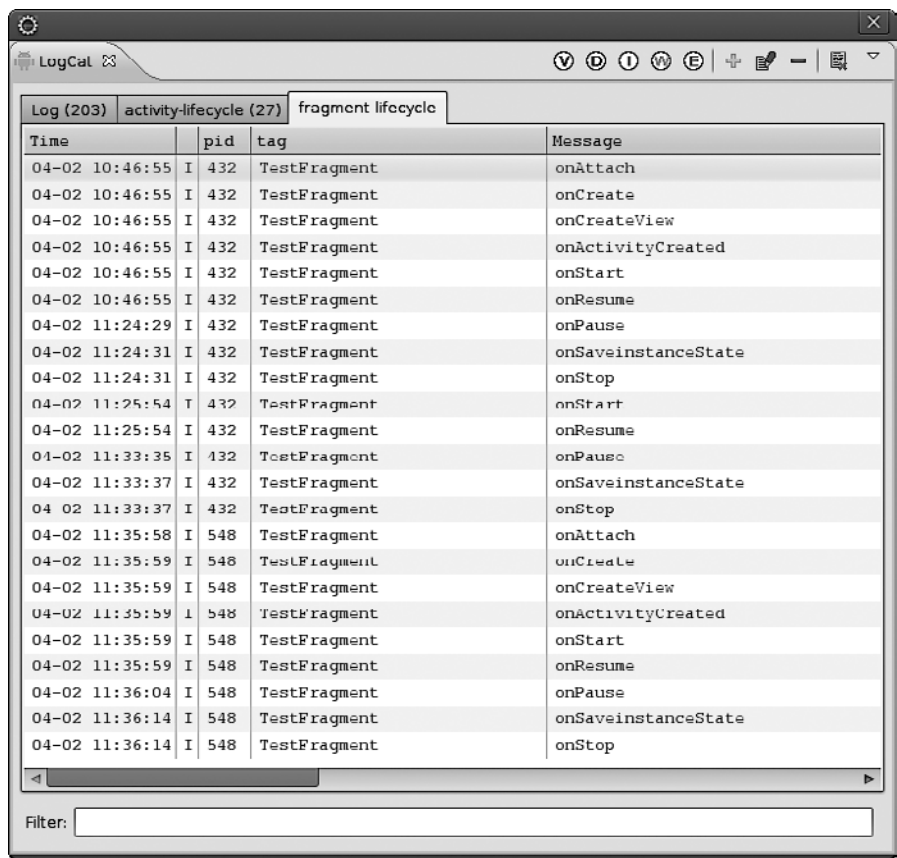


Рис. 10.5. Информация из регистрационного журнала, где показаны новый процесс и восстановленное состояние активности

стратегию сборки мусора: компоненты активности, невидимые для пользователя, могут сохранять свое состояние, а потом уничтожаются. Это всего лишь означает, что система удаляет ссылки на эти компоненты, и после этого такие компоненты могут попасть в сборку мусора. В Android используется еще одна стратегия освобождения памяти: приказывая всем компонентам приложения сохранить состояние, можно удалить целые процессы и восстановить память, которую они занимали. Именно так в Android реализуется сборка мусора, охватывающая сразу по нескольку процессов.

Очистка памяти и жизненные циклы

Жизнь активности в Android кажется скоротечной и полной опасностей. Процесс активности может быть завершен (killed), а объект Activity — уничтожен (destroyed), казалось бы, по прихоти системы. Кроме того, вам никто не гарантирует, что все требуемые переопределения методов жизненного цикла будут вызваны во время завершения процесса.

Для понимания жизненных циклов в Android целесообразно начать с того, что происходит, когда уничтожается экземпляр Activity и когда завершается процесс.

- Уничтожение активности. Активность уничтожается и вызывается метод `onDestroy`, когда Android хочет сбросить *данный экземпляр класса Activity*. «Сбросить» в этом случае означает, что система Android **обнуляет все ссылки, представленные на экземпляр Activity**. А это значит, что если в вашем коде нет ссылки на данную Activity, то Activity в перспективе попадет в сборку мусора. Слово «*уничтожение*» в данном случае не совсем точное — фактически оно означает полное стирание.

После вызова метода `onDestroy` вы можете быть уверены, что данный экземпляр вашего подкласса Activity больше использоваться не будет. Но это еще не означает, что ваше приложение или процесс, в котором оно работает, закончит работу. На самом деле может быть инстанцирован и вызван новый экземпляр того же подкласса Activity. Например, это происходит непосредственно практически сразу же после изменения конфигурации (в частности, после изменения ориентации экрана). В таком случае объект Activity уничтожается для того, чтобы загрузка ресурсов началась заново, уже для новой конфигурации.

- Завершение процесса. Когда в системе Android начинается дефицит памяти, она ищет процессы, которые можно было бы завершить. Как правило, приложения Android работают в отдельных процессах. Поэтому сборка мусора, происходящая в одном процессе, не имеет доступа ко всей памяти системы Android. Это означает, что в условиях дефицита памяти Android находит процессы, у которых в текущий момент отсутствуют какие-либо активные компоненты, и завершает (дословно — «убивает») их. В крайних случаях Android **завершает и такие процессы, которые имеют используемые компоненты**. В простых приложениях процесс становится кандидатом на удаление после того, как к нему будет применен метод `onPause`. То есть нельзя гарантировать, что другие методы жизненного цикла Activity удастся вызвать после вызова `onPause`, поскольку Android может срочно понадобиться свободная память, а чтобы ее добыть, придется завершить процесс.

В обоих описанных случаях приложению, по-видимому, придется сохранить какое-либо состояние, которое временно существует в пользовательском интерфейсе приложения. К этому состоянию относится разнообразный пользовательский ввод, который система пока не успела обработать, состояние определенного визуального индикатора, не входящего в состав модели данных, и т. д. Вот почему все компоненты приложения, и особенно все активности, должны иметь возможность переопределять некоторые методы жизненного цикла.

Методы жизненного цикла класса Activity

Теперь, разобравшись, когда и почему вообще вызываются методы жизненного цикла, рассмотрим отдельные методы из приведенного выше листинга.

- 1 Метод `onCreate` вызывается после создания экземпляра Activity. Именно здесь происходит большая часть инициализации основной массы приложений:

считывание макетов в память и создание экземпляров View, связывание с данными и т. д. Обратите внимание: если данный экземпляр Activity не был разрушен, а процесс не был завершен, то этот метод больше не вызывается. Он вызывается только при создании нового экземпляра класса Activity. Аргументом для данного метода является объект Bundle, содержащий сохраненное состояние приложения. Если сохраненного состояния нет, то этот аргумент имеет значение null.

- ❷ Метод `onRestart` вызывается только в том случае, если активность была остановлена. «Остановлена» — означает, что активность не находится на переднем плане и не взаимодействует с пользователем. Этот метод вызывается до метода `onStart`.
- ❸ Метод `onStart` вызывается, когда объект Activity и его виды становятся видны пользователю.
- ❹ Метод `onResume` вызывается, когда пользователь получает возможность взаимодействовать с объектом Activity и его видами.
- ❺ Метод `onPause` вызывается перед тем, как должен стать видимым другой экземпляр Activity, а актуальная Activity перестает взаимодействовать с пользователем.
- ❻ Метод `onStop` вызывается, когда активность более невидима пользователю и он не может с ней взаимодействовать.
- ❼ Метод `onDestroy` вызывается перед тем, как экземпляр Activity должен быть уничтожен, то есть снят с использования. Перед вызовом этого метода активность уже перестала взаимодействовать с пользователем и больше не отображается на экране. Если этот метод вызывается в результате вызова `finish`, то вызов `isFinishing` возвратит `true`.

Сохранение и восстановление состояния экземпляра

Очистка памяти и жизненный цикл компонента — без этих этапов в работе никак не обойтись. Вот почему подклассы вашей Activity должны сохранять состояние. Ниже описано, как и когда это нужно делать.

Класс Bundle существует для того, чтобы хранить в нем сериализованные данные в форме пар «ключ — значение». Данные могут относиться к примитивным типам или любому типу, реализующему интерфейс `Parcelable` (см. подраздел «Объект `Parcelable` для передачи данных» раздела «Сериализация» главы 3). О классе Bundle подробнее рассказано на сайте разработчиков Android по адресу <http://developer.android.com/reference/android/os/Bundle.html>. При сохранении состояния активности используются методы `put` класса Bundle.

При вызове методов `onCreate` и `onRestoreInstanceState` объект Bundle передается этому методу. Объект Bundle содержит данные, которые в него поместил предыдущий экземпляр того же класса Activity, чтобы информация о конкретной активности сохранялась между ее инстанцированиями. То есть если экземпляр Activity имеет состояние, не считая того, которое долговременно хранится в модели данных, это состояние можно сохранять, а затем восстанавливать в многочисленных экземплярах класса Activity. С точки зрения пользователя, он возвращается к работе

с той самой активностью, от работы с которой пришлось оторваться, но фактически пользователь может видеть совершенно новый экземпляр класса `Activity`, возможно исполняемый в совершенно новом процессе.

Вероятно, вы заметили, что в жизненном цикле метода `onPause` не предоставляется объект `Bundle` для сохранения состояния. Итак, когда сохраняется состояние? Существуют отдельные методы класса `Activity`, предназначенные для сохранения состояния, а также методы для уведомления о том, что состояние восстанавливается.

8 Здесь приложение получает возможность сохранить состояние экземпляра. Состояние экземпляра — это такое состояние, которое не сохраняется в долговременной памяти вместе с моделью данных приложения. Примером состояния экземпляра может быть, например, состояние индикатора или другого элемента, полностью входящего в состав объекта `Activity`. Этот метод имеет реализацию и в родительском классе: он вызывает метод `onSaveInstanceState` каждого объекта `View` в данном экземпляре `Activity`. В результате сохраняется состояние этих объектов `View`, и часто это единственное состояние, которое приходится сохранять таким образом. Данные, которые должен сохранить ваш подкласс, сохраняются при помощи методов `put` класса `Bundle`.

10 Метод `onRestoreInstanceState` вызывается, когда имеется состояние экземпляра, которое следует восстановить. Если этот метод вызывается, то такой вызов происходит после `onStart` и до `onPostCreate`, редко используемого метода жизненного цикла, описанного в пункте «Второстепенные методы жизненного цикла класса `Activity`» далее.

Изменения конфигурации и жизненный цикл активности

Выше мы рассказали, как можно спровоцировать систему Android завершить процесс, в котором работает активность или любой другой компонент приложения. Для этого нужно просто запустить достаточно много приложений, чтобы системе пришлось завершить некоторые процессы. Если после этого просмотреть регистрационный журнал и рис. 10.5, то можно увидеть, что ID процесса изменяется и что создается новый экземпляр подкласса `Activity`, определяющий, как программа будет взаимодействовать с пользователем. Этот новый экземпляр перезагружает все ресурсы для данной активности, а если в программе имеются какие-либо данные приложения, которые требуется перезагрузить, то они также будут загружены заново. В итоге получается, что пользователь продолжает работать с якобы «той же самой» активностью, как будто ничего и не произошло. Новый экземпляр выглядит точно как старый, поскольку имеет ровно то же состояние, что и старый.

Существует и другой способ принудить Android использовать новый экземпляр `Activity`: изменить конфигурацию системы. Самое распространенное изменение конфигурации в приложении — это изменение ориентации экрана. Но феномены, причисляемые к изменениям конфигурации, гораздо разнообразнее: например, возможность подключения физической клавиатуры, изменение локали, изменение размеров шрифта и др. Наиболее общим фактором при всех изменениях конфигурации является то, что такие изменения *могут* требовать перезагрузки ресурсов, обычно потому, что требуется перерасчет компоновки элементов.

Простейший способ убедиться в том, что все ресурсы, используемые в активности, перезагружены с учетом новой конфигурации, — обязательно сбрасывать старый экземпляр активности и создавать новый, чтобы при этом и перезагружались все ресурсы. Чтобы это произошло, когда приложение работает в эмуляторе, нажмите клавишу 9 на числовой клавиатуре. После этого в эмуляторе изменится ориентация экрана. В регистрационном журнале вы увидите примерно такую же информацию, как на рис. 10.6. В журнале будет показано, что вызван метод `onDestroy`, поскольку экземпляр `Activity` сбрасывается в рамках процесса изменения конфигурации, а не из-за того, что в системе не хватает памяти и она пытается завершить процесс, чтобы высвободить память. Вы также заметите, что во всех новых экземплярах объекта `Activity` ID (идентификатор) процесса остается одним и тем же — системе не приходится восстанавливать ту память, которую использует приложение.

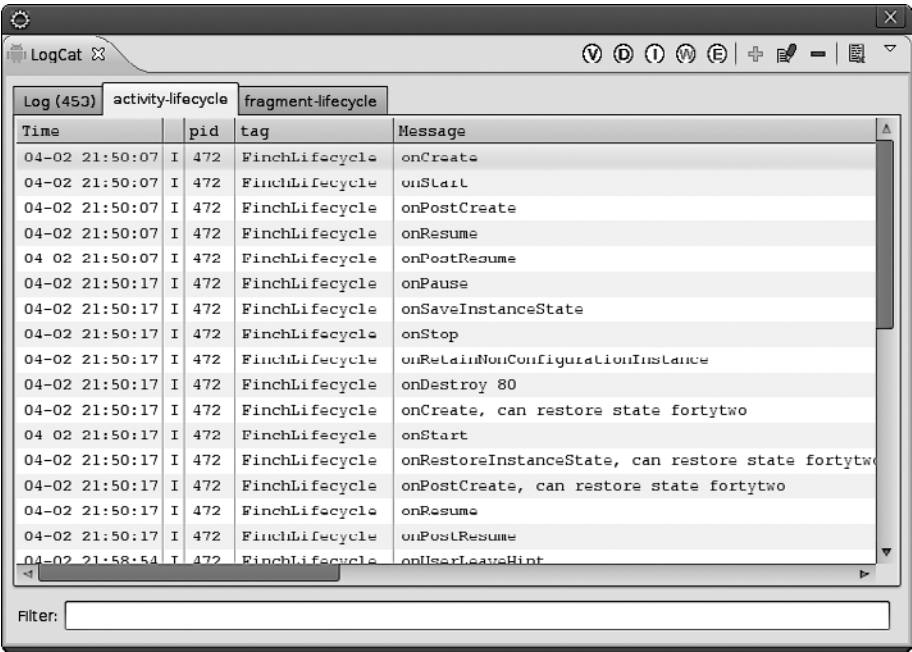


Рис. 10.6. При вызове метода `onDestroy` идентификационный номер процесса не изменяется

Такой подход может показаться расточительным: новый экземпляр `Activity`? Зачем? Почему бы не пользоваться тем, который уже есть? Разве создание нового экземпляра не замедлит работу? Тем не менее в большинстве случаев ресурсы, загружаемые активностью, когда она запускается, составляют основную часть состояния экземпляра `Activity`. Во многих случаях основной объем вычислений, протекающих в активности, происходит тогда, когда она считывает XML-файл и рассчитывает компоновку. И, как правило, изменения конфигурации — например, ориентации экрана или локали — требуют пересчета практически для всех макетов, загруженных из ресурсов. Итак, изменение конфигурации практически неизбежно

приводит к перезапуску активности, а также к затратам процессорного времени, которые требуются для такого перезапуска.

Не забывайте, что, когда Android «уничтожает» активность, происходит, в сущности, всего одна вещь: сбрасывается ссылка на данную активность, и эта активность в итоге попадает в сборку мусора. Всякий раз, когда пользователь переходит от одной активности к другой, выполняются все расчеты, необходимые для создания новой активности. Если делать то же самое, когда происходит изменение конфигурации, то это будет не слишком сложной задачей для системы.

Второстепенные методы жизненного цикла класса Activity

В документации Android для описания жизненного цикла активности используются еще несколько методов (кроме основных методов жизненного цикла), которые также вызываются на разных этапах «жизненного пути» активности.

- 11 Метод `onPostCreate` вызывается после вызова метода `onRestoreInstanceState`. Он может быть полезен, если ваше приложение требует сохранять состояние в два этапа. Методу передается объект `Bundle`, содержащий состояние экземпляра.
- 12 Метод `onPostResume` вызывается после `onResume`, когда экземпляр `Activity` должен быть видим и должен взаимодействовать с пользователем.
- 13 Метод `onUserLeaveHint` вызывается перед тем, как активность должна перестать быть видимой и перестать взаимодействовать с пользователем из-за действий самого пользователя. Например, пользователь может нажать клавишу **Back** (Назад) или **Home** (Домой). Именно здесь удобно удалять уведомления и диалоговые окна.

В листинге программы, показанном на рис. 10.6, вы видите, что мы реализовали переопределения этих методов, чтобы зарегистрировать, когда они будут вызваны. Эти методы существуют для таких случаев, когда, например, вам требуется дополнительный этап для восстановления состояния экземпляра.

Однако если вам по-настоящему нужно сохранить определенные данные на период, в течение которого несколько раз изменится конфигурация, и эти данные не входят в состояние, сохраняемое в модели данных, которая будет сохранена, и не входят в `Bundle`, то можно воспользоваться методом `onRetainNonConfigurationInstance`, чтобы «припрятать» ссылку на объект. Затем ссылка может быть запрошена новым экземпляром `Activity` при помощи метода `getLastNonConfigurationInstance`:

- 9 Метод `onRetainNonConfigurationInstance` вызывается после `onStop`. Это означает, что его вызов не гарантируется. А если он и вызван, то не гарантируется, что возвращенная ссылка будет сохранена и предоставлена последующему экземпляру `Activity`. Метод `getLastNonConfigurationInstance()` может быть вызван в методе `onCreate` или позже, при восстановлении состояния активности.

Чтобы проиллюстрировать, как используются эти методы, вернемся к объекту, содержащему идентификатор задачи для активности, когда вызывается метод `onRetainNonConfigurationInstance`. А когда вызовется метод `onRetainNonConfigurationInstance(Bundle)`, мы убедимся, что идентификатор задачи не изменился. Таким образом, подтверждается, что, хотя экземпляр компонента или даже весь процесс является новым с точки зрения пользователя, задача осталась той же самой.

Рассказывая о практическом применении этих методов, чаще всего говорят о ситуации, когда требуется сохранять результаты в веб-запросе: нужно повторить запрос, но время задержки до попадания запроса на веб-сервер может достигать нескольких секунд. Итак, данные могут быть воссозданы, если система не может сохранить их до тех пор, как эту информацию получит новый экземпляр объекта `Activity`. Но гораздо разумнее было бы просто кэшировать данные. Правда, в главе 13 мы продемонстрируем, как использовать локальную базу данных в качестве посредника для кэширования в приложениях с передачей состояния представления (RESTful). В таком случае необходимость в описанной выше оптимизации снижается.

Визуализация жизненного цикла фрагмента

Если вы занимаетесь разработкой для Android 3.0 Honeycomb, API уровня 11 или выше, то в вашем распоряжении будет API для работы с фрагментами. Если же вы предпочитаете писать программы для более ранних версий, чем Honeycomb, но собираетесь использовать в создаваемом пользовательском интерфейсе объекты `Fragment`, то можете применить пакет совместимости Android, который мы рассматривали в главе 7. Код примеров в этой главе представлен в двух формах: в первом случае код ориентирован на работу с API уровня 11, а во втором случае — пригоден для работы с более ранними версиями, вплоть до API уровня 4, который соответствует версии Android 1.6. Следующие примеры кода `Fragment` идентичны во всем, кроме объявления пакета для класса `Fragment`. В том, что касается жизненного цикла `Fragment`, код работает идентично.

Этот код, как и в классе `Activity`, показанном выше, организует обратные вызовы жизненного цикла таким образом, что их можно отслеживать по мере выполнения программы:

```
package com.oreilly.demo.pa.ch10.finchlifecycle;

import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class TestFragment extends Fragment {

    // получение метки для записей журнала
    private final String TAG = this.getClass().getSimpleName();

    public TestFragment() {

    }

    @Override
```

```
public void onAttach(Activity activity) { ❶
    super.onAttach(activity);
    Log.i(TAG, "onAttach");
}

@Override
public void onCreate(Bundle savedInstanceState) { ❷
    super.onCreate(savedInstanceState);
    if (null != savedInstanceState) {
        // Здесь восстанавливается состояние
    }
    Log.i(TAG, "onCreate");
}

@Override
public View onCreateView(LayoutInflater inflater,
                        ViewGroup container, ❸
                        Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.fragment_content, container,
                            false);
    Log.i(TAG, "onCreateView");
    return v;
}

@Override
public void onActivityCreated(Bundle savedInstanceState) { ❹
    super.onActivityCreated(savedInstanceState);
    Log.i(TAG, "onActivityCreated");
}

@Override
public void onStart() { ❺
    super.onStart();
    Log.i(TAG, "onStart");
}

@Override
public void onResume() { ❻
    super.onResume();
    Log.i(TAG, "onResume");
}

@Override
public void onPause() { ❼
    super.onPause();
    Log.i(TAG, "onPause");
}

@Override
public void onStop() { ❽
    super.onStop();
}
```

```

        Log.i(TAG, "onStop");
    }

    //////////////////////////////////////////////////
    // Вызывается во время жизненного цикла, когда состояние экземпляра
    // следует сохранить/восстановить
    //////////////////////////////////////////////////

    @Override
    public void onSaveInstanceState(Bundle toSave) {
        super.onSaveInstanceState(toSave);
        Log.i(TAG, "onSaveInstanceState");
    }
}

```

Как и при работе с фильтром LogCat, который мы использовали для нахождения записей журнала, демонстрирующих обратные вызовы к компонентам Activity, теперь мы настроим фильгр для обратных вызовов, идущих к Fragment.

Если повторить шаги, сделанные выше, — запускать другие приложения, пока не увидите в окне LogCat, что начался вызов методов жизненного цикла Fragment, — то окажется, что каждый экземпляр Fragment, находящийся в экземпляре Activity, действует на располагающиеся в нем виды (Views) так же, как и содержащая его активность. Вызываются похожие преобразования жизненного цикла и состояний.

Рассмотрим каждый вызываемый метод подробнее, учитывая, что теперь мы знаем, когда каждый из них вызывается.

- ❶ Метод `onAttach` вызывается, когда экземпляр Fragment ассоциируется с экземпляром Activity. Это еще не означает полной инициализации Activity.
- ❷ Метод `onCreate` вызывается, когда создается или воссоздается экземпляр Fragment. Если происходит восстановление после предшествовавшего разрушения фрагмента или активности, в которой он содержался, то аргумент `bundle` окажется ненулевым, если было сохранено какое-либо состояние.
- ❸ Метод `onCreateView` вызывается, когда экземпляр Fragment должен создать иерархию объектов View, которые в нем содержатся. Роль фрагмента в активности довольно необычна. В чем-то фрагмент напоминает по поведению ViewGroup, но он не относится к иерархии класса View. Можно сказать, что фрагмент позволяет экземпляру активности содержать несколько наборов видов. В нашем примере загружается исключительно простой макет, содержащий единственный вид TextView.
- ❹ Метод `onActivityCreated` вызывается, когда активность, содержащая экземпляр фрагмента, уже создана, а также созданы объекты View, содержащиеся во Fragment. На этом этапе вы можете свободно искать объекты View, например, по их ID.
- ❺ Метод `onStart` вызывается, когда фрагмент становится видимым и работает. Это происходит примерно так же, как и с методом `onStart`, относящимся к активности.
- ❻ Метод `onResume` вызывается, когда фрагмент становится видимым и работает.
- ❼ Метод `onPause` вызывается в тех же условиях, что и метод `onPause`, относящийся к активности, когда система собирается увести фрагмент с переднего плана.

- 8 Метод `onStop` вызывается, перед тем как фрагмент должен остановить работу.
- 9 Метод `onSaveInstanceState` вызывается, когда необходимо сохранить состояние экземпляра, чтобы, если этот экземпляр будет уничтожен (фактически будут удалены указывающие на него ссылки), любое предназначенное для сохранения состояние, специфичное для класса, можно было сохранить в объекте `Bundle`, переданном этому вызову.

Объекты-фрагменты не являются компонентами. Фрагмент можно считать способом разбиения `Activity` на несколько объектов, содержащихся в `Activity`. Каждый фрагмент имеет собственную иерархию видов `View`, и эта иерархия действует так, как если бы она располагалась внутри `Activity`.

Класс `Activity` и работоспособное приложение

Понимание жизненных циклов приложения — необходимая предпосылка для того, чтобы реализовать работоспособное приложение, а также для того, чтобы осознавать, почему те или иные программы могут работать неправильно. Замедленную работу, перерасход ресурсов и неожиданные явления в пользовательском интерфейсе зачастую можно диагностировать после наблюдения за жизненным циклом приложения. Жизненный цикл сложно понять, просто просмотрев код или страницу с документацией по классу `Activity`. Чтобы вы могли рассмотреть жизненный цикл на практике, мы внедрим в нашей реализации методов жизненного цикла Android регистрирующие вызовы, запустим несколько программ и рассмотрим, как протекает жизненный цикл в работающей программе. При использовании данного каркаса можно оставить в коде приложения регистрирующие вызовы, продолжая разработку. Регистрация в этих методах приложения часто оказывается нелишней и помогает диагностировать проблемы.

Большинство методов, вызываемых при изменениях в жизненном цикле, реализуются на покомпонентной основе, а некоторые — на попроцессной. Компоненты всех типов — `Service`, `BroadcastReceiver`, `ContentProvider` и `Activity` — имеют собственный жизненный цикл. Жизненные циклы всех компонентов, кроме `Activity`, были рассмотрены в главе 3. Большинство жизненных циклов проще, чем цикл `Activity`. Это объясняется тем, что класс `Activity` взаимодействует с пользователем. Когда `Activity` уже не является видимой частью пользовательского интерфейса, память, занятая ресурсами, связанными с этой активностью, вполне может быть очищена при необходимости. Управление памятью, которая занята ресурсами, относящимися к компонентам, — одна из основных целей жизненного цикла компонентов.

Жизненный цикл активности и работа пользователя с системой

Если приложение хорошо приспособлено для мобильной среды вообще, то в цикле управления его жизненным циклом потребуется меньше кода:

- если информация, используемая активностью, всегда является актуальной и находится в базе данных, то вам не придется специально сохранять эту информацию в коде в методе жизненного цикла приложения;

- если пользовательский интерфейс вашего приложения имеет минимум информационного состояния, то вам не придется сохранять много информации в методе жизненного цикла активности — если вообще придется.

Эти ограничения кажутся достаточно строгими, но в мобильных и других устройствах они вполне уместны. Батарея мобильного телефона может разрядиться в любой момент, и чем меньшая часть состояния и модели данных приложения содержится в памяти, тем меньше потеряет пользователь, если устройство неожиданно отключится. Пользователь мобильного устройства может отвлечься на телефонный звонок, и у него уже не будет времени вернуться к приложению, чтобы сохранить в нем данные. Работа с мобильными приложениями отличается от работы с типичными интерактивными программами для ПК, где документы в файловых системах становятся моделями данных, содержащимися в оперативной памяти, и эти модели нужно специально сохранять — иначе данные будут потеряны.

В этой и в следующих главах вы увидите, что жизненный цикл приложения, модель данных и другие аспекты архитектуры приложения, а также взаимодействие системы и пользователя тесно переплетены. И если избрать путь наименьшего сопротивления как минимум в том, что касается реализации методов жизненного цикла, то у нас будут получаться отказоустойчивые, простые в использовании и работоспособные приложения, полноправные представители мира Android. Если рассматривать разрядку батареи так же, как и ситуацию, когда пользователь больше не работает с активностью или когда система завершает активность, чтобы высвободить память и другие системные ресурсы, то вы упростите свою реализацию и облегчите жизнь пользователю. При работе с мобильными устройствами следует избегать использования таких явных действий, как «сохранение» и «выход из системы», — это еще одно условие создания работоспособных приложений.

Методы жизненного цикла класса Application

Методы жизненного цикла класса Application довольно редко используются в простых приложениях, что неудивительно. Ими не следует злоупотреблять даже в сложном приложении. Очень легко набить переопределения класса Application такими данными, которые будут висеть в памяти, влияя на работу нескольких активностей. Таким образом, мы на корню губим существующие в Android возможности управления ресурсами на покомпонентной основе. Например, если вы перенесете ссылку на определенные данные от объекта Activity к объекту Application, то та борьба за ресурсы, которая разворачивается в системе в условиях дефицита памяти, просто перенесется в жизненный цикл приложения и управлять этими данными придется отдельно от жизненного цикла активности.

Здесь мы реализуем методы жизненного цикла класса Application и покажем, какое место они занимают в жизненном цикле приложения Android. Кроме того, вам может быть полезна информация, выводимая при журналировании этих методов:

```
package com.finchframework.finch;
```

```
import android.app.Application;  
import android.content.res.Configuration;
```

```
import android.util.Log;

/**
 * @author zigurd
 *
 *      Это каркас подкласса Application. Здесь иллюстрируется, что вам
 *      может понадобиться сделать в подклассе Application.
 *
 *      Чтобы инстанцировать этот класс, нужно сослаться на него
 *      в теге application в файле описания.
 */
public class FinchApplication extends Application {
    private final String TAG = this.getClass().getSimpleName();

    @Override
    public void onCreate() {
        // сначала вызываем родительский класс
        super.onCreate();

        // Именно здесь можно разместить код, который будет управлять
        // глобальными данными, важными для работы нескольких активностей.
        // Но лучше держать как можно больше информации в базе данных,
        // а не в памяти.
        Log.i(TAG, „onCreate");
    }

    @Override
    public void onTerminate() {
        Log.i(TAG, "onTerminate");
    }

    @Override
    public void onLowMemory() {
        // от кэширования в оперативной памяти здесь придется отказаться
        Log.i(TAG, "onLowMemory");
    }

    @Override
    public void onConfigurationChanged(Configuration newConfig) {
        Log.i(TAG, "onConfigurationChanged");
        if (Log.isLoggable(TAG, Log.VERBOSE)) {
            Log.v(TAG, newConfig.toString());
        }
    }
}
```

Ранее мы упоминали о том, что во многих приложениях не требуется делать подклассы `Application`. Поэтому мастер новых проектов Android не создает подкласс `Application`, а также не ставит на него ссылку в файле описания. Подобно исходно-

му объекту, который запускается одновременно с интерактивным приложением, подкласс `Application`, который вы создаете, инстанцируется системой Android в рамках запуска приложения. Все происходит так же, как и с инстанцированием `Activity`. Система использует свойство `android:name` тега `application` и создает правильный экземпляр класса `Application`. Простейший способ сделать все это правильно — открыть вкладку **Application** (Приложение) в редакторе описаний (манифестов). Первое поле на этой вкладке называется **Name** (Имя) (рис. 10.7). Если нажать кнопку **Browse** (Обзор) рядом с этим полем, то можно отобразить подклассы `Application`, присутствующие в вашем приложении.

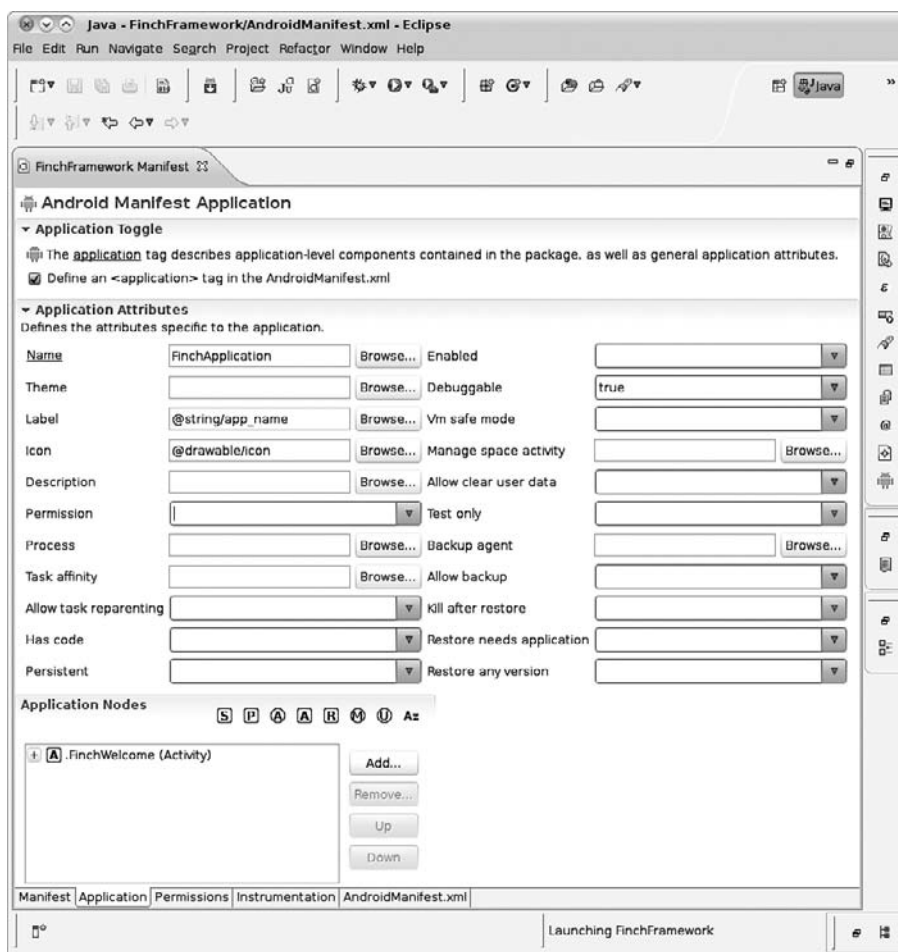


Рис. 10.7. Поле Name (Имя) на вкладке Application (Приложение) редактора описаний, где можно ввести имя определенного вами подкласса от `Application`

Как и в случае с методами жизненного цикла класса `Activity`, важнее всего знать, когда вызываются методы жизненного цикла `Application`. Разумеется, это можно узнать при отладке приложения и путем установки контрольных точек в каждом

методе. Но чаще наиболее интересную информацию удастся добыть, наблюдая за поведением приложений, работающих подолгу, и фильтруя регистрационный журнал по тегам, используемым в подклассах `Activity` и `Application`. Тогда сразу становится понятно, когда вызываются методы жизненного цикла.

Два обратных вызова, которые особенно интересно отследить в классе `Application`, — это `onLowMemory` и `onTerminate`. Они с достаточной точностью сообщают вам, когда с точки зрения системы начинается дефицит памяти и когда завершается ваше приложение. Вторая ситуация обычно неочевидна, поскольку в большинстве приложений Android не требуется явного выхода из программы. Это объясняется особенностями управления памятью в Android, во взаимодействии с жизненными циклами компонентов. Такой механизм легко удаляет из памяти неиспользуемый код, если он был правильно реализован — с учетом проблем, связанных с жизненными циклами и с управлением памятью.

11 Создание пользовательского интерфейса

В этой главе вы научитесь применять API и различные инструменты в процессе разработки пользовательского интерфейса. Классы пользовательского интерфейса Android и инструменты, входящие в SDK, были специально разработаны, чтобы вы могли с легкостью создавать удобные в использовании приложения Android.

Пример, рассмотренный в этой главе, в готовом виде на экране планшета будет выглядеть так, как показано на рис. 11.1.

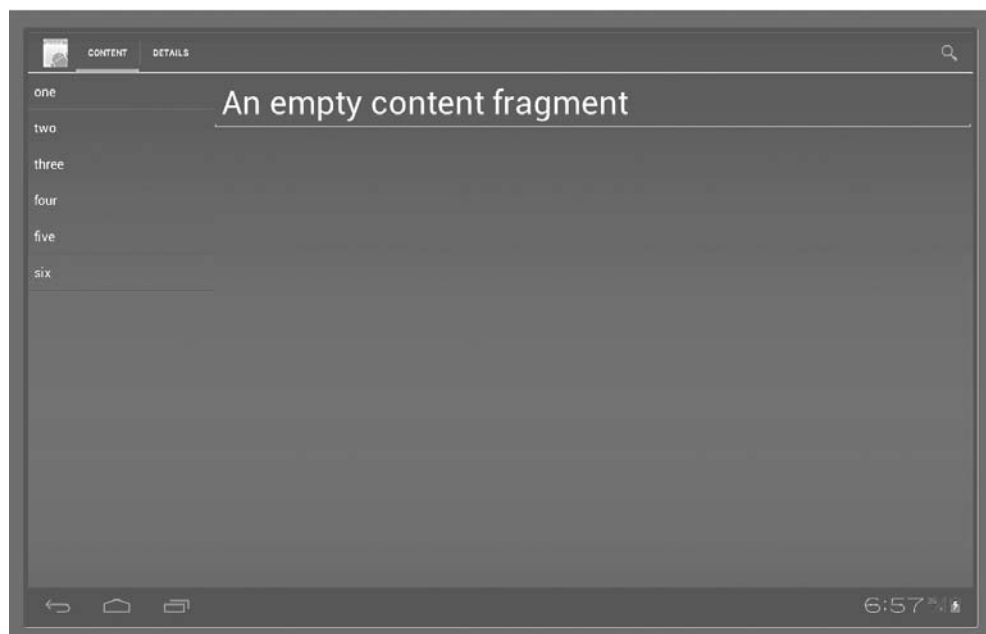


Рис. 11.1. Вид программы на экране планшета

Аналогичный скриншот, сделанный с телефона, показан на рис. 11.2.

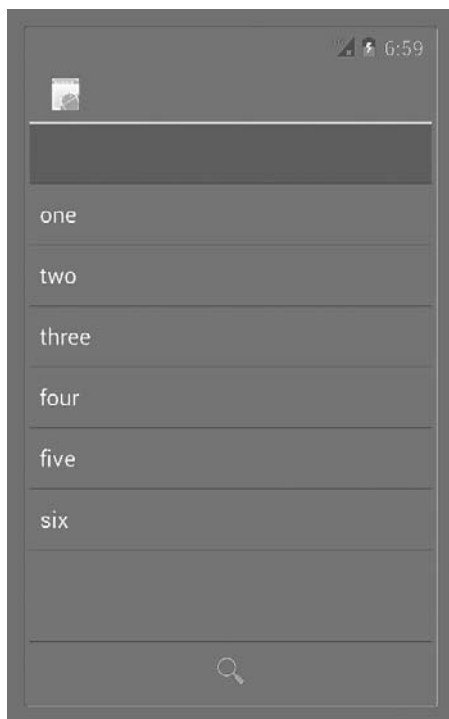


Рис. 11.2. Вид программы на экране телефона

В обоих случаях выполняется один и тот же код. Это означает, что все решения о внешнем виде пользовательского интерфейса, зависящие от параметров экрана, принимаются во время исполнения. И практически все эти решения принимает сама система Android.

Инструменты, предлагаемые в Android для создания пользовательского интерфейса, можно комбинировать практически как угодно. Пример API Demos, а также другие примеры программ в SDK содержат различные образцы использования классов пользовательского интерфейса в Android, в частности Activity и Fragment. Но вы можете сами скомбинировать такие приемы в приложении.

В коде примеров для этой главы реализуются самые современные пользовательские интерфейсы в соответствии с рекомендациями по созданию таких интерфейсов. Иллюстрируются наиболее важные классы пользовательских интерфейсов, предоставляются регистрируемая информация (логи) и инструментарий, помогающие понять, что стоит за событиями, происходящими на экране.

Общий дизайн интерфейса

При разработке общего дизайна пользовательского интерфейса создаются все элементы, которые будут видны и доступны для применения в пользователь-

ском интерфейсе. Именно на данном этапе вы определяете, что хотите видеть в итоге.

Мы рассмотрим типичный дизайн пользовательского интерфейса Android, расскажем, как разработать его на основе единой базы кода, которая отвечает следующим требованиям.

- Приложение работает как на планшетах, так и на телефонах, делегируя классам фрагментов максимально возможный объем кода пользовательского интерфейса. При этом классы активностей должны реагировать на то, как система Android «решает» использовать ресурсы макета окна в зависимости от размера экрана и пиксельной плотности.
- Код работает как в книжной, так и в альбомной ориентации. Для обеспечения четкости представления в книжном и альбомном вариантах ориентации используются различные варианты компоновки.
- Здесь показано, как можно добавлять и убирать фрагменты из объекта Activity, а также как взаимодействуют переходы и транзакции, осуществляемые с фрагментами.
- Код показывает, как можно совмещать и разделять панель действий и находящиеся на ней элементы, как пользоваться вкладками и следовать фрагменто-ориентированной стратегии для расположения пользовательского интерфейса как на экране планшета, так и на экране телефона.

Фрагмент, активность и масштабируемый дизайн

Для создания красивого и функционального пользовательского интерфейса воспользуемся фрагментами. Подробное обсуждение класса `Fragment`, менеджера фрагментов (`FragmentManager`) и жизненного цикла фрагментов приводится в разделе «Жизненный цикл фрагмента» главы 7.

Экземпляр `Fragment`, как и экземпляр `Activity`, позволяет добавлять вложенные объекты `View` в иерархию видов. Но для компоновки фрагмента можно либо изменить спецификацию макета прямо в XML-файле, либо воспользоваться визуальным редактором компоновки внутри активности, как если бы она была видом.

На рис. 11.1 и 11.2 показано, как приложение из этой главы выглядит на большом и маленьком экранах. Мы рассмотрим, как получить две такие картинки. При планировании внешнего вида макетов мы воспользуемся «каркасными моделями» и решим, как поделить пользовательский интерфейс на отдельные активности на небольшом экране.

Начнем с дизайна для большого планшетного экрана. При этом рассмотрим, как можно представить различные виды информации рядом друг с другом, *и только потом* обсудим, как разделить пользовательский интерфейс на активности для работы на небольшом экране, а также создать навигационную иерархию.

Достичь нужной нам модульности помогут фрагменты. Благодаря тому что фрагмент содержит код, который обрабатывает взаимодействие с расположенными в этом фрагменте объектами View, именно фрагмент значительно упрощает многократное использование кода и макетов. Причем упрощение достигается, несмотря на то что на большом и малом экранах отличается как организация пользовательского интерфейса, так и поток данных.

На рис. 11.3 показан пользовательский интерфейс. А каркасная модель на рис. 11.4 демонстрирует, как мы собираемся разделить данный пользовательский интерфейс между двумя активностями, если потребуется работать на небольшом устройстве (телефоне).

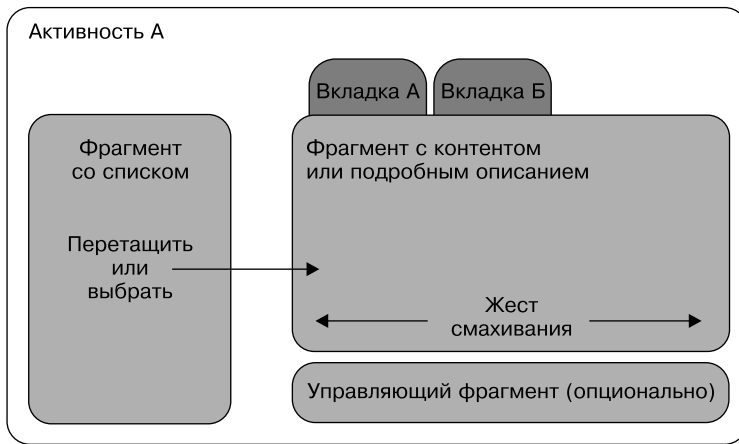


Рис. 11.3. Каркасная модель, демонстрирующая расположение фрагментов на устройстве с большим экраном

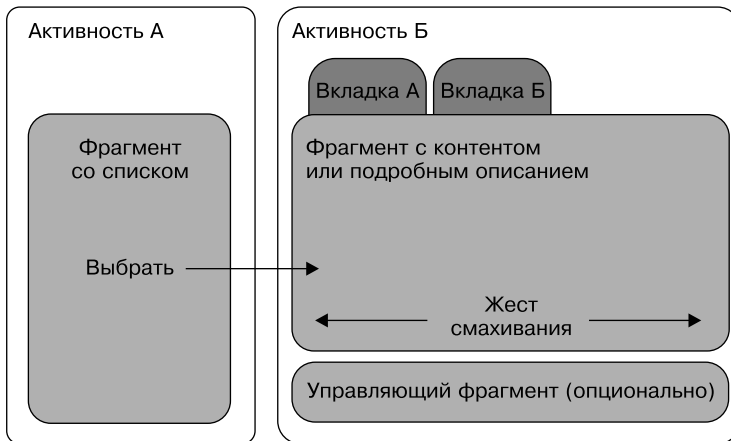


Рис. 11.4. Каркасная модель, демонстрирующая расположение фрагментов на устройстве с небольшим экраном

Визуальное редактирование пользовательских интерфейсов

Когда система Android только появилась, в ней существовали лишь примитивные инструменты для визуального редактирования пользовательских интерфейсов. В большинстве книг по программированию для Android читателю не рекомендуется пользоваться ранними редакторами пользовательских интерфейсов и на примерах объясняется, как размечать пользовательские интерфейсы в XML.

Одно из наиболее значительных изменений в разработке для Android в последних версиях SDK заключается в том, что эти примитивные инструменты наконец-то заменены полнофункциональным визуальным редактором пользовательских интерфейсов. Этот редактор характеризуется достаточной мощностью и выразительностью, поэтому разработчикам рекомендуется в первую очередь обращаться к нему, как только возникает необходимость создать пользовательский интерфейс. Работая таким способом, вы по определению не допустите синтаксических ошибок в XML — а такие ошибки бывает очень сложно найти и исправить.

Новый редактор позволяет даже выполнять рефакторинг спецификаций пользовательских интерфейсов, работая с визуальными инструментами. Он также может находить дублирующиеся спецификации пользовательских интерфейсов и извлекать их в общий код.



Поскольку Android SDK является средой для перекрестной разработки, создать визуальный редактор графических пользовательских интерфейсов оказалось сложнее, чем выполнить подобную работу для полностью «автономной» среды разработки. Например, когда компания Sun создала редактор пользовательских интерфейсов Matisse для работы в Swing, его авторы, конечно, воспользовались тем фактом, что сама интегрированная среда разработки (IDE) NetBeans написана с применением Swing. В IDE доступны одинаковые классы, отвечающие как за отображение пользовательского интерфейса, так и за реагирование на ввод данных в приложение. Поэтому как только потребуется показать разработчику, как будет выглядеть создаваемый пользовательский интерфейс, для его отрисовки вызываются классы Swing.

С Android SDK возникает более сложная ситуация, и она трудна сразу в нескольких отношениях. Eclipse работает при помощи виртуальной машины Java, а не виртуальной машины Dalvik, а пользовательский интерфейс Eclipse построен с использованием классов SWT. Поэтому редактору графических интерфейсов для Android приходится в значительной степени интегрировать поведения классов пользовательского интерфейса Android в плагин Eclipse, чтобы графические интерфейсы Android могли отображаться в такой среде. И они отображаются, несмотря на то что все лежащие в основе процесса графические классы и классы пользовательского интерфейса отличаются от тех классов, которые фактически применяются в среде времени исполнения Android.

Начнем с чистого листа

В этом разделе спроектируем пользовательский интерфейс в нисходящем порядке. Начнем с класса Activity, содержащего основные элементы пользовательского

интерфейса, в той форме, как эта активность будет отображаться на планшете. Далее в экземплярах этого класса `Activity` будут создаваться объекты `Fragment`.

Это важный этап, который необходимо завершить прежде, чем мы начнем использовать визуальные инструменты для редактирования файлов макета. Дело в том, что мы должны превратить наши каркасные модели в XML-код, размечающий пользовательский интерфейс. Поскольку от класса `Fragment` всегда создаются подклассы, мы будем создавать их до того, как приступим к компоновке наших фрагментов.

Что касается каркасной модели, чтобы отслеживать ход интересующих нас процессов, создадим три подкласса от `Fragment`:

- `ContentFragment`;
- `DetailFragment`;
- `QueryResultsListFragment`.

Многие обратные вызовы жизненного цикла в подклассах нашего `Fragment` реализованы для обеспечения регистрации, простой визуализации жизненного цикла объекта `Fragment` и для того, чтобы можно было выстраивать специфичный для приложения код вокруг методов, образующих скелет нашего кода. Ниже приведен исходный код класса `QueryResultsListFragment`.

В этом коде цифрами отмечены следующие моменты.

- ❶ Методы жизненного цикла с кодом для регистрации.
- ❷ Реализация `onCreateView`, наполняющего информацией наш вид и вызывающего `attachAdapter`.
- ❸ Закрытый (приватный) метод `attachAdapter`, активирующий в этом фрагменте список для отображения некоторых тестовых данных.
- ❹ Реализация интерфейса `OnItemClickListener`, отвечающего за взаимодействие с пользователем. В данном случае этот интерфейс выполняет простой код, отсылающий некоторые тестовые данные объектам `Fragment` в правой части экрана либо открываемой впоследствии активности, если экран слишком мал для размещения двух фрагментов. Обратите внимание: здесь нет кода для принятия такого решения.

```
package com.finchframework.uiframework;
```

```
import android.app.Activity;
import android.app.Fragment;
import android.content.res.Configuration;
import android.os.Bundle;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
```

```
import android.widget.AdapterView;
import android.widget.ListView;

public class QueryResultsListFragment extends Fragment implements
    OnItemClickListener{

    // строка для регистрации имени класса
    private final String TAG = getClass().getSimpleName();

    // включение или отключение регистрации
    private final boolean L = true;

    public void onAttach(Activity activity) {

        super.onAttach(activity);
        // уведомление о том, что код ассоциирован с активностью
        if (L)
            Log.i(TAG, "onAttach " + activity.getClass().getSimpleName());
    }

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Сообщаем системе, что у нас есть меню с параметрами.
        this.setHasOptionsMenu(true);

        if (null != savedInstanceState)
            restoreState(savedInstanceState);
        // уведомление
        if (L) Log.i(TAG, "onCreate");
    }

    // Отделяем это от методов, получающих сохраненное состояние.
    private void restoreState(Bundle savedInstanceState) {
        // НЕ ЗАБЫТЬ: автоматически сгенерированная заглушка метода.
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) { ❶

        final ListView list = (ListView) inflater.inflate(
            R.layout.list_frag_list, container, false);
        if (L) Log.i(TAG, "onCreateView");

        attachAdapter(list);
        list.setOnItemClickListener(this);

        return list;
    }
}
```

```

public void onStart() { ❷
    super.onStart();
    if (L) Log.i(TAG, "onStart");
}

public void onResume() {
    super.onResume();
    if (L) Log.i(TAG, "onResume");
}

public void onPause() {
    super.onPause();
    if (L) Log.i(TAG, "onPause");
}

public void onStop() {
    super.onStop();
    if (L) Log.i(TAG, "onStop");
}

public void onDestroyView() {
    super.onDestroyView();
    if (L) Log.i(TAG, "onDestroyView");
}

public void onDestroy() {
    super.onDestroy();
    if (L) Log.i(TAG, "onDestroy");
}

public void onDetach() {
    super.onDetach();
    if (L) Log.i(TAG, "onDetach");
}

////////////////////////////////////
// Второстепенные методы жизненного цикла
////////////////////////////////////

public void onActivityCreated() {
    // уведомление о том, что существует и объемлющая активность,
    // и ее иерархия видов
    if (L) Log.i(TAG, "onActivityCreated");
}

////////////////////////////////////
// Переопределение реализаций методов ComponentCallbacks во Fragment
////////////////////////////////////

```

```

@Override
public void onConfigurationChanged(Configuration newConfiguration) {
    super.onConfigurationChanged(newConfiguration);

    // Этого не произойдет, если мы не объявим в файле описания
    // обрабатываемых изменений.
    if (L)
        Log.i(TAG, "onConfigurationChanged");
}

@Override
public void onLowMemory() {
    // Невозможно с уверенностью сказать, будет этот код вызываться
    // до обратных вызовов или после них.
    if (L)
        Log.i(TAG, "onLowMemory");
}

////////////////////////////////////
// Код для обработки меню
////////////////////////////////////

public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.search_menu, menu);
}

////////////////////////////////////
// Код, специфичный для приложения
////////////////////////////////////

/**
 * Прикрепляем адаптер, загружающий данные в указанный список.
 * @param list
 */
private void attachAdapter(final ListView list) { ❸

    // Создаем упрощенный адаптер, загружающий массив строк.
    ArrayAdapter<String> numbers = new ArrayAdapter<String>(
        list.getContext().getApplicationContext(),
        android.R.layout.simple_list_item_1,
        new String [] {
            "one", "two", "three", "four", "five", "six"
        });

    // Приказываем списку использовать его.
    list.setAdapter(numbers);
    list.setOnItemClickListener(this);
}

```

```

////////////////////////////////////
// Реализация интерфейса OnItemClickListener
////////////////////////////////////

@Override
public void onItemClick(AdapterView<?> arg0, View view, int position,
    long id) { ❹
    // В качестве примера отправки данных нашим фрагментам
    // создадим пакет с int (целочисленными значениями) и string
    // (строковыми значениями), в зависимости от того,
    // на каком элементе был сделан щелчок.
    Bundle b = new Bundle();
    int ordinal = position + 1;
    b.putInt("place", ordinal);
    b.putString("placeName", Integer.toString(ordinal));
    TabManager.loadTabFragments(getActivity(), b);
}
}

```

Единственная функция, которую должны выполнять подклассы нашего `Fragment`, — это возврат `View` от метода `onCreateView`. Поэтому мы также определим простой макет для каждого класса `Fragment` для загрузки и возврата результирующей иерархии видов.

Компоновка фрагментов

При создании скелетных классов `Fragment`, которые содержат простые макеты, мы можем воспользоваться визуальным редактором пользовательских интерфейсов Android для того, чтобы скомпоновать каждый экран будущего приложения.

Давайте превратим каркасную модель в XML-код для файла `main.xml`, который будет использовать главная активность. Применим для этого визуальный редактор.

Компоновка фрагментов при помощи визуального редактора

На рис. 11.5 показан результат перетаскивания двух элементов `Fragment` с палитры, расположенной слева, на нужное место с последующей корректировкой их размеров. Корректировка осуществляется при помощи перетаскивания сторон прямоугольников, представляющих собой модели фрагментов.

Некоторые параметры макета удобнее редактировать в XML-виде. В следующем листинге обратите внимание на то, что в качестве значения параметра `class` задано полностью квалифицированное имя подкласса `Fragment` для данного фрагмента.

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

```

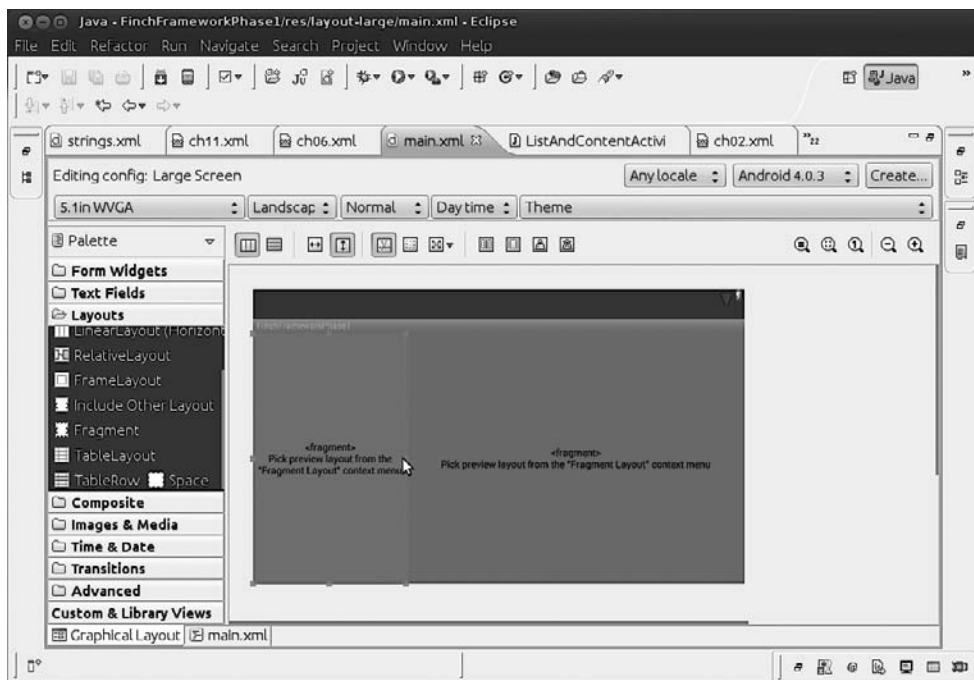


Рис. 11.5. Компоновка нескольких фрагментов на большом экране планшета

```
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:orientation="horizontal" >
```

```
<fragment
    android:id="@+id/list_frag"
    android:name=
        "com.finchframework.uiframework.QueryResultsListFragment"
    android:layout_width="250dp"
    android:layout_height="match_parent"
    class="com.finchframework.uiframework.QueryResultsListFragment" />
```

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
```

```
<fragment
    android:id="@+id/content_frag"
    android:name=
        "com.finchframework.uiframework.ContentFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```

        class="com.finchframework.uiframework.ContentFragment" />

        <fragment
            android:id="@+id/detail_frag"
            android:name="com.finchframework.uiframework.DetailFragment"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            class="com.finchframework.uiframework.DetailFragment" />
    </LinearLayout>

</LinearLayout>

```

Несколько макетов

Android позволяет разработчикам использовать в приложении альтернативные ресурсы в зависимости от размера экрана и его пиксельной плотности. Вероятно, вы уже встречались с использованием нескольких вариантов графических ресурсов в программах, рассчитанных на более высокое и сравнительно низкое разрешение экрана. Все приложения Android, создаваемые с помощью нового мастера установки, содержат каталоги для отрисовываемых объектов соответственно низкого, среднего и высокого разрешения.

В данном примере мы будем придерживаться принятых в Android соглашений об именовании при создании нескольких каталогов с макетами. Варианты макетов будут предназначаться для обычных экранов, больших экранов и больших экранов в альбомном режиме. Макет из рассмотренного выше листинга будет находиться в каталоге `layout_large`.

Создадим файл макета для экранов сотовых телефонов — в Android такой размер считается нормальным. Макет для следующей программы опять же будет называться `main.xml`, но мы определим его в каталог под названием `layout`.

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal" >

    <fragment
        android:id="@+id/list_frag"
        android:name="
            com.finchframework.uiframework.QueryResultsListFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.finchframework.uiframework.QueryResultsListFragment" />

</LinearLayout>

```

В этом файле всего один фрагмент. Система определяет, какой макет использовать, на основании того, каков размер экрана. Код выполняет в основном адап-

тацию; здесь нет кода, который бы спрашивал: «Какой макет используется?» В этом нет необходимости. Код для реагирования на действия пользователя находится в классах `Fragment`. Поэтому класс `Activity`, загружающий данные макеты, может больше ничего не выполнять.

Кроме двух предыдущих макетов, в коде примера есть также каталог `layout-large-port`, в котором находится еще одна версия `main.xml`. Этот вариант предназначен для работы на больших планшетах в альбомном режиме. В данном макете фрагменты лежат друг на друге, как карты в колоде, а не отображаются рядом друг с другом. Как и в случае с другими макетами, коду не требуется выполнять иных операций, связанных с выбором конкретного макета.

Сворачивание и разворачивание масштабируемого пользовательского интерфейса

Итак, пока у нас есть три фрагмента. В одном будет находиться список, расположенный в левой части экрана в большом макете. Два других фрагмента будут содержать вкладки, размещенные в правой части экрана. Если на экране не хватает места для обоих правых фрагментов, то будет запускаться активность, предназначенная для отображения требуемого фрагмента. И до сих пор мы не видели кода, который бы решал, какой фрагмент куда пойдет. Кроме того, мы рассмотрели применение декларативного пользовательского интерфейса, реализующего три совершенно разных представления на экране. Причем для всех трех представлений применяется один и тот же код на Java.

Определение размера и разрешения экрана

Основной аспект реализации данного приложения заключается в том, что код никак не связан с определением размера экрана, пиксельной плотностью или ориентацией. Представьте себе, насколько сложным было бы приложение, если бы вам потребовалось запрограммировать в нем логику решений: когда фрагменты могут уместиться на экране рядом друг с другом, а когда их придется сложить поверх друг друга. На самом деле мы будем придерживаться избранного подхода и дальше: наш код вообще не будет принимать решений, связанных с размером экрана. Код написан так, чтобы он мог работать при любом варианте конфигурации, указанном в файлах с макетами.

Программа будет реагировать на решение о компоновке, принимаемое *системой*. Далее приведен код главной активности из нашего примерного приложения. Как и в случае с подклассом `Fragment`, описанным выше, этот код не принимает непосредственных решений, связанных с размером экрана, а также с тем, умещаются ли на экране оба фрагмента. Код может разместить на экране и один, и два фрагмента, но не принимает решения о том, сколько фрагментов окажется в данной активности — один или два.

```
package com.finchframework.uiframework;

import com.finchframework.uiframework.R;

import android.app.ActionBar;
import android.app.Activity;
import android.content.res.Configuration;
import android.os.Bundle;
import android.util.Log;

public class ListAndContentActivity extends Activity {
    // строка для регистрации имени класса
    private final String TAG = getClass().getSimpleName();

    // включаем и отключаем регистрацию
    private final boolean L = true;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // чтобы не усложнять метод
        doCreate(savedInstanceState);

        // Если бы у нас было состояние,
        // которое требуется восстановить,
        // мы указали бы это в сообщении лога.
        if (L) Log.i(TAG, "onCreate" +
            (null == savedInstanceState ? " Restored state" : ""));
    }

    @Override
    protected void onRestart() {
        super.onRestart();
        // уведомление о том, что активность будет запущена
        if (L) Log.i(TAG, "onRestart");
    }

    @Override
    protected void onStart() {
        super.onStart();
        // уведомление о том, что активность запускается
        if (L) Log.i(TAG, "onStart");
    }

    @Override
    protected void onResume() {
        super.onResume();
        // уведомление о том, что активность будет взаимодействовать
        // с пользователем
    }
}
```

```

        if (L) Log.i(TAG, "onResume");
    }

    protected void onPause() {
        super.onPause();
        // уведомление о том, что активность прекращает взаимодействовать
        // с пользователем
        if (L) Log.i(TAG, "onPause" + (isFinishing() ? " Finishing" : ""));
    }

    @Override
    protected void onStop() {
        super.onStop();
        // уведомление о том, что активность становится невидимой
        if (L) Log.i(TAG, "onStop");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        // уведомление о том, что активность будет уничтожена
        if (L) Log.i(TAG, "onDestroy"
            // Заканчиваем?
            + (isFinishing() ? " Finishing" : ""));
    }

    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        saveState(outState);

        // вызывается при необходимости сохранить состояние
        if (L) Log.i(TAG, "onSaveInstanceState");
    }

    @Override
    protected void onRestoreInstanceState(Bundle savedInstanceState) {
        super.onRestoreInstanceState(savedInstanceState);
        if (null != savedInstanceState) restoreState(savedInstanceState);

        // Если бы у нас было состояние, которое требуется восстановить,
        // мы указали бы это в сообщении лога.
        if (L) Log.i(TAG, "onRestoreInstanceState" +
            (null == savedInstanceState ? " Restored state" : ""));
    }
}

////////////////////////////////////
// Мелкие методы жизненного цикла — вероятно, они вам не понадобятся
////////////////////////////////////

```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (null != savedInstanceState) restoreState(savedInstanceState);

    // Если бы у нас было состояние,
    // которое требуется восстановить,
    // мы указали бы это в сообщении лога.
    if (L) Log.i(TAG, "onCreate" + (null ==
        savedInstanceState ? " Restored state" : ""));
}

@Override
protected void onResume() {
    super.onResume();
    // уведомление о том, что восстановление активности завершено
    if (L) Log.i(TAG, "onPostResume");
}

@Override
protected void onUserLeaveHint() {
    super.onUserLeaveHint();
    // уведомление о том, что пользователь прекратил работать
    // с данной активностью
    if (L) Log.i(TAG, "onUserLeaveHint");
}

////////////////////////////////////
// Переопределение реализаций вызовов методов ComponentCallbacks
// в активности
////////////////////////////////////

@Override
public void onConfigurationChanged(Configuration newConfiguration) {
    super.onConfigurationChanged(newConfiguration);

    // Этого не произойдет, если мы не внесем осуществляемые изменения
    // в файл описания.
    if (L) Log.i(TAG, "onConfigurationChanged");
}

@Override
public void onLowMemory() {

    // Невозможно с уверенностью сказать, будет этот код вызываться
    // до обратных вызовов или после них.
    if (L) Log.i(TAG, "onLowMemory");
}

```

```

////////////////////////////////////
// Далее следует код, специфичный для приложения
////////////////////////////////////

/**
 * Именно здесь мы восстанавливаем сохраненное ранее состояние.
 * @param savedInstanceState – это пакет, полученный нами в ходе обратного вызова
 */
private void restoreState(Bundle savedInstanceState) {
    // Добавьте здесь ваш код для восстановления состояния.

}

/**
 * Добавьте состояние данной активности в пакет и/или отправьте данные,
 * стоящие в очереди.
 */
private void saveState(Bundle state) {
    // Здесь должен находиться ваш код для добавления состояния
    // к пакету.
}

/**
 * Выполняем нужные операции инициализации при создании экземпляра данной
 * активности.
 * @param savedInstanceState.
 */
private void doCreate(Bundle savedInstanceState) {
    setContentView(R.layout.main);

    if (null != savedInstanceState) restoreState(savedInstanceState);

    ActionBar bar = getActionBar(); ❶
    bar.setDisplayShowTitleEnabled(false);
    bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    // Инициализируем вкладки (если фрагментов с вкладками
    // не существует, код просто не срабатывает). ❷
    int names[] = {R.string.content, R.string.detail };
    int fragments[] = { R.id.content_frag, R.id.detail_frag };
    TabManager.initialize(this, 0, names, fragments);
}
}

```

Наиболее важны в этом коде следующие строки.

- ❶ Код, находящий панель действий и определяющий ряд параметров для ее потенциального использования.
- ❷ Код, инициализирующий управление вкладками. TabManager — это вспомогательный класс, рассмотренный далее в этой главе.

Делегирование задач классам фрагментов

Делегирование пользовательских взаимодействий подклассам `Fragment` в данном случае — всего лишь пример, показывающий, что при правильной организации кода отпадает необходимость принятия непосредственных решений о размере экрана. У нас на экране есть фрагменты, если они могут на нем поместиться. Когда мы оперируем виджетами внутри этих фрагментов, код для обработки наших операций находится в созданных нами подклассах `Fragment`.

```
package com.finchframework.uiframework;

import com.finchframework.uiframework.TabManager.SetData;

import android.app.ActionBar.Tab;
import android.app.ActionBar.TabListener;
import android.app.Activity;
import android.app.Fragment;
import android.app.FragmentTransaction;
import android.content.res.Configuration;
import android.os.Bundle;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;
import android.widget.FrameLayout;

public class ContentFragment extends Fragment implements
    TabListener, SetData { ❶

    // строка для регистрации имени класса
    private final String TAG = getClass().getSimpleName();

    // включаем и отключаем регистрацию
    private final boolean L = true;

    public void onAttach(Activity activity) {
        super.onAttach(activity);

        // уведомление о том, что фрагмент ассоциирован с активностью
        if (L) Log.i(TAG, "onAttach " +
            activity.getClass().getSimpleName());
    }

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // уведомление о том, что
```

```
        Log.i(TAG, "onCreate");
    }

    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        FrameLayout content = (FrameLayout)
            inflater.inflate(R.layout.content, container, false);
        if (L) Log.i(TAG, "onCreateView");
        return content;
    }

    public void onStart() {
        super.onStart();
        Log.i(TAG, "onStart");
    }

    public void onResume() {
        super.onResume();
        Log.i(TAG, "onResume");
    }

    public void onPause() {
        super.onPause();
        Log.i(TAG, "onPause");
    }

    public void onStop() {
        super.onStop();
        Log.i(TAG, "onStop");
    }

    public void onDestroyView() {
        super.onDestroyView();
        Log.i(TAG, "onDestroyView");
    }

    public void onDestroy() {
        super.onDestroy();
        Log.i(TAG, "onDestroy");
    }

    public void onDetach() {
        super.onDetach();
        Log.i(TAG, "onDetach");
    }
}

////////////////////////////////////
// Мелкие методы жизненного цикла
////////////////////////////////////
```

```

public void onActivityCreated() {
    // уведомление о том, что существует и объемлющая активность,
    // и ее иерархия видов
    Log.i(TAG, "onActivityCreated");
}

////////////////////////////////////
// Переопределение реализаций методов ComponentCallbacks во фрагменте
////////////////////////////////////

@Override
public void onConfigurationChanged(Configuration newConfiguration) {
    super.onConfigurationChanged(newConfiguration);

    // Этого не произойдет, если мы не внесем осуществляемые изменения
    // в файл описания.
    if (L) Log.i(TAG, "onConfigurationChanged");
}

@Override
public void onLowMemory() {
    // Невозможно с уверенностью сказать, будет этот код вызываться
    // до обратных вызовов или после них.
    if (L) Log.i(TAG, "onLowMemory");
}

////////////////////////////////////
// Реализация TabListener
////////////////////////////////////

@Override
public void onTabReselected(Tab tab, FragmentTransaction ft) {
    // НЕ ЗАБЫТЬ: автоматически сгенерированная заглушка метода.
}

@Override
public void onTabSelected(Tab tab, FragmentTransaction ft) {
    ft.show(this);
}

@Override
public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    ft.hide(this);
}

////////////////////////////////////
// Реализация SetData
////////////////////////////////////

```

```
@Override
public void setData(Bundle data) { ❷
    // отображаем число
    EditText t = (EditText) getActivity().findViewById(R.id.editText1);
    int i = data.getInt("place");
    t.setText(Integer.toString(i));
}
}
```

Здесь показан подкласс `Fragment`, отображающий контент в правой части экрана. В нашем приложении есть два таких подкласса, но, поскольку они не особенно отличаются, мы подробно рассматриваем здесь лишь один из них.

- ❶ Обратите внимание на то, что подкласс фрагмента реализует два интерфейса: `TabListener` и `SetData`. Интерфейс `TabListener` позволяет этому подклассу фрагмента обрабатывать пользовательские взаимодействия, когда пользователь работает с вкладками с панели действий.
- ❷ Благодаря интерфейсу `SetData` в подклассе фрагмента, содержащемся в списке в левой части большого макета либо в активности, которая запущена для отображения данного фрагмента, выводятся данные, которые требуется отобразить во фрагменте.

Обеспечение совместной работы активности, фрагмента, панели действий и нескольких макетов

В приложении, рассматриваемом в этой главе, мы собираемся организовать совместную работу активностей, фрагментов, панели действий и нескольких макетов, использующих декларативную систему пользовательского интерфейса Android. Эта задача решается для написания типового скелетного приложения, которое впоследствии может дорабатываться для решения широкого диапазона задач.

Предоставление системе Android возможности выбора макетов — важная деталь при реализации данного паттерна проектирования. Вторую часть составляет краткий вспомогательный код, используемый классами `Fragment` и `Activity` для управления вкладками на панели действий и объектами `Fragment`, выбранными посредством этих вкладок.

Панель вкладок

Вместе с классом `Fragment` панель действий позволяет создавать в Android масштабируемые пользовательские интерфейсы. Роль панели действий заключается в том, чтобы предоставить контейнер для меню, полей для ввода текста, а также для другой окантовки, в которой заключены основные элементы пользовательского интерфейса. Кроме того, панель действий обеспечивает простой способ отображения

этой части пользовательского интерфейса на экранах самого разного размера. Нам нужно организовать код нашего пользовательского интерфейса таким образом, чтобы вкладки появлялись на панели действий лишь при наличии фрагментов, которые можно выбирать при помощи этих вкладок.

Вкладки и фрагменты

Здесь мы поговорим о фрагментах, используемых для отображения списка и вкладок, о содержащемся в них коде пользовательского интерфейса, а также о вкладках панели действий, применяемых для навигации между фрагментами. В данном фреймворке мы создадим вспомогательные методы, которые могут использоваться как подклассами `Fragment`, так и `Activity` для обеспечения взаимодействий с фрагментами и вкладками. Это статические методы, поэтому мы организуем их в собственном классе, как показано в следующем листинге:

```
package com.finchframework.uiframework;

import android.app.ActionBar;
import android.app.ActionBar.Tab;
import android.app.ActionBar.TabListener;
import android.app.Activity;
import android.app.Fragment;
import android.content.Intent;
import android.os.Bundle;

public class TabManager { ❶

    /**
     * Общий вспомогательный код для инициализации вкладок, совместно
     * используемый активностями, которые содержат фрагменты и вкладки.
     *
     * Предполагается, что фрагменты ужеinstancированы и что они были
     * указаны в числе ресурсов.
     *
     * Этот код можно вызывать и не зная, присутствуют ли в макете
     * фрагменты-вкладки. Если найти такие фрагменты не удастся,
     * код просто не срабатывает.
     *
     * @param activity
     *      Активность, содержащая вкладки и соответствующие фрагменты.
     * @param defaultIndex
     *      Индекс фрагмента, отображаемого первым.
     * @param nameIDs
     *      Массив идентификаторов для имен вкладок.
     * @param fragmentIDs
     *      Массив идентификаторов для ресурсов-фрагментов.
     */
    public static void initialize(Activity activity, int defaultIndex,
        int[] nameIDs, int[] fragmentIDs) { ❷
```

```

// Сколько их у нас?
int n = nameIDs.length;
int i = 0;

// Находим как минимум один фрагмент,
// который должен реализовать TabListener.
TabListener f = (TabListener) activity.getFragmentManager()
    .findFragmentById(fragmentIDs[i]);

// Проверка на ноль: если такие фрагменты отсутствуют,
// вызов не наносит никакого вреда.
if (null != f) {

    // Получаем панель действий и удаляем имеющиеся вкладки.
    ActionBar b = activity.getActionBar();
    b.removeAllTabs();

    // Создаем новые вкладки, присваиваем им метки и слушатели.
    for (; i < n; i++) {
        f = (TabListener) activity.getFragmentManager()
            .findFragmentById(fragmentIDs[i]);
        Tab t = b.newTab().setText(nameIDs[i]).setTag(f)
            .setTabListener(f);
        b.addTab(t);
    }
    b.getTabAt(defaultIndex).select();
}
}

/**
 * Если в данной активности у нас есть вкладки и фрагменты,
 * передаем данные пакета фрагментам. В противном случае
 * запускаем активность, которая должна содержать фрагменты.
 *
 * @param activity
 * @param data
 */
public static void loadTabFragments(Activity activity,
    Bundle data) { ❸
    int n = activity.getActionBar().getTabCount();
    if (0 != n) {
        doLoad(activity, n, data);
    } else {
        activity.startActivity(new Intent(activity,
            ContentControlActivity.class).putExtras(data));
    }
}

/**
 * интерфейс для передачи данных фрагменту
 */

```

```

public interface SetData {
    public void setData(Bundle data);
}

/**
 * Перебираем вкладки, получаем их метки и используем
 * их как ссылки на фрагмент, через которые передаем
 * фрагментам информацию из пакета.
 *
 * @param activity
 * @param n
 * @param data
 */
private static void doLoad(Activity activity, int n, Bundle data) {
    int i;
    ActionBar actionBar = activity.getActionBar();

    for (i = 0; i < n; i++) {
        SetData f = (SetData) actionBar.getTabAt(i).getTag();
        f.setData(data);
    }
    actionBar.selectTab(actionBar.getTabAt(0));
}
}

```

- ❶ В классе `Fragment`, который можно выбрать при помощи вкладки на панели действий, также реализован интерфейс `SetData`. Здесь есть определение данного интерфейса вместе со вспомогательным методом, обертывающим данный интерфейс специальным кодом. Этот код абстрагирует разницу между случаями, когда, с одной стороны, одна активность содержит все фрагменты и, с другой стороны, для отображения фрагментов запускается новая активность.
- ❷ Метод `initialize` проверяет наличие фрагментов для инициализации, которые ассоциированы с вкладками. В предыдущем листинге мы видели, что подклассы `Fragment`, которые можно выбирать при помощи вкладок на панели действий, содержат код для обработки взаимодействий с вкладками. Именно этот метод `initialize` соединяет вкладки с фрагментами и при этом происходит вызов интерфейса. Но прежде, чем произвести такую инициализацию, метод проверяет, есть ли вообще в наличии фрагменты для инициализации. Это еще один пример адаптации к макетам, как с фрагментами, так и без них.
- ❸ Метод `LoadTabFragments` — единственный фрагмент всего нашего примера, о котором с натяжкой можно сказать, что он «принимает решение». Если имеются фрагменты, которые нужно наполнить данными, то этот код запускает активность, в которой, как мы знаем, будут отображаться фрагменты. И если нам необходимо запустить такую активность, мы передаем ей данные посредством поля `extras` в аргументе `Intent`, относящемся к `startActivity`.

Другая активность

Хорошо, а что делать, если мы имеем дело с маленьким экраном? В предыдущем листинге мы видели, что метод `loadTabFragments` вызывает запуск активности в случае, если фрагменты, выбираемые через вкладки, не находятся на экране. При этом активность «может быть уверена», что в ней не будет ничего, кроме двух фрагментов, а также вкладок, которые не уместились в другой активности, запустившей ее. Это еще один пример узнавания геометрии экрана без необходимости запрашивания дополнительной информации или каких-либо решений.

Вот код этой активности:

```
package com.finchframework.uiframework;
import android.app.ActionBar;
import android.app.Activity;
import android.content.res.Configuration;
import android.os.Bundle;
import android.util.Log;

public class ContentControlActivity extends Activity {

    // строка для регистрации имени класса
    private final String TAG = getClass().getSimpleName();

    // включаем и отключаем регистрацию
    private final boolean L = true;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // чтобы не усложнять метод
        doCreate(savedInstanceState);

        // Если бы у нас было состояние, которое требуется восстановить,
        // мы указали бы это в сообщении лога.
        if (L) Log.i(TAG, "onCreate" +
            (null == savedInstanceState ? " Restored state" : ""));
    }

    @Override
    protected void onRestart() {
        super.onRestart();
        // Уведомление о том, что активность будет запущена
        if (L) Log.i(TAG, "onRestart");
    }

    @Override
    protected void onStart() {
        super.onStart();
    }
}
```

```
// уведомление о том, что активность запускается
if (L) Log.i(TAG, "onStart");
}

@Override
protected void onResume() {
    super.onResume();
    // уведомление о том, что активность будет взаимодействовать
    // с пользователем
    if (L) Log.i(TAG, "onResume");
}

protected void onPause() {
    super.onPause();
    // уведомление о том, что активность прекращает взаимодействовать
    // с пользователем
    if (L) Log.i(TAG, "onPause" + (isFinishing() ? " Finishing" : ""));
}

@Override
protected void onStop() {
    super.onStop();
    // уведомление о том, что активность становится невидимой
    if (L) Log.i(TAG, "onStop");
}

@Override
protected void onDestroy() {
    super.onDestroy();
    // уведомление о том, что активность будет уничтожена
    if (L) Log.i(TAG, "onDestroy"
        // Заканчиваем?
        + (isFinishing() ? " Finishing" : ""));
}

@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    saveState(outState);
    // вызывается при необходимости сохранить состояние
    if (L) Log.i(TAG, "onSaveInstanceState");
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    if (null != savedInstanceState) restoreState(savedInstanceState);

    // Если бы у нас было состояние, которое требуется восстановить,
    // мы указали бы это в сообщении лога.
```

```

        if (L) Log.i(TAG, "onRestoreInstanceState" +
            (null == savedInstanceState ? " Restored state" : ""));
    }

    //////////////////////////////////////
    // Мелкие методы жизненного цикла – вероятно, они вам не понадобятся
    //////////////////////////////////////

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (null != savedInstanceState) restoreState(savedInstanceState);

        // Если бы у нас было состояние, которое требуется восстановить,
        // мы указали бы это в сообщении лога.
        if (L) Log.i(TAG, "onCreate" +
            (null == savedInstanceState ? " Restored state" : ""));
    }

    @Override
    protected void onStart() {
        super.onStart();
        // уведомление о том, что восстановление активности завершено
        if (L) Log.i(TAG, "onStart");
    }

    @Override
    protected void onResume() {
        super.onResume();
        // уведомление о том, что пользователь прекратил работать
        // с данной активностью
        if (L) Log.i(TAG, "onResume");
    }

    @Override
    protected void onPause() {
        super.onPause();
        // уведомление о том, что пользователь прекратил работать
        // с данной активностью
        if (L) Log.i(TAG, "onPause");
    }

    @Override
    protected void onStop() {
        super.onStop();
        // уведомление о том, что пользователь прекратил работать
        // с данной активностью
        if (L) Log.i(TAG, "onStop");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        // уведомление о том, что пользователь прекратил работать
        // с данной активностью
        if (L) Log.i(TAG, "onDestroy");
    }

    @Override
    public void onConfigurationChanged(Configuration newConfiguration) {
        super.onConfigurationChanged(newConfiguration);
        // Этого не произойдет, если мы не внесем осуществляемые изменения
        // в файл описания.
        if (L) Log.i(TAG, "onConfigurationChanged");
    }

    @Override
    public void onLowMemory() {

```

```

        // Невозможно с уверенностью сказать, будет этот код вызываться
        // до обратных вызовов или после них.
        if (L) Log.i(TAG, "onLowMemory");
    }

    //////////////////////////////////////
    // Далее следует код, специфичный для приложения
    //////////////////////////////////////

    /**
     * Именно здесь мы восстанавливаем сохраненное ранее состояние.
     * @param savedInstanceState – это пакет, полученный нами в ходе обратного вызова
     */
    private void restoreState(Bundle savedInstanceState) {
        // Добавьте здесь ваш код для восстановления состояния.
    }

    /**
     * Добавьте состояние данной активности в пакет и/или отправьте данные,
     * стоящие в очереди.
     */
    private void saveState(Bundle state) {
        // Здесь должен находиться ваш код для добавления состояния
        // к пакету.
    }

    /**
     * Выполняем нужные операции инициализации при создании экземпляра
     * данной активности.
     * @param savedInstanceState
     */
    private void doCreate(Bundle savedInstanceState) {
        setContentView(R.layout.content_control_activity);

        if (null != savedInstanceState) restoreState(savedInstanceState);

        ActionBar bar = getActionBar();
        bar.setDisplayShowTitleEnabled(false);
        bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

        // Инициализируем вкладки.
        int names[] = {R.string.content, R.string.detail };
        int fragments[] = { R.id.content_frag, R.id.detail_frag };
        TabManager.initialize(this, 0, names, fragments);

        // Загружаем данные, если таковые имеются.
        Bundle b = getIntent().getExtras();
        TabManager.loadTabFragments(this, b);
    }
}

```

В этом коде наиболее важны следующие строки.

- ❶ Эта активность вызывает метод `Tabmanager.initialize`, точно как и предыдущая. Поскольку методы `TabManager` работают с любым макетом, который может выбрать система для любого из классов активностей, данный вызов не связан с какой-либо условной логикой.
- ❷ Аналогично вызов метода `TabManager.loadTabfragments` во всем идентичен тому, который вызвал запуск данной активности, за тем исключением, что в контексте данной активности вкладки и фрагменты будут существовать только при условии, что в них будут загружены указанные данные.

12

Использование поставщиков содержимого

Когда приложения Android совместно применяют данные, они пользуются API поставщиков содержимого, чтобы предоставлять информацию, находящуюся в их базе данных. Например, поставщик содержимого, отвечающий за список контактов в Android, позволяет неограниченному количеству приложений многократно использовать контакты, хранящиеся в долговременной памяти на платформе Android. Просто активировав этот поставщик содержимого, приложение может интегрировать доступ к пользовательским контактам, сохраненным на локальном устройстве и синхронизированным с облаком Google. Приложения могут записывать и считывать данные, относящиеся к поставщикам содержимого, не предоставляя собственный код, применяемый для управления базой данных. Таким образом, поставщики содержимого предоставляют разработчику замечательные возможности, позволяющие с легкостью писать приложения, которые обладают усложненными механизмами управления данными. Как правило, в таких приложениях практически не приходится писать собственного кода, обслуживающего долговременное хранение информации.

API поставщиков содержимого позволяет клиентским приложениям запрашивать у операционной системы нужные данные при помощи универсального идентификатора ресурса (URI) — подобно тому как браузер запрашивает информацию из Интернета. Клиент не знает, какое именно приложение будет предоставлять данные в ответ на запрос по URI. Клиент просто предоставляет URI операционной системе, и уже сама платформа должна запустить подходящее приложение, чтобы предоставить требуемый результат. Платформа также предоставляет механизм разрешений, позволяющий клиентам ограничивать доступ к данным поставщика содержимого.

API поставщика содержимого предоставляет к совместно используемой информации полный доступ для создания, считывания, обновления и удаления данных. Это означает, что приложения могут использовать запросы, передаваемые по URI, для следующих целей:

- создавать новые записи;
- получать одну, две записи или ограниченное множество записей;
- обновлять записи;
- удалять записи.

В этой главе мы покажем, как написать собственный поставщик содержимого. Для этого мы изучим внутренние механизмы поставщика содержимого SimpleFinchVideoContentProvider, который возьмем в качестве примера. Он входит в дерево исходников Finch. Все ссылки на файлы содержатся в исходном каталоге этой главы. То есть когда мы говорим в этом разделе о файле `AndroidManifest.xml`, то имеем в виду файл `$(FinchVideo)/AndroidManifest.xml`. Этот код мы используем, чтобы объяснить, как создается поставщик содержимого. Для этого мы реализуем каждый из методов, которые требует определить основной API поставщиков содержимого — класс `ContentProvider`. Мы также расскажем о том, как интегрировать в этот поставщик содержимого базу данных SQLite. Будет описано, как реализовать базовую функцию поставщика содержимого, которая предназначена для ассоциирования URI, указывающих на данные из строк базы данных. Вы увидите, как в поставщике содержимого инкапсулированы функции долговременного хранения данных и как он позволяет приложению совместно использовать данные между несколькими процессами, когда вы объявляете свой поставщик в файле `AndroidManifest.xml`. Мы покажем, как привязывать данные поставщика содержимого к компонентам пользовательского интерфейса Android, и завершим, таким образом, обсуждение архитектуры MVC («Модель-вид-контроллер»), о которой мы до сих пор говорили в книге. Наконец, мы создадим активность для просмотра данных, которая будет автоматически обновляться в ответ на изменения, происходящие с данными.



В этой главе мы исходим из того, что локальный поставщик содержимого использует базу данных SQLite. Учитывая, что мы располагаем методами `query`, `insert`, `update` и `delete`, относящимися к API, немного неправильно было бы говорить об отображении их на что-то еще, хотя теоретически API может хранить и получать информацию из любого источника, который поддерживает требуемые операции, как, например, обычный файл.

Эту главу можно считать вводной, и затронутые в ней темы будут подробнее рассмотрены в следующей главе. Там мы покажем, как расширить и оптимизировать саму концепцию поставщика содержимого. В процессе работы вы научитесь извлекать максимальную пользу из API поставщика содержимого, чтобы обеспечить интеграцию сетевых сервисов с передачей состояния представления (RESTful). Такая простая архитектура позволяет избежать многих ошибок, распространенных в мобильном программировании, даже притом, что она строится только на базовых компонентах Android. **Вы увидите, что такой подход логически выводит нас к созданию мобильной архитектуры, позволяющей значительно повысить надежность и производительность приложений Android.**

Мы подробно изучим приложение для просмотра списков видео вместе с выводом метаданных, которое позволяет в упрощенном виде проиллюстрировать данную архитектуру. Это приложение использует предложенный нами подход, выполняя загрузку, синтаксический разбор и кэширование записей о видеороликах с YouTube, получаемых от веб-службы с передачей состояния представления, расположенной по адресу <http://gdata.youtube.com>. Мы просто воспользуемся gData в качестве примера RESTful-службы, которую сможем интегрировать в поставщик содержимого Android. Пользовательский интерфейс приложения будет применять поставщики содержимого для динамического отображения записей о видео, по мере того как они

будут загружаться из сети и проходить синтаксический разбор. Этот подход можно будет использовать для интеграции множества веб-сервисов, доступных в Интернете, в вашем приложении на основе Android. Кстати, по ссылке gData имеется отличный демонстрационный пример от Google, мы советуем вам с ним ознакомиться.

Понятие о поставщиках содержимого

Поставщики содержимого заключают в себе такой функционал управления данными, что другие части приложения, например вид и контроллер, могут не заниматься долговременным хранением данных приложения. Сформулируем эту мысль иначе: поставщик содержимого обеспечивает долговременное хранение данных, так как вид и контроллер не стоит нагружать этими задачами. Специализированные программные уровни, не пытающиеся выполнять задачи, решаемые на других уровнях, — характерная черта качественного кода. Ошибки и излишняя сложность возникают там, где отдельные уровни ПО пытаются решать задачи, находящиеся вне их компетенции. То есть пользовательский интерфейс должен состоять только из хорошо скомпонованных компонентов пользовательского интерфейса, идеально подходящих для сбора событий от конечного пользователя. Хорошо написанный контроллер приложения будет содержать только логику предметной области, то есть логику мобильного приложения. И применительно к теме этой главы упростить код удастся тогда, когда оба типа кода могут передать задачу долговременного хранения данных третьей логической стороне: поставщикам содержимого. Вспоминая раздел «SQL и модель построения архитектуры вокруг базы данных в приложениях Android» главы 9, можно сказать, что поставщики содержимого хорошо подходят для реализации такой модели данных, центром которой не является документ.

При использовании поставщиков содержимого приложению не требуется открывать собственных таблиц SQLite, поскольку этот аспект будет обрабатываться за интерфейсом поставщика содержимого с использованием таблиц этого поставщика содержимого. Ранее для того, чтобы совместно пользоваться данными, мобильным приложениям приходилось сохранять эти данные в файлах локальной файловой системы. Формат конфигурации этих данных определялся приложением. Напротив, при работе с Android приложения вполне могут работать, полагаясь лишь на те ресурсы для хранения данных, которые предоставляет поставщик содержимого.

Прежде чем углубиться в изучение SimpleFinchVideoContentProvider, мы сделаем обзор простого приложения для просмотра описаний видео, а также в целом опишем задачи, связанные с реализацией поставщика содержимого.

Реализация поставщика содержимого

Чтобы воспользоваться преимуществами такой проектной архитектуры, нужно написать свой поставщик содержимого. Для этого требуется выполнить следующие задачи.

- Создать API поставщика содержимого, общедоступный для использования клиентом. Для этого нужно сделать следующее:

- определить `CONTENT_URI` для вашего поставщика содержимого;
 - создать названия столбцов для обмена информацией с клиентами;
 - объявить общедоступные статические объекты `String`, которыми клиенты будут пользоваться для указания столбцов;
 - определить типы `MIME` для любых новых типов данных.
- Реализовать собственный поставщик содержимого. Для этого нужно:
- расширить основной API поставщиков содержимого, класс `ContentProvider`, для создания собственной реализации поставщика содержимого;
 - задать `URI` поставщика содержимого;
 - создать базу данных `SQLite` и ассоциированные с ней курсоры для хранения данных поставщика содержимого;
 - использовать курсоры для предоставления клиентам доступа к данным и в то же время для поддержки динамического обновления данных;
 - определять процесс, при помощи которого двоичные данные будут возвращаться клиенту;
 - реализовать базовые методы для работы с данными `query`, `insert`, `update` и `delete`, относящиеся к `Cursor` и предназначенные для возврата клиенту.
- Обновить файл `AndroidManifest.xml` для обновления своего поставщика содержимого в теге `<provider>`.

Когда мы закончим обсуждение базовой реализации поставщика содержимого, мы опишем задачи, связанные с использованием поставщиков содержимого. Это понадобится нам для разработки более совершенной сетевой архитектуры, о которой мы говорили выше.

Просмотр описаний видео при помощи программы Finch

Программа `Finch`, предназначенная для просмотра информации о видео, позволяет пользователю создавать список метаданных, описывающих видеоролики. Мы поговорим о двух вариантах такого приложения и о двух вариантах поставщиков содержимого, лежащих в основе подобных приложений. Первая версия, показанная в этой главе, — простая программа для построения списка видео, использующая поставщик содержимого `SimpleFinchVideoContentProvider`. Эта программа разработана как учебный пример, который поможет вам научиться реализовывать ваш поставщик содержимого. Вторая версия приложения, к которой мы перейдем в следующей главе, использует немного более сложный поставщик содержимого, обладающий дополнительной способностью получать контент из онлайн-сервиса **YouTube**, предназначенного для поиска видеофайлов. Вторая версия приложения также позволяет кэшировать результаты и показывать эскизы видео.

Теперь подробно исследуем первое приложение. В нем есть одна активность: `SimpleFinchVideoActivity`, позволяющая пользователю создавать и выстраивать в виде

списка метаданные к его собственным видеофайлам (например, название видеоролика, его описание, URI и идентификатор). Эта активность показана на рис. 12.1.

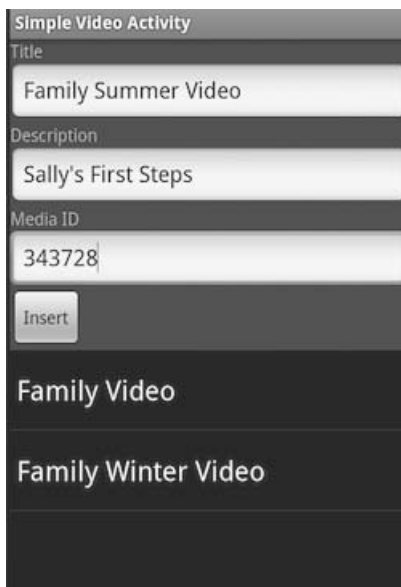


Рис. 12.1. Активность для простого поставщика содержимого, позволяющая пользователям вводить метаданные для собственных видеороликов

Чтобы работать с этим приложением, просто введите нужные данные для записи видео, а потом нажмите кнопку **Insert** (Вставить). Список под текстовыми полями использует принятый в Android шаблон MVC («Модель-вид-контроллер») для автоматического обновления вида с данными.

Простая база данных для видео

Для сохранения данных, которые вы вводите в это приложение, класс `SimpleFinchVideoContentProvider` создает собственную базу данных при помощи следующего предложения на языке SQL:

```
CREATE TABLE video (_id INTEGER PRIMARY KEY, title TEXT,
                    decription TEXT, uri TEXT);
```

Столбец `_id` необходим для работы с системой курсоров **Android**. Он предоставляет уникальный идентификатор для строки в курсоре, а также уникальный идентификатор для объекта в базе данных. Таким образом, необходимо определить этот столбец с атрибутами `SQL INTEGER PRIMARY KEY AUTOINCREMENT`, чтобы гарантировать, что определенное значение будет уникальным.

В столбцах `title` и `description` сохраняются соответственно заголовок и данные, описывающие видео. В столбце `uri` содержится медийный уникальный идентификатор ресурса, который может использоваться для воспроизведения видеоролика, присутствующего в списке, в рабочей версии данного приложения.

Структура простой версии кода

В этом разделе кратко охарактеризуем важные файлы, входящие в состав простого видеоприложения Finch.

- `AndroidManifest.xml`. Мы создали описание для простого видеоприложения, использующего поставщик содержимого. В файле описания будет содержаться ссылка на активность `SimpleFinchVideoActivity`, а также на поставщик содержимого `SimpleFinchVideoContentProvider`.
- `$(FinchVideo)/src/com/oreilly/demo/pa/finchvideo/FinchVideo.java`. Класс `FinchVideo` содержит атрибут `AUTHORITY` (его мы рассмотрим позже) и класс `SimpleVideo`, определяющий имена столбцов поставщика содержимого. Ни в классе `FinchVideo`, ни в классе `SimpleVideo` не содержится никакого исполняемого кода.
- `$(FinchVideo)/src/com/oreilly/demo/pa/finchvideo/provider/SimpleFinchVideoContentProvider.java`. Класс `SimpleFinchVideoContentProvider` — это поставщик содержимого для простой базы данных о видео. Он обрабатывает запросы на получение ресурсов по URI так, как этого требует простое видеоприложение. Этот файл мы будем изучать в первой части данной главы.
- `$(FinchVideo)/src/com/oreilly/demo/pa/finchvideo/SimpleFinchVideoActivity.java`. Класс `SimpleFinchVideoActivity` — это активность, позволяющая пользователю просмотреть список видео.

Определение общедоступного API поставщика содержимого

Хотя в главе 3 мы уже говорили о том, как клиенты используют поставщики содержимого, здесь мы подробнее расскажем будущим авторам поставщиков содержимого о том, как полностью реализовать общедоступный интерфейс (API) такого поставщика. Чтобы клиенты могли использовать поставщик содержимого, необходимо создать общедоступный класс API, содержащий набор констант. Клиенты смогут применять эти константы для доступа к полям столбцов объектов `Cursor`, возвращаемых посредством метода запроса вашего поставщика. Кроме того, в этом классе будет определяться URI источника вашего поставщика содержимого, фактически — основа всей используемой в поставщике системы обмена информацией по URI. Наш класс `FinchVideo.SimpleVideos` предоставляет API для класса `SimpleFinchVideo`.

Сначала мы объясним каждый компонент класса, в частности сообщим базовую информацию о его полях, а потом покажем его полный листинг.

Определение CONTENT_URI

Чтобы клиентское приложение могло запрашивать данные у поставщика содержимого, оно должно передать URI, указывающий релевантные данные одному из методов преобразователя содержимого (`content resolver`). Эти методы относятся к преобразователю содержимого (`content resolver`) системы Android. Методы называются `query`, `insert`, `update` и `delete`. Они отражают методы того преобразователя

содержимого, который будет определен в разделе «Написание и интеграция поставщика содержимого» данной главы. После такой активизации преобразователь содержимого будет использовать строку источника (authority string) для сопоставления входящего URI с CONTENT_URI каждого известного ему поставщика содержимого, чтобы найти поставщик, подходящий для конкретного клиента. Итак, CONTENT_URI определяет тип **URI (уникального идентификатора ресурса), который может обработать ваш поставщик содержимого.**

CONTENT_URI состоит из следующих частей.

- content:// — префикс, сообщаящий фреймворку Android о том, что необходимо найти поставщик содержимого для преобразования данного URI.
 - Источник (authority) — данная строка уникально идентифицирует поставщик содержимого и состоит из двух частей: организационной части и идентификатора поставщика. Организационная часть уникально идентифицирует ту организацию, которая создала поставщик содержимого. Идентификатор поставщика означает конкретный поставщик содержимого, созданный этой организацией. В поставщиках содержимого, встроенных в операционную систему Android, организационная часть опускается. Например, встроенный источник media, возвращающий одно изображение или более, не имеет организационной части. Напротив, любые поставщики содержимого, созданные разработчиками, не работающими в команде создателей Android, должны определять обе части этой строки. Таким образом, источник нашего простого видеоприложения Finch имеет вид com.oreilly.demo.pa.finchvideo.SimpleFinchVideo. Организационная часть — это com.oreilly.demo.pa.finchvideo, а идентификатор поставщика содержимого — SimpleFinchVideo. В документации **Google упоминается, что оптимальное решение при написании источника в CONTENT_URI — это полностью квалифицированное имя класса, реализующего поставщик содержимого.**
- Источник уникально идентифицирует конкретный поставщик содержимого. Android будет вызывать этот поставщик содержимого в ответ на получаемые запросы, которые приходится обрабатывать.

- Путь — поставщику содержимого не обязательно интерпретировать остальную часть URI, но он должен подчиняться ряду требований.
 - Если поставщик содержимого может возвращать данные нескольких типов, то URI должен быть составлен так, чтобы в определенной части пути был указан тип возвращаемых данных.

Например, встроенный в систему Android поставщик содержимого contacts может возвращать разнообразную информацию: имена людей, номера телефонов, способы связи и т. д. Поставщик содержимого, работающий с контактами, использует строки, которые входят в состав URI, для различения того, какой тип данных запрашивается пользователем. Например, при запросе определенного человека, отмеченного в списке контактов, уникальный идентификатор ресурса будет выглядеть примерно так:

content://contacts/people/1

При запросе определенного телефонного номера URI может выглядеть так:

```
content://contacts/people/1/phone/3
```

В первом случае MIME-тип возвращаемых данных будет `vnd.android.cursor.item/person`, а во втором — `vnd.android.cursor.item/phone`.

- Поставщик содержимого должен быть способен вернуть как один элемент, так и множество идентификаторов элементов. Поставщик содержимого возвращает одиночный элемент, когда идентификатор элемента оказывается в последней части URI. Вспомнив наш предыдущий пример, отметим, что уникальный идентификатор ресурса `content://contacts/people/1/phone/3` возвратил единственный телефонный номер типа `vnd.android.cursor.item/phone`. Если бы мы имели дело с URI `content://contacts/people/1/phone`, то приложение вернуло бы список всех телефонных номеров определенного человека — того, который имеет идентификационный номер 1, — а MIME-тип возвращаемых данных был бы `vnd.android.cursor.dir/phone`.

Как было указано выше, поставщики содержимого могут интерпретировать части пути уникального идентификатора ресурса таким образом, как это требуется им для работы. Это означает, что часть пути может использовать элементы, входящие в состав пути, для того, чтобы фильтровать данные, возвращаемые вызывающей стороне. Например, встроенный в систему поставщик содержимого `media` может возвращать как внешние, так и внутренние данные, в зависимости от того, содержится ли в уникальном идентификаторе ресурса слово `external` (внешний) или `internal` (внутренний).

Полный идентификатор `CONTENT_URI` для простого видеоприложения **Finch** имеет вид `content://com.oreilly.demo.pa.finchvideo.SimpleFinchVideo/video`.

`CONTENT_URI` должен иметь тип `public static final Uri`. Он определяется в классе `FinchVideo` нашего простого видеоприложения. В общедоступном классе `API` мы сначала расширим класс `BaseColumns`, а потом определим строку, называемую `AUTHORITY`:

```
public final class FinchVideo.SimpleVideos extends BaseColumns {  
    public static final String SIMPLE_AUTHORITY =  
        "com.oreilly.demo.pa.finchvideo.FinchVideo";  
}
```

Затем мы определим сам `CONTENT_URI`:

```
public static final class FinchVideo.SimpleVideos implements BaseColumns {  
    public static final Uri CONTENT_URI =  
        Uri.parse("content://" + AUTHORITY + "/video");  
}
```

Проще говоря, при определении данного уникального идентификатора ресурса нам просто требуется выбрать строку источника, в качестве организационной части которой должен использоваться пакет **Java**, играющий для приложения роль идентификатора. В данном качестве нам лучше подойдет общедоступный пакет `API`, а не пакет реализации. Об этом мы говорили в подразделе «Пакеты **Java**» раздела «Область видимости» главы 2. Идентификатор поставщика содержимого — это просто

имя класса данного поставщика содержимого. URI поставщика для простого видеоприложения Finch имеет следующий вид:

```
"content://" + FinchVideo.FinchVideoContentProvider.SIMPLE_AUTHORITY + "/" +  
    FinchVideo.SimpleVideos.VIDEO
```

Создание имен столбцов

Процедура обмена данными между поставщиками содержимого и их клиентами принципиально похожа на то, как база данных SQL обменивается данными с приложениями, использующими базу данных. И в том и в другом случае для обмена информацией используются курсоры, наполняемые строками и столбцами, содержащими данные. Поставщик содержимого должен определять имена столбцов, которые он поддерживает, так же как приложения базы данных определяют такие столбцы для себя. Когда поставщик содержимого использует в качестве хранилища информации базу данных SQLite, очевидно, следует называть столбцы поставщика содержимого так же, как и столбцы базы данных. В SimpleFinchVideoContentProvider ситуация обстоит именно так. Поэтому между столбцами SimpleFinchVideoContentProvider и столбцами основной базы данных не требуется никакого отображения.



Не все приложения предоставляют все свои данные клиентам поставщика содержимого, а некоторые более сложные приложения, возможно, будут создавать производные виды, доступные для клиентов поставщика содержимого. Проекционная карта, которая будет описана в подразделе «Класс SimpleFinchVideoContentProvider и переменные экземпляров» раздела «Полный код поставщика содержимого: поставщик SimpleFinchVideoContentProvider» далее, используется для обработки этих сложных моментов.

Объявление строк описания столбцов

Столбцы поставщика содержимого SimpleFinchVideoProvider определяются в классе FinchVideo.SimpleVideos, рассматриваемом в этом разделе. Каждый поставщик содержимого должен определять столбец `_id`, в котором будет содержаться порядковый номер каждой строки. Значение каждого `_id` должно быть уникальным в рамках поставщика содержимого. Клиент будет прикреплять этот номер к URI записи типа `vnd.android.cursor.item` при попытке запросить отдельно взятую запись.

Когда поставщик содержимого опирается при работе на базу данных SQLite, как в случае с SimpleFinchVideoProvider, столбец `_id` должен иметь тип `INTEGER PRIMARY KEY AUTO INCREMENT`. Таким образом, все строки будут иметь уникальные номера `_id`, и эти `_id` не будут использоваться многократно, даже в случае удаления строк. Так можно поддерживать целостность ссылочных данных (**referential integrity**), поскольку мы гарантируем, что каждая новая строка получает `_id`, не использовавшийся ранее. Если `_id` строк могут применяться многократно, существует вероятность того, что кэшированные URI будут указывать на неверные данные.

Ниже приведен полный программный листинг API простого поставщика содержимого (видео), используемого в программе Finch, — класс FinchVideo.SimpleVideos.

Обратите внимание на то, что мы включили в код только те константы, которые служат для достижения целей, описанных выше. Здесь мы не занимаемся определением констант реализации поставщика содержимого, поскольку они не несут никакой пользы для клиента и могут «завязать» клиента на использование конкретной реализации поставщика содержимого. Мы стремимся хорошо спроектировать программу и гарантировать, что ее уровни будут хорошо отделяться друг от друга там, где клиенты не должны иметь непосредственных компиляционных зависимостей от классов реализации поставщика содержимого. Полный листинг общедоступного API для API поставщика видео для программы Finch приведен ниже:

```
/**
 * столбцы Simple Videos
 */
public class FinchVideo {
    public static final class SimpleVideos implements BaseColumns {
        // этот класс не может быть инстанцирован
        private SimpleVideos() {}

        // uri со ссылкой на все видеоролики
        public static final Uri VIDEOS_URI = Uri.parse("content://" +
            SIMPLE_AUTHORITY + "/" + SimpleVideos.VIDEO);

        /**
         * content:// style URL для этой таблицы
         */
        public static final Uri CONTENT_URI = VIDEOS_URI; ❶

        /**
         * MIME-тип {@link #CONTENT_URI}, предоставляющий каталог записей
         */
        public static final String CONTENT_TYPE =
            "vnd.android.cursor.dir/vnd.finch.video"; ❷

        /**
         * MIME-тип {@link #CONTENT_URI} подкаталога отдельно взятого
         * видео.
         */
        public static final String CONTENT_VIDEO_TYPE =
            "vnd.android.cursor.item/vnd.finch.video";

        ❸
        /**
         * сам видеофайл
         * <P>Type: TEXT</P>
         */
        public static final String VIDEO = "video";

        /**
         * имя столбца для заголовка видеофайла
         * <P>Type: TEXT</P>
         */
    }
}
```

```

    public static final String TITLE = "title";

    /**
     * имя столбца для описания видео
     */
    public static final String DESCRIPTION = "description";

    /**
     * имя столбца для media uri
     */
    public static final String URI = "uri";

    /**
     * уникальный идентификатор для медиаэлемента
     */
    public static final String MEDIA_ID = "media_id";
}

...
// API для FinchVideo.Videos также определяется в этом классе
}

```

Рассмотрим пояснения к коду.

- ❶ Мы используем `VIDEOS_URI` для определения значения `CONTENT_URI`. Уникальный идентификатор видеофайлов содержит `URI` этого содержимого в описанном выше виде.
- ❷ Это `MIME`-тип записей видео, которые будет хранить наш поставщик содержимого. В подразделе «Реализация метода `getType`» раздела «Полный код поставщика содержимого: поставщик `SimpleFinchVideoContentProvider`» далее мы расскажем, как наш поставщик содержимого использует этот тип.
- ❸ Это названия столбцов, которыми клиенты могут пользоваться для доступа к значениям, содержащимся в объектах `Cursor`, создаваемых нашим поставщиком.

Написание и интеграция поставщика содержимого

Теперь, когда мы рассмотрели общую структуру простого приложения, составляющего список видеофайлов, и обеспечили для клиентов способ доступа к нашему поставщику содержимого, изучим, как приложение реализует и использует `SimpleFinchVideoContentProvider`.

Типичные задачи, решаемые поставщиком содержимого

В следующих разделах приводится общее руководство по решению задач, связанных с написанием поставщика содержимого. Здесь будет сделано введение в паттерн MVC («Модель-вид-контроллер»), применяемый в Android, и завершено объяснение кода `SimpleFinchVideoContentProvider`.

Дополнение поставщика содержимого

Приложения расширяют класс `ContentProvider` для управления уникальными идентификаторами ресурсов, относящимися к данным конкретных типов — например, MMS-сообщениям, изображениям, видео и т. д. Допустим, при работе с классом поставщика содержимого, который обрабатывает видеoinформацию, метод `ContentProvider.insert` вставляет данные, описывающие видео, в столбцы таблицы базы данных `SQLite`, подходящие для такой информации. В частности, это могут быть столбцы с заголовком, описанием видео и т. п.

Начнем написание поставщика содержимого с реализации следующих двух методов:

- `onCreate` — предоставляет точку привязки, позволяющую вашему поставщику содержимого инициализироваться. Любой код, который вы хотите выполнить однократно, например код, устанавливающий соединение с базой данных, должен находиться в этом методе;
- `String getType(Uri uri)` — возвращает MIME-тип данных, предоставляемых поставщиком содержимого по данному уникальному идентификатору ресурса. Идентификатор (URI) приходит от клиентского приложения, заинтересованного в доступе к данным.

На следующем этапе реализации переопределяются основные методы поставщика содержимого, обеспечивающие доступ к данным.

- `insert(Uri uri, ContentValues values)` — вызывается, когда клиентскому коду требуется вставить информацию в базу данных, обслуживаемую вашим поставщиком содержимого. Как правило, реализация этого метода прямым или косвенным образом вызывает операцию вставки, применяемую к базе данных.
- `Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)` — вызывается каждый раз, когда клиенту требуется считать информацию из базы данных поставщика содержимого. Как правило, здесь получение данных происходит при помощи предложения `SELECT` на языке `SQL`, а в ответ на него возвращается курсор, содержащий запрошенные данные. Разработчики вызывают этот метод опосредованно, при помощи метода `managedQuery Activity`, либо применяют метод `startManagingQuery` к значениям, возвращаемым от описываемого метода. Если активности не удастся справиться с возвращенным курсором либо не удастся закрыть курсор, то в приложении возникнет серьезная утечка памяти, что чревато низкой производительностью, а возможно — и аварийным завершением программы.
- `update(Uri uri, ContentValues values, String selection, String[] selectionArgs)` — вызывается всякий раз, когда клиенту требуется обновить одну строку или более в базе данных поставщика содержимого. Он преобразуется в предложение `UPDATE` на языке `SQL`.
- `delete(Uri uri, String selection, String[] selectionArgs)` — вызывается каждый раз, когда клиенту требуется удалить одну строку или более в базе данных поставщика содержимого. Он преобразуется в предложение `DELETE` на языке `SQL`.

Каждый из этих четырех методов совершает действия применительно к данным, на которые ссылается конкретный URI. Типичная реализация каждого из этих методов начинается с сопоставления аргумента входящего URI с определенным типом данных. Например, реализация поставщика содержимого должна определить, ссылается ли URI на конкретный видеоролик или на группу видеороликов. После того как поставщик сопоставляет URI с данными, выполняются соответствующие операции SQL. **Затем каждый метод возвращает значение. Это значение либо содержит данные, на которые указывает ссылка, либо описывает данные, затронутые операцией, либо ссылается на количество элементов, которые были затронуты в результате операции.** Например, на запрос конкретного видеоролика будет возвращен курсор, содержащий только один видеоэлемент, если заданный уникальный идентификатор ресурса ссылался на единственный элемент, который присутствует в локальной таблице.

Сопоставление URI с табличными данными — важнейшая часть работы, которую выполняет поставщик содержимого. Конечно, можно подумать, что не так уж сложно будет провести синтаксический разбор URI поставщика содержимого самостоятельно, но в Android предоставляется отличная утилита, способная сделать эту работу за вас. Она не только удобна, но и, что гораздо важнее, помогает разработчикам использовать тот формат URI поставщика содержимого, который мы рассмотрели выше, в качестве стандарта. Класс `URIMatcher` обеспечивает отображение URI, содержащих строки источника, пути и идентификатора, на определенные приложением константы, используемые с предложениями `case`, которые занимаются обработкой конкретных подтипов URI. На этом этапе поставщик может определить, какими операциями SQL он будет пользоваться для управления строками таблицы. Типичный поставщик содержимого создает статический экземпляр `URIMatcher` и заполняет его при помощи статического инициализатора. Этот инициализатор вызывает `URIMatcher.addURI` для организации сопоставления первого уровня (**first-level mapping**), которое позже используется в методах поставщика содержимого. О том, как это делается в нашем простом поставщике видеосодержимого, рассказано в подразделе «Класс `SimpleFinchVideoContentProvider` и переменные экземпляров» раздела «Полный код поставщика содержимого: поставщик `SimpleFinchVideoContentProvider`» далее.

Управление файлами и двоичные данные

Поставщикам содержимого часто приходится управлять большими фрагментами двоичных данных, например битовыми картами или музыкальным клипом. Необходимость хранения больших файлов с данными не может не отразиться на проектировании приложения и, скорее всего, серьезно повлияет на производительность программы. Поставщик содержимого может передавать файлы через URI. При этом в идентификаторе ресурса заключается информация о физическом местоположении нужных файлов, а сам клиент может этого и не узнать. Итак, клиенты используют уникальные идентификаторы ресурсов, содержащиеся в поставщиках содержимого, чтобы получать доступ к самим файлам, но не к информации о том, где именно эти файлы находятся. Такой уровень опосредованности позволяет поставщику содержимого управлять этими файлами наиболее целесообразным способом, не допуская утечки информации к клиенту. Если бы такая утечка происходила, она

могла бы даже приводить к изменениям кода в клиенте, если бы поставщику содержимого потребовалось изменить способ хранения физических файлов. В принципе, гораздо проще изменять только сам поставщик, а не все его потенциальные клиенты. Клиентам совершенно не нужно знать, что множество медиафайлов, которыми располагает поставщик содержимого, могут находиться во флеш-памяти, на карте памяти или вообще в сети, поскольку поставщик содержимого предоставляет файлы при помощи набора уникальных идентификаторов ресурсов, а клиент уже способен обработать эти идентификаторы. При обращении с каждым конкретным URI клиент просто будет использовать метод `ContentResolver.openInputStream`, а потом считывать данные из результирующего потока.

Кроме того, при совместном использовании больших объемов данных несколькими приложениями (поскольку приложение Android не должно считывать или перезаписывать файлы, созданные другим приложением) поставщик содержимого может использоваться для доступа к соответствующим (совместно используемым) байтам. Следовательно, когда поставщик содержимого возвращает указатель к файлу, этот указатель должен иметь вид `content:// URI`, а не имя файла в формате Unix. При использовании `content:// URI` файл можно открывать и считывать, только располагая правом доступа, полученным от поставщика содержимого, который владеет файлом, а не от клиентского приложения (клиентское приложение не должно иметь доступа к правам на файл).

Кроме того, необходимо учитывать, что система ввода-вывода, относящаяся к файловой системе, гораздо быстрее и многофункциональнее, чем система обработки объектов двоичной компоновки (BLOB) базы данных SQLite. Поэтому гораздо лучше использовать файловую систему Unix для непосредственного хранения двоичных данных. Да и какая польза от помещения двоичной информации в базу данных, если поиск по этим данным невозможен?

Для реализации вышеописанного подхода в приложении документация SDK Android предлагает одну стратегию, которая заключается в том, что поставщик содержимого долговременно сохраняет данные в файле, а также сохраняет в базе данных уникальный идентификатор ресурса, `content:// URI`. Этот идентификатор указывает на файл, как это показано на рис. 12.2. Клиентские приложения передают URI, содержащийся в этом поле, `ContentProvider.openStream` для получения потока байтов от указанного в URI файла.

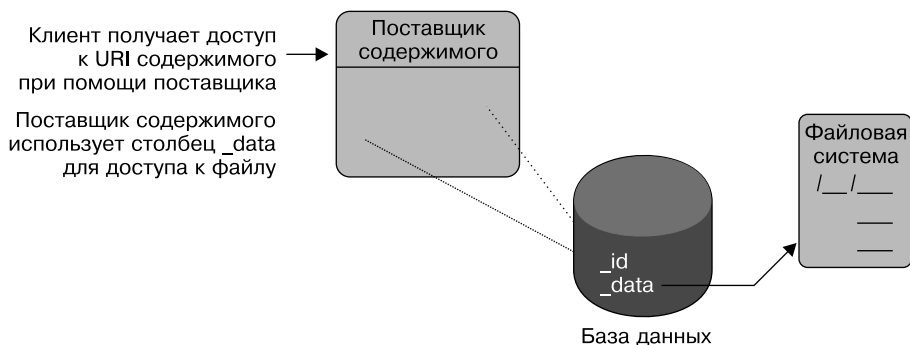


Рис. 12.2. Типичное использование курсоров и поставщиков содержимого в паттерне MVC, применяемом в Android

Говоря подробнее, чтобы осуществить такой файловый подход, в документации рекомендуется не создавать гипотетическую пользовательскую таблицу таким образом:

```
CREATE TABLE user ( _id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT,  
                    password TEXT, picture BLOB );
```

а сделать две таблицы, которые будут выглядеть вот так:

```
CREATE TABLE user ( _id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT,  
                    password TEXT, picture TEXT );  
CREATE TABLE userPicture ( _id INTEGER PRIMARY KEY AUTOINCREMENT,  
                             _data TEXT );
```

В столбце `picture` таблицы `user` будет сохраняться `content:// URI`, указывающий на строку в таблице `userPicture`. Столбец `_data` из таблицы `userPicture` будет указывать на реальный файл из файловой системы Android.

Если бы путь к файлу сохранялся непосредственно в таблице `user`, то клиенты получали бы путь, но не могли открыть расположенный по нему файл, так как файлом владеет приложение, передающее файлы поставщику содержимого, а клиенты не имеют права доступа к этим файлам. Но в решении, предложенном здесь, доступ контролируется классом `ContentResolver`, который будет рассмотрен ниже.

При обработке запросов класс `ContentResolver` ищет столбец с названием `_data`. Если удается найти файл, указанный в этом столбце, то метод `openOutputStream` поставщика содержимого открывает файл и вернет клиенту объект `java.io.OutputStream`. Именно этот объект был бы возвращен клиенту, если бы он сам мог открыть файл. Класс `ContentResolver` входит в состав того же приложения, что и поставщик содержимого. Поэтому он может открыть файл, тогда как клиент — не может.

Ниже в этой главе мы продемонстрируем поставщик содержимого, который использует свойственную для таких поставщиков функцию управления файлами, чтобы сохранять эскизы изображений.

Модель MVC в Android и наблюдение за содержимым

Очень важно обрисовать общую картину того, как паттерн MVC работает в Android с поставщиками содержимого. Мы поговорим о паттерне MVC в Android и более подробно в разделе «Сетевой вариант “Модель-вид-контроллер”» главы 13.

Чтобы оценить всю мощь фреймворка, использующего поставщики содержимого, необходимо знать о том, как события обновления курсора вызывают динамические обновления пользовательского интерфейса в Android. Нам кажется, что, затронув эту тему, мы сможем акцентировать внимание на часто упускаемых из виду путях обмена информацией, присутствующих в традиционном паттерне программирования «Модель-вид-контроллер». В целом работа этого паттерна строится так: вид (**view**) **принимает события пользовательского ввода и доносит эту информацию контроллеру (controller)**. Контроллер вносит изменения в модель (**model**), а модель посылает события обновления виду, а также любому другому наблюдателю, который регистрируется как заинтересованный в работе модели. Вид отображает содержимое модели — обычно это происходит без прямого привлечения

логики приложения — и в идеальном случае просто итерирует данные, относящиеся к модели.

В Android паттерн MVC работает так, как показано на рис. 12.3.

- Модель состоит из поставщика содержимого и курсоров, возвращаемых его методом `query`, а также из данных, которые содержатся в таблицах SQLite.
- Поставщики содержимого пишутся так, чтобы они могли посылать события уведомления при любом изменении данных. Это делается путем вызова `ContentResolver.notifyChange`. Поскольку только поставщик содержимого обладает доступом с возможностью изменения данных, ему будет известно обо всех изменениях данных.
- Уведомления направляются к компоненту пользовательского интерфейса, частую — `ListView`. При этом происходит наблюдение за объектами `Cursor`, связанными с уникальными идентификаторами ресурсов поставщика содержимого. Сообщения об обновлении курсора поступают от модели к виду, в ответ на вызов `notifyChange` внутри поставщика содержимого. Вид и контроллер соответствуют видам и активностям Android, а также классам, которые слушают генерируемые ими события. В частности, система доставляет сообщения `ContentObserver.onChange` экземплярам `ContentObserver`, зарегистрированным при помощи `Cursor.registerContentObserver`. Классы Android автоматически регистрируются на получение обновлений курсора, всякий раз, когда разработчик вызывает метод вроде `ListView.setAdapter(ListAdapter)`. В списковом виде имеется внутренний наблюдатель содержимого, а адаптер списка будет регистрироваться на получение обновлений от объекта `Cursor`.

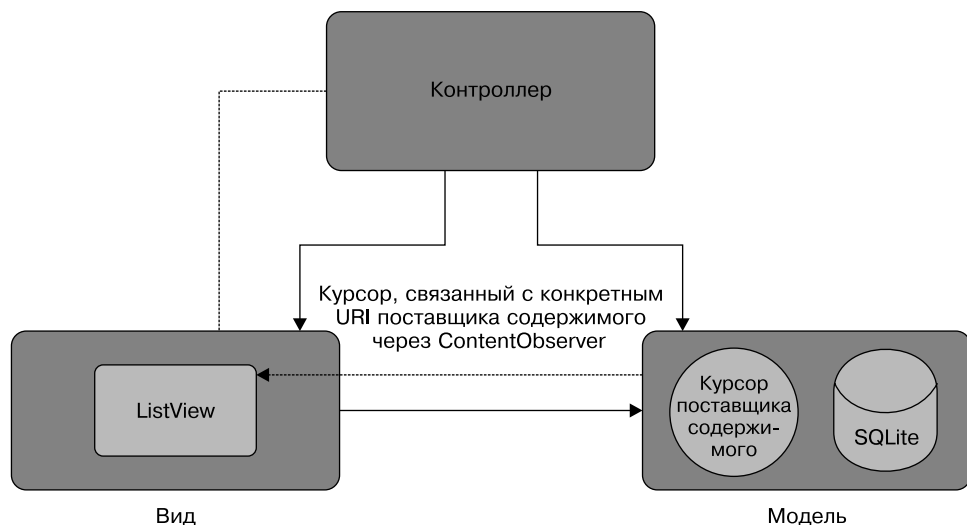


Рис. 12.3. Типичный пример использования курсоров и поставщиков содержимого при применении паттерна Android MVC

Чтобы представить себе, как эти уведомления работают на практике, предположим, что активность готовится вызвать `ContentResolver.delete`. Как вы вскоре

увидите, соответствующий поставщик содержимого сначала удалит строку из своей базы данных, а затем уведомит об этом URI преобразователя содержимого, который соответствует данной строке. Любые курсоры-слушатели, задействованные в любом виде, просто получают уведомления о том, что данные изменились. Виды, в свою очередь, получают событие обновления и перерисуют себя таким образом, чтобы на экране отображалось новое состояние. Виды отрисовывают любое состояние, которое относится к области, занимаемой ими на дисплее. Если в этой области находился элемент, который только что был удален, то он исчезнет из пользовательского интерфейса. Объекты `Cursor` действуют в качестве посредников между потребителями информации курсора и системой поставщика содержимого. События идут от поставщика содержимого через курсор в систему видов. Высокая степень автоматизации этой цепи сильно упрощает жизнь разработчику — ему необходимо выполнить минимальный объем работы, чтобы все заработало. Кроме того, программам не придется явно опрашивать систему, чтобы отображение модели в них оставалось актуальным, так как модель сама уведомляет вид о том, что ее состояние изменилось.

Полный код поставщика содержимого: поставщик `SimpleFinchVideoContentProvider`

Теперь, когда мы поговорили о важных задачах, связанных с написанием поставщика содержимого с применением паттерна «Модель-вид-контроллер» в Android (то есть о системе обмена информацией, используемой поставщиками содержимого в Android), рассмотрим, как написать собственный поставщик содержимого. Класс `SimpleFinchVideoContentProvider` наследуется `ContentProvider` так:

```
public class SimpleFinchVideoContentProvider extends ContentProvider {
```

Класс `SimpleFinchVideoContentProvider` и переменные экземпляров

Как обычно, лучше сначала разобраться с основными переменными классов, используемыми методом, а потом изучать, как этот метод работает. Для работы с `SimpleFinchVideoContentProvider` нам понадобится освоить следующие члены класса:

```
private static final String DATABASE_NAME = "simple_video.db";  
private static final int DATABASE_VERSION = 2;  
private static final String VIDEO_TABLE_NAME = "video";  
private DatabaseHelper mOpenHelper;
```

- `DATABASE_NAME` — имя файла базы данных, расположенной на устройстве. В случае с простым видеоприложением Finch путь к файлу базы данных будет таким: `/data/data/com.oreilly.demo.pa.finchvideo/databases/simple_video.db`.
- `DATABASE_VERSION` — версия базы данных, соответствующая этому коду. Если номер выше, чем версия самой базы данных, то приложение вызывает метод `DatabaseHelper.onUpdate`.
- `VIDEO_TABLE_NAME` — имя таблицы видео, находящейся внутри базы данных `simple_video`.

- `mOpenHelper` — переменная вспомогательной утилиты базы данных. Такая переменная инициализируется во время работы метода `onCreate`. Этот помощник обеспечивает доступ к базе данных для методов `insert`, `query`, `update` и `delete`.
- `sUriMatcher` — статический блок инициализации, в котором осуществляется инициализация статических переменных, если эти переменные не удастся инициализировать в одной строке. Например, работа нашего простого поставщика видео начинается с добавления соответствий содержимого поставщика и URI в `UriMatcher`, вот так:

```
private static UriMatcher sUriMatcher;

private static final int VIDEOS = 1;
private static final int VIDEO_ID = 2;

static {
    sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    sUriMatcher.addURI(AUTHORITY, FinchVideo.SimpleVideos.VIDEO_NAME,
        VIDEOS);
    // использование хеш-символа (#) указывает на сопоставление с id
    sUriMatcher.addURI(AUTHORITY,
        FinchVideo.SimpleVideos.VIDEO_NAME + "/" + "#", VIDEO_ID);
    ...
    // дальнейшая инициализация
```

Класс `UriMatcher` является базовой утилитой, используемой в **Android** для отображения URI поставщиков содержимого. Для применения экземпляра этого класса его нужно заполнить отображениями строки URI, например `videos`, на константу. Наше отображение действует следующим образом. Сначала приложение передает аргумент, `UriMatcher.NO_MATCH`, конструктору класса `UriMatcher`, относящегося к поставщику содержимого, чтобы определить значение, соответствующее ситуации, в которой заданный URI не совпадает ни с одним имеющимся URI. Затем приложение добавляет отображения для множества видео к константе `VIDEOS`, а потом — отображение конкретного видеоролика к константе `VIDEO_ID`. При отображении всех **URI поставщика содержимого на целочисленное значение поставщик содержимого может применить оператор `switch` для перехода к коду, используемому для обработки нескольких видеофайлов или одного видеофайла.**

В результате этого процесса такой URI, как `content://com.oreilly.demo.pa.finchvideo.SimpleFinchVideo/video`, ассоциируется с константой `VIDEOS`, соответствующей всем видеофайлам. URI отдельной видеозаписи, например `content://oreilly.demo.pa.finchvideo.SimpleFinchVideo/video/7`, ассоциируется с константой `VIDEO_ID`, соответствующей одному видеофайлу. Хеш-символ (`#`) в конце обнаружителен совпадений URI — это джокерный символ для любого URI, оканчивающегося на целое число.

- `sVideosProjectionMap` — проекционный контейнер, используемый в методе запроса. Этот `HashMap` ассоциирует имена столбцов поставщика содержимого с именами столбцов базы данных. Пользоваться проекционным контейнером не обязательно, но если он применяется, то в нем должны содержаться все имена

столбцов, которые могут быть возвращены в запросе. В случае с SimpleFinchVideoContentProvider имена столбцов поставщика содержимого и базы данных идентичны, поэтому sVideosProjectionMap не требуется. Но мы покажем примеры приложений, где такой контейнер нужен. В следующем коде создается пример проекционного ассоциирования:

```
// образец проекционного контейнера, в рассматриваемом здесь приложении
// он не используется
sVideosProjectionMap = new HashMap<String, String>();
sVideosProjectionMap.put(FinchVideo.Videos._ID,
    FinchVideo.Videos._ID);
sVideosProjectionMap.put(FinchVideo.Videos.TITLE,
    FinchVideo.Videos.TITLE);
sVideosProjectionMap.put(FinchVideo.Videos.VIDEO,
    FinchVideo.Videos.VIDEO);
sVideosProjectionMap.put(FinchVideo.Videos.DESCRPTION,
    FinchVideo.Videos.DESCRPTION);
```

Реализация метода onCreate

При инициализации простого поставщика видео Finch хранилище данных SQLite организуется следующим образом:

```
private static class DatabaseHelper extends SQLiteOpenHelper {
    public void onCreate(SQLiteDatabase sqLiteDatabase) {
        createTable(sqLiteDatabase);
    }

    // метод create table также может вызываться из onUpgrade
    private void createTable(SQLiteDatabase sqLiteDatabase) {
        String qs = "CREATE TABLE " + VIDEO_TABLE_NAME + " (" +
            FinchVideo.SimpleVideos._ID + " INTEGER PRIMARY KEY, " +
            FinchVideo.SimpleVideos.TITLE_NAME + " TEXT, " +
            FinchVideo.SimpleVideos.DESCRPTION_NAME + " TEXT, " +
            FinchVideo.SimpleVideos.URI_NAME + " TEXT)";
        sqLiteDatabase.execSQL(qs);
    }
}
```

При создании таблиц SQLite, предназначенных для поддержки работы поставщика содержимого, программист обязан делать в таблице поле с основным ключом, которое называется `_id`. Хотя необходимость создания этого поля совсем неочевидна, если вы подробно не читали документацию по разработке в Android, система управления содержимым, действующая в Android, действительно требует присутствия поля `_id` в курсоре, который возвращается методом запроса. Столбец `_id` используется в запросах и сопоставляется со специальным символом `#`, присутствующим в URL поставщиков содержимого. Например, URL типа `content://contacts/people/25` будет ассоциироваться со строкой данных в таблице `contacts`, имеющей номер `_id` 25. На самом деле это требование сводится всего лишь к использованию специального названия для основного ключа таблицы.

Реализация метода getType

Далее мы реализуем метод `getType`, который предназначен для определения типов MIME произвольных URI, получаемых от клиента. Как видно из следующего кода, мы сопоставляем константы `VIDEOS` и `VIDEO_ID` с типами MIME, которые мы определили в нашем общедоступном API:

```
public String getType(Uri uri) {
    switch (sUriMatcher.match(uri)) {
        case VIDEOS:
            return FinchVideo.SimpleVideos.CONTENT_TYPE;

        case VIDEO_ID:
            return FinchVideo.SimpleVideos.CONTENT_VIDEO_TYPE;

        default:
            throw new IllegalArgumentException("Unknown video type: " +
                                           uri);
    }
}
```

Реализация API для поставщика содержимого

Реализация поставщика содержимого должна переопределять методы работы с данными (`insert`, `query`, `update` и `delete`) базового класса `ContentProvider`. В простом видеоприложении эти методы определяются в классе `SimpleFinchVideoContentProvider`.

Метод query

После сопоставления входящего URI с контентом метод `query` нашего поставщика содержимого выполняет соответствующую выборку из базы данных, доступной для чтения. Эта задача делегируется `SQLiteDatabase.query`, после чего результаты возвращаются в виде объекта `Cursor` (курсор базы данных). Курсор содержит все строки базы данных, подходящие под описание аргументом (URI). После того как мы сделаем запрос, действующий в Android механизм поставки содержимого автоматически будет поддерживать использование экземпляров курсора с многочисленными экземплярами процессов. Таким образом, метод `query` нашего поставщика содержимого может просто возвращать курсор как обычное возвращаемое значение, предоставляя его клиентам, которые могут работать с другим процессом.

Кроме того, метод `query` поддерживает параметры `uri`, `projection`, `selection`, `selectionArgs` и `sortOrder`, применяемые так же, как и аргументы метода `SQLiteDatabase.query`, рассмотренного в главе 9. Как и в случае с любым предложением `SELECT` на языке SQL, параметры метода `query` позволяют клиентам нашего поставщика содержимого выбирать только конкретные видеоролики, соответствующие параметрам запроса. Клиент, вызывающий простой поставщик видеoinформации, может не только передать уникальный идентификатор ресурса, но и сообщить дополнительное условие `where` с аргументами `where`. Например, такие аргументы позволяют разработчику запрашивать все видеоролики от определенного автора.



Итак, паттерн «Модель-вид-контроллер» в Android основан на передаче курсоров и данных, которые в них содержатся, а также на встроенной во фреймворк системе доставки сообщений об обновлении наблюдателя контента. Поскольку клиенты, работающие в различных процессах, совместно используют объекты `Cursor`, реализация поставщика содержимого должна следить за тем, чтобы не закрывать курсор, который получен в методе `query`. Если в таком случае курсор будет закрыт, клиенты не увидят выдаваемых исключений. А курсор все время будет вести себя как пустой объект и не будет больше получать событий обновления — поскольку задача правильного управления возвращенными курсорами должна решаться на уровне активности.

Когда запрос к базе данных выполнен, наш поставщик вызывает метод `Cursor.setNotificationUri`, чтобы задать уникальный идентификатор ресурса. По этому идентификатору инфраструктура поставщика содержимого будет определять, какие события обновления поставщика будут поступать в новоиспеченный курсор. Данный уникальный идентификатор ресурса становится точкой взаимодействия между клиентами, наблюдающими за данными, на которые ссылается этот идентификатор, с одной стороны, и поставщиком содержимого, уведомляющего по этому идентификатору, — с другой. Вызов этого простого метода обеспечивает курсирование сообщений об обновлении поставщика содержимого; этот процесс мы рассмотрели в разделе «Модель MVC в Android и наблюдение за содержимым» данной главы.

Ниже приводится код метода `query` нашего простого поставщика видеoinформации. Этот код выполняет сопоставление URI, запрашивает базу данных, а затем возвращает курсор:

```
@Override
public Cursor query(Uri uri, String[] projection, String where,
                    String[] whereArgs, String sortOrder)
{
    // Если порядок сортировки не указан — использовать значение.
    // заданное по умолчанию.
    String orderBy;
    if (TextUtils.isEmpty(sortOrder)) {
        orderBy = FinchVideo.SimpleVideos.DEFAULT_SORT_ORDER;
    } else {
        orderBy = sortOrder;
    }

    int match = sUriMatcher.match(uri);  ❶

    Cursor c;

    switch (match) {
        case VIDEOS:
            // запросить у базы данных все видеофайлы
            c = mDb.query(VIDEO_TABLE_NAME, projection,
                          where, whereArgs,
                          null, null, sortOrder);

            c.setNotificationUri(
```

```

        getContext().getContentResolver(),
        FinchVideo.SimpleVideos.CONTENT_URI); ❷
    break;
case VIDEO_ID:
    // запросить у базы данных конкретный видеофайл
    long videoID = ContentUris.parseId(uri);
    c = mDb.query(VIDEO_TABLE_NAME, projection,
        FinchVideo.Videos._ID + " = " + videoID +
            (!TextUtils.isEmpty(where) ?
                " AND (" + where + ")" : ""),
        whereArgs, null, null, sortOrder);
    c.setNotificationUri(
        getContext().getContentResolver(),
        FinchVideo.SimpleVideos.CONTENT_URI);
    break;
default:
    throw new IllegalArgumentException("unsupported uri: " + uri);
}

return c; ❸
}

```

Пояснения к коду следующие.

- ❶ Здесь происходит сопоставление URI с константами, применяется специальный встроенный механизм (URI matcher).
- ❷ Установка `FinchVideo.SimpleVideos.CONTENT_URI` в качестве URI приводит к тому, что курсор получает события уведомления, связанные с преобразованием любых данных, на которые указывает этот URI. В данном случае курсор будет получать все события, относящиеся ко всем видео, поскольку именно на все видеофайлы ссылается идентификатор `FinchVideo.SimpleVideos.CONTENT_URI`.
- ❸ Непосредственное возвращение курсора. Как было указано выше, действующая в Android система поставщиков содержимого поддерживает совместное использование несколькими процессами любых данных, содержащихся в курсоре. Межпроцессное совместное использование данных обеспечивается «просто так», как неотъемлемое свойство поставщиков содержимого. Можно просто вернуть курсор, и он станет доступен активностям, действующим в различных процессах.

Метод insert

Теперь поговорим о методе, который получает значения от клиента, проводит их валидацию, а затем добавляет новую строку в базу данных, содержащую эти значения. Значения передаются классу `ContentProvider`, находящемуся в объекте `ContentValues`:

```

@Override
public Uri insert(Uri uri, ContentValues initialValues) {
    // валидация запрошенного uri

```

```

if (sUriMatcher.match(uri) != VIDEOS) {
    throw new IllegalArgumentException("Unknown URI " + uri);
}

ContentValues values;
if (initialValues != null) {
    values = new ContentValues(initialValues);
} else {
    values = new ContentValues();
}

verifyValues(values);

// вставка initialValues в новую строку базы данных
SQLiteDatabase db = mOpenHelper.getWritableDatabase();
long rowId = db.insert(VIDEO_TABLE_NAME,
    FinchVideo.SimpleVideos.VIDEO_NAME, values);
if (rowId > 0) {
    Uri videoUri =
        ContentUris.withAppendedId(
            FinchVideo.SimpleVideos.CONTENT_URI, rowId);
    getContext().getContentResolver().
        notifyChange(videoUri, null);
    return videoUri;
}

throw new SQLException("Failed to insert row into " + uri);
}

```

Кроме того, метод `insert` будет заниматься сопоставлением входящих URI с контентом, выполнять соответствующие операции вставки информации в базу данных, а затем возвращать уникальный идентификатор ресурса, который будет ссылаться на новую строку базы данных. Поскольку метод `SQLiteDatabase.insert` возвращает ID новой строки, созданной в базе данных, то есть значение столбца `_id` для данной строки, поставщик содержимого легко собирает нужный URI, прикрепляя значение переменной `rowId` к URI поставщика содержимого, который определяется в общедоступном API этого поставщика. Упомянутый API рассматривается в главе 3.

Пояснения к коду следующие.

- ❶ Мы используем утилиты Android для управления уникальными идентификаторами ресурсов поставщика содержимого. В частности, метод `ContentUris.withAppendedId` применяется для прикрепления переменной `rowId` в качестве номера строки возвращенного URI информации, вставленной в базу данных.
- ❷ Здесь поставщик содержимого выполняет уведомление по URI, что приведет к инициации события об обновлении содержимого и доставки его слушателям курсора. Обратите внимание на то, что единственная причина, по которой событие направляется к наблюдателям контента, заключается в том, что уведомление иницируется поставщиком содержимого.

Метод update

Метод update работает так же, как insert, но применяет операцию update к соответствующей базе данных, изменяя, таким образом, строки базы данных, на которые ссылается URI. Метод update возвращает количество строк, затронутых операциями обновления:

```
@Override
public int update(Uri uri, ContentValues values, String where,
                  String[] whereArgs)
{
    // Вызов для уведомления uri после того, как произойдет удаление,
    // является явным.
    getContext().getContentResolver().notifyChange(uri, null);

    SQLiteDatabase db = mOpenHelper.getWritableDatabase();
    int affected;
    switch (sUriMatcher.match(uri)) {
        case VIDEOS:
            affected = db.update(VIDEO_TABLE_NAME, values,
                                where, whereArgs);
            break;
        case VIDEO_ID:
            String videoId = uri.getPathSegments().get(1);
            affected = db.update(VIDEO_TABLE_NAME, values,
                                FinchVideo.SimpleVideos._ID + "=" + videoId
                                + (!TextUtils.isEmpty(where) ?
                                   " AND (" + where + ')' : ""),
                                whereArgs);
            break;

        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }

    getContext().getContentResolver().notifyChange(uri, null);
    return affected;
}
```

Метод delete

Метод delete похож на update, но он удаляет строки, на которые ссылается заданный URI. Подобно update, метод delete возвращает количество строк, затронутых операцией удаления:

```
@Override
public int delete(Uri uri, String where, String[] whereArgs) {
    int match = sUriMatcher.match(uri);
    int affected;

    switch (match) {
        case VIDEOS:
```

```

        affected = mDb.delete(VIDEO_TABLE_NAME,
            (!TextUtils.isEmpty(where) ?
                " AND (" + where + ')' : ""),
            whereArgs);

        break;
    case VIDEO_ID:
        long videoId = ContentUris.parseId(uri);
        affected = mDb.delete(VIDEO_TABLE_NAME,
            FinchVideo.SimpleVideos._ID + "=" + videoId
                + (!TextUtils.isEmpty(where) ?
                    " AND (" + where + ')' : ""),
            whereArgs);

        // вызов для уведомления uri после того, как произойдет удаление
        getContext().getContentResolver().
            notifyChange(uri, null);

        break;
    default:
        throw new IllegalArgumentException("unknown video element: " +
            uri);
}

return affected;
}

```

Необходимо обратить внимание на то, что приведенные выше объяснения относятся только к нашей простой реализации поставщика содержимого. В более сложных сценариях может потребоваться объединение данных из нескольких таблиц при формировании запроса либо каскадное удаление заданного фрагмента данных. Поставщик содержимого может выбирать собственную схему управления данными, пользуясь **API базы данных SQLite в Android**, если эта схема не конфликтует с работой API клиента поставщика содержимого.

Определение того, как часто следует уведомлять наблюдатели содержимого

Как видно из листинга, описывающего операции управления данными в поставщике содержимого, уведомление не происходит в системе управления содержимым Android «просто так». В частности, вставка информации в таблицу SQLite не вызывает автоматической установки триггера базы данных, который инициировал бы обновления с подачи поставщика содержимого. От разработчика поставщика содержимого зависит, будет ли реализована схема, определяющая подходящее время для отправки уведомлений и решающая, какой URI посылать при изменении данных поставщика содержимого. Как правило, поставщики содержимого в Android немедленно после события отправляют уведомления всем URI, которые изменились в ходе конкретной операции с данными.

При выборе схемы уведомлений разработчику следует рассмотреть возможность такого компромисса: отказаться от доскональных результатов уведомления, но более точно выстроить обновления после изменений. Так можно снизить нагрузку на систему пользовательского интерфейса. Если списку сообщается, что в нем изменился только один элемент, то можно перерисовать этот элемент лишь в случае, когда он находится в поле видимости. Но доскональные уведомления имеют и еще один недостаток: при их применении через систему проходит большее количество событий. Вероятно, пользовательский интерфейс будет перерисовываться чаще, поскольку он будет получать больше отдельных событий об уведомлении. При более общем подходе к уведомлениям через систему проходит меньше событий. Но если событий меньше, то это зачастую означает, что при получении уведомления об изменениях приходится перерисовывать более значительную часть пользовательского интерфейса. Например, список может получить единственное событие, требующее от него обновить все элементы, в то время как фактически изменились всего три элемента. Советуем учитывать возможность такого компромисса при подборе схемы обновлений. Например, можно дожидаться окончания считывания большого количества событий, а потом инициировать единственное событие типа «все изменилось», а не посылать обновления после каждого отдельного события.

Зачастую поставщики содержимого просто уведомляют об изменении данных все клиенты, URI которых указывают на изменившуюся информацию.

Объявление вашего поставщика содержимого

В пункте «Использование поставщика содержимого» подраздела «Поставщики содержимого» раздела «Другие компоненты Android» главы 3 было показано, как клиенты получают доступ к поставщику содержимого, чтобы работать с ним. Теперь у нас есть собственный поставщик содержимого, осталось только открыть клиентам доступ к нему. Для этого нужно добавить в файл `AndroidManifest.xml` следующую XML-строку:

```
<provider android:name=".provider.SimpleFinchVideoContentProvider"
    android:authorities="oreilly.demo.pa.finchvideo.SimpleFinchVideo"/>
```

Когда ваше приложение будет собрано, в его файле APK окажутся классы для реализации поставщика содержимого, а в файле описания будет находиться строка XML, напоминающая ту, которую мы показали выше. После этого доступ к поставщику содержимого будет открыт для кода всех приложений с платформы Android. Предполагается, что код запросил и получил права на такой доступ. Эти операции были описаны в главе 3.

Итак, в этой главе мы справились с задачей создания собственного простого поставщика содержимого. В следующей главе рассмотрим некоторые новаторские паттерны, связанные с поставщиками содержимого.

13 Поставщики содержимого как фасад для веб-сервисов RESTful

В главе 6 мы говорили о том, что при работе с пользовательскими интерфейсами, которым необходимо взаимодействовать с удаленными службами, возникают нетривиальные проблемы — например, необходимость не занимать поток пользовательского интерфейса решением долговременных задач. Кроме того, в главе 3 мы отмечали, что API поставщика содержимого в Android обладает симметрией, схожей с симметрией веб-служб типа REST (с передачей состояния представления). Операции с данными, совершаемые в поставщике содержимого, соответствуют операциям с данными в REST-службах, и ниже будет показано, как преобразовать уникальные идентификаторы ресурсов из поставщика содержимого в такую форму, которая позволяет запрашивать данные из сети. Советуем пользоваться преимуществами, свойственными для такой симметрии, при написании поставщиков содержимого. Поставщик содержимого должен создаваться как асинхронный буфер между доменными (уникальными) аспектами вашего приложения и сетевыми запросами, получающими данные. Обработкой этих данных занимается уже ваше приложение. Если писать приложение по такому принципу, оно значительно упростится и поможет избежать распространенных ошибок, связанных с разработкой пользовательских интерфейсов и работой в сети, типичных для программирования в Android и вообще на языке Java.

Исторически разработчики пользовательских интерфейсов на языке Java как для корпоративных, так и для мобильных приложений писали программы для мобильной среды и для настольных персональных компьютеров так, что эти приложения получались довольно «хрупкими». Иногда сетевые запросы выполнялись прямо в потоке пользовательского интерфейса, часто данные, полученные в результате этих запросов, не кэшировались. В большинстве приложений для отображения в пользовательском интерфейсе чего бы то ни было нужно было обращаться к сети — всякий раз, когда пользователь требовал вывести на экран новую информацию. Трудно поверить, что рабочие станции Unix образца 1980-х и 1990-х годов зачастую блокировались, когда оказывался закрыт доступ к удаленно смонтированным файловым системам. Если приложения использовали схему динамического кэши-

рования на локальной машине, то они могли продолжать работать и в то время, пока файловый сервер оставался недоступен, а потом вновь синхронизироваться с ним, когда доступ возобновлялся. Разработчики должны были специально обращать внимание (что делалось редко) на то, чтобы приложения правильно получали доступ к сетевым данным и правильно их сохраняли.

Данная тенденция нашла поддержку и в J2ME, где разработчики могли сохранять состояние сети в довольно слабой системе управления записями, которая называлась RMS. Эта библиотека не поддерживала ни язык запросов, ни систему уведомлений, используемую с паттерном «Модель-вид-контроллер». Разработчики, имевшие дело с J2ME, были вынуждены порождать собственные обычные потоки Java, в которых потом выполнялись сетевые запросы, но зачастую этим пренебрегали, и приложение получалось ненадежным. Если веб-браузерам приходилось загружать сетевые данные в поток пользовательского интерфейса, то часто возникали ситуации их полного зависания, вплоть до того, что операционной системе приходилось завершить процесс браузера, чтобы выйти из такого положения. При каждом подвисании сети пользовательский интерфейс блокировался. Страницы, а также все изображения, на которые они ссылались, всегда требовалось загружать заново при каждом просмотре, из-за этого работа пользователя с программой шла очень медленно, если какой-нибудь запрос не приводил вдобавок к зависанию всего приложения. Сухой остаток из всех этих историй заключается в том, что, как правило, операционные системы возлагали задачи загрузки и кэширования информации на само приложение, практически не предоставляя прямой библиотечной поддержки, которая помогла бы разработчикам правильно реализовывать эти функции.

Для решения всех этих проблем можно написать полностью асинхронный интерфейс, который будет управлять и взаимодействием с сетью, и хранением данных. При таком подходе разработчику не приходится думать, правильно ли он поступил, когда запросил данные из сети. При применении такого API получение данных будет безопасным, независимо от того, работает ли в данный момент поток пользовательского интерфейса. Соображения такого рода становятся особенно важны при программировании для мобильных устройств, поскольку ненадежная связь с сетью повышает вероятность зависания кода, если он написан неправильно.

Рекомендуем использовать API поставщика содержимого как асинхронную модель сети и как кэш для состояний сети. Таким образом, виду и контроллеру вашего приложения не потребуются собственные механизмы для открытия соединений или доступа к базе данных. API поставщика содержимого несложно ассоциировать с API имеющихся веб-служб на основе REST. Поставщик содержимого просто находится между приложением и сетью, переадресовывая в сеть запросы и кэшируя результаты, если это требуется. В этой главе мы покажем, как такой подход помогает упростить приложение, а также опишем общие преимущества данного подхода. В частности, поговорим о том, как он позволяет внедрять еще более интересные свойства веб-программирования и технологии Ajax в приложениях Android. Подробнее о программировании с применением Ajax можно почитать по адресу <http://ru.wikipedia.org/wiki/AJAX>.

Разработка приложений Android с передачей состояния представления (RESTful)

Не только мы считаем данный подход перспективным. На конференции Google I/O, состоявшейся в мае 2010 года, сотрудник Google Вирджил Добжански прочитал доклад, в рамках которого обрисовал следующие три принципа использования поставщиков содержимого в тех случаях, когда веб-службы с передачей состояния представления требуется интегрировать в приложения Android.

- Activity ► Service ► ContentProvider. Этот вариант связан с использованием активности, содержащей службу для доступа к данным приложения. В свою очередь, служба обращается к поставщику содержимого для доступа к этой информации. В данном сценарии активность вызывает асинхронный метод службы, которая осуществляет асинхронные REST-активации.
- Activity ► ContentProvider ► Service. Активность связывается с поставщиком содержимого, который, в свою очередь, делегирует службе задачу асинхронной загрузки данных. При таком подходе активность может пользоваться API поставщика содержимого для взаимодействия с данными. Поставщик содержимого активизирует методы, применяемые к реализации асинхронной службы, чтобы активизировать REST-запросы. При данном подходе основной упор делается на использование той удобной симметрии, которая существует между API поставщика содержимого и применением протокола HTTP с передачей состояния представления.
- Activity ► ContentProvider ► SyncAdapter. Адаптеры синхронизации в Android обеспечивают каркас для синхронизации пользовательских данных между устройством и облаком. Например, в Google Contacts используется адаптер синхронизации. При данном сценарии активность использует API поставщика содержимого для доступа к данным, синхронизированным при помощи такого адаптера.

В этой главе мы подробно рассмотрим второй подход, для этого мы напишем второй вариант нашей программы для работы с Finch-видео. Данная стратегия позволит вам воспользоваться несколькими важными преимуществами, которые очень пригодятся вам при написании собственных приложений. Отдавая должное той элегантности, с которой при этом подходе интегрируется работа с сетью и паттерн Android «Модель-вид-контроллер», мы назвали следующий раздел «Сетевой вариант «Модель-вид-контроллер»».

После прочтения этой главы советуем вам посмотреть доклад с конференции Google: <http://www.google.com/events/io/2010/sessions/developing-RESTful-android-apps.html>.

Сетевой вариант «Модель-вид-контроллер»

Нам кажется, что удобно представить второй из описанных выше принципов как сетевой вариант паттерна «Модель-вид-контроллер», где сам поставщик содержимого получает данные из сети, а затем закачивает их в обычный паттерн MVC,

действующий в **Android**. Мы рассмотрим поставщик содержимого как модель сетевого состояния — поставщик может выполнять запросы на получение данных с состоянием, имеющимся в локальной системе, либо получать данные из сети. При применении такого подхода код контроллера и вида не должен непосредственно создавать сетевые запросы для доступа к данным приложения и для управления ими. Вместо этого вид и контроллер вашего приложения должны использовать `API ContentResolver` для того, чтобы запрашивать данные через поставщик содержимого. И только поставщик содержимого должен в асинхронном режиме загружать сетевые ресурсы и сохранять результаты в локальном кэше данных. Кроме того, поставщик содержимого всегда должен быстро реагировать на запрос, с самого начала избегая этапа сетевой активации, которая может потребоваться для выполнения запроса с использованием какой-либо информации, уже находящейся в локальной базе данных. При выполнении запроса по такому принципу гарантируется, что поток пользовательского интерфейса будет заблокирован не дольше, чем это необходимо, и что пользовательский интерфейс должен отобразить те или иные данные как можно быстрее. Таким образом, увеличивается скорость реагирования приложения и пользователю становится гораздо приятнее работать с таким интерфейсом. Рассмотрим, в какой последовательности поставщик содержимого запрашивает данные.

1. Поставщик содержимого выполняет сопоставление входящего **URI** и запрашивает содержимое локальной базы данных на предмет элементов, которые до этого совпали с данными запроса.
2. Наш поставщик содержимого всегда пытается получить наиболее актуальное состояние для запроса и после этого порождает асинхронный **REST**-запрос для загрузки содержимого из сети. Это поведение можно сделать конфигурируемым на базе запроса.
3. Поставщик содержимого возвращает клиенту курсор из исходного локального запроса.
4. Асинхронный загрузочный поток должен решить, необходимо ли обновить данные, находящиеся в кэше поставщика содержимого. Если такая необходимость есть, то поставщик содержимого загружает данные из сети и производит их синтаксический разбор.
5. Когда содержимое приходит из сети, поставщик содержимого непосредственно вставляет каждый новый фрагмент данных в базу данных, а потом уведомляет клиентов **URI** о приходе новой информации. Поскольку сама операция вставки уже происходит в рамках поставщика содержимого, не требуется вызывать `ContentResolver.insert`. Клиенты, содержащие существующие курсоры (данные в которых устарели), могут вызвать `Cursor.requery` для обновления своих данных.

В результате такой последовательности происходит обновление и вида и контроллера, получающих новые данные из сети, но только поставщик содержимого создает сетевой запрос. Запрос на получение ресурсов, которые в настоящий момент отсутствуют в поставщике содержимого, мы расцениваем как запрос на получение этих ресурсов из сети. В таком случае сетевой запрос, загружающий данные в кэш, можно считать побочным продуктом запроса активности к поставщику.

На рис. 13.1 проиллюстрированы операции, происходящие в поставщике содержимого при выполнении этой последовательности.

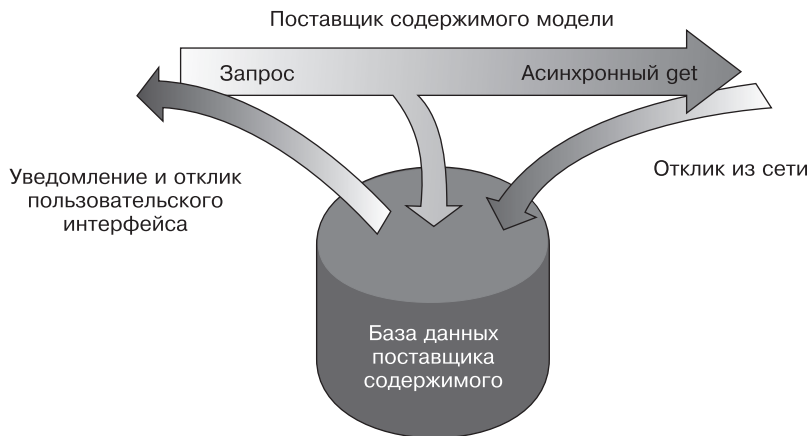


Рис. 13.1. Сетевой поставщик содержимого, кэширующий контент по инициативе клиента

С каждым запросом в этой последовательности используется единственный объект `Cursor`, создаваемый поставщиком содержимого, а затем возвращаемый в виду. Лишь от поставщика содержимого требуется уведомлять пользовательский интерфейс при изменении данных. Вид и контроллер не обязаны собирать данные и, конечно, не должны обновлять модель. Когда данные доступны, поставщик содержимого уведомляет курсор о том, что можно делать запрос. Роль управления данными целиком заключена в поставщике содержимого, благодаря этому упрощается код вида и контроллера. Клиент поставщика содержимого запрашивает данные и быстро получает курсор; когда прибывают данные из сети, курсор об этом уведомляется. Необходимо еще раз отметить, что работа механизма оповещения требует, чтобы объект `Cursor` и база были открытыми все время, пока клиент использует их. Закрытые курсоры и базы данных приводят к тому, что в клиентских видах не отображаются никаких результатов. В такой ситуации сложно понять, почему пуст компонент (например, список) — потому, что курсор был по ошибке закрыт, или потому, что запрос действительно не дал результатов.

Общая характеристика достоинств

Теперь в обобщенном виде представим достоинства подхода с применением сетевого варианта «Модель-вид-контроллер».

- Улучшается общая ощутимая работоспособность приложения, а также фактическая производительность, что объясняется применением кэширования. Это одни из наиболее очевидных достоинств данного паттерна. Часто мобильные программы работают так, как работал бы Веб без системы кэширования.

- Хранение данных в памяти — не лучший вариант, поскольку вы не знаете, когда Android удалит вашу активность из памяти. В рассматриваемом подходе особый акцент делается на максимально оперативном сохранении информации в поставщике содержимого.
- Исключено возникновение многих потенциальных проблем с безопасностью, свойственных для пользовательского интерфейса. Компоненты-виды в Android изначально создавались с расчетом на динамическое обновление и отражение информации, которая в настоящий момент содержится в курсоре. Другие системы компонентов, которые могут быть знакомы, например, читателям, работавшим с J2ME Swing, оставляют решение этой задачи разработчику. В такой ситуации нельзя исключать возможность того, что списковый компонент, последовательно удаляющий элементы данных, может при этом выйти за пределы своей модели.
- Описываемый подход позволяет максимально результативно использовать систему управления курсором, а также встроенные в пользовательский интерфейс возможности динамического обновления в ответ на события, связанные с наблюдением за содержимым. Разработчикам пользовательских интерфейсов не требуется писать собственные системы опроса (поллинга) и обновления; можно просто положиться на наблюдение за содержимым и на работу интерфейса поставщика содержимого.
- Если запросы к сети строятся правильно, то в данном случае поток пользовательского интерфейса не может подолгу занимать канал связи с сетью.
- Для доставки сетевых событий в программе не обязательно должен присутствовать пользовательский интерфейс. Даже если при прибытии из сети определенного события отсутствует необходимая активность, поставщик содержимого все равно сможет его обработать. Когда пользователь загружает активность, запрос обнаружит событие, прибывшее в фоновом режиме. При отсутствии действующей активности пользовательского интерфейса поступающие события не будут просто сбрасываться системой.
- Элементы приложения инкапсулируются и специализируются, поскольку, как мы уже упоминали, поставщик содержимого управляет всеми взаимодействиями, связанными с базой данных SQLite и с сетью. Вид и контроллер просто используют поставщик содержимого как универсальную систему для управления данными.
- Приложения писать проще именно потому, что достаточно сложно использовать API неправильно. Нужно просто выполнять вызовы поставщика содержимого, а система будет обрабатывать функциональность REST.
- Наконец, в книге по мобильному программированию легко сосредоточиться на проблемах, связанных с устройствами (то есть с аппаратным обеспечением), но если клиенты работают, преимущественно опираясь на свой кэш и обращаясь в сеть только при абсолютной необходимости, то вам удастся значительно снизить нагрузку на систему, которая предоставляет данные на устройство. Такой паттерн весьма оптимизирует работу как для клиента, так и для сервера.

Контекст предлагаемого нами подхода

Еще раз подчеркнем, что мы предлагаем писать приложения, в которых поставщики содержимого получают доступ к сетевым данным и кэшируют их, когда это возможно. На первый взгляд такой подход может показаться обременительным, но вспомните — ведь браузеры также используют асинхронный механизм загрузки содержимого, на которое ссылается уникальный идентификатор ресурса. Читателям, которые знакомы с основами веб-программирования, стандартный API системы Android может показаться еще более гибким и расширяемым, чем интерфейс программирования приложений, применяемый в AJAX. Но AJAX давно известен своей архитектурой, защищенной от неумелого обращения (так называемая «защита от дурака»). Современные браузеры загружают информацию, на которую указывают уникальные идентификаторы ресурсов, при помощи асинхронных механизмов ввода-вывода (http://en.wikipedia.org/wiki/Asynchronous_io), благодаря чему устраняется множество причин, по которым интерфейс браузера может зависнуть. На самом деле браузер выполняет не так много работы, если определенный URI (ссылку) не удастся загрузить, но все дело в том, что в такой ситуации исключается опасность блокировки пользовательского интерфейса, если сетевое соединение перестает отвечать. Если бы завис поток пользовательского интерфейса, то застопорилась бы работа всего браузера. Он даже не смог бы сообщить вам о том, что завис, — особенно потому, что многие браузеры являются однопоточными. Браузеры позволяют приостановить запрос на загрузку любой страницы, а потом загрузить другую страницу — возможно, новая загрузка пойдет быстрее. Более того, все современные браузеры используют веб-кэш для долговременного хранения информации, и мы просто рекомендуем выстраивать в приложениях Android аналогичную архитектуру.

Кроме описываемого нами паттерна, Google предлагает специальную документацию, предназначенную для ускорения реакции приложения и уменьшения вероятности получения уведомлений типа «Приложение не отвечает». Данная документация расположена по адресу <http://developer.android.com/guide/practices/design/responsiveness.html>.

Пример кода: динамическое построение списка и кэширование видеоконтента YouTube

Для демонстрации описанной архитектуры мы покажем приложение Finch для построения списков видеороликов, которое позволяет пользователю искать видео с мобильного устройства при помощи API с передачей состояния представления по адресу <http://gdata.youtube.com>. Написанный нами код рассчитан на непостоянство сетевого соединения в мобильном окружении. Приложение заранее сохраняет пользовательские данные, чтобы с ним можно было работать и тогда, когда

сетевое соединение отсутствует. Это может означать, что приложение будет отображать устаревшие результаты, сохраненные на локальном устройстве, но тем не менее будет работать.

Когда пользователь выполняет запрос, приложение пытается получить в ответ на этот запрос наиболее актуальные результаты с YouTube. Если приложение сможет успешно загрузить новые результаты, то оно сотрет все хранящиеся на устройстве результаты, которые старше одной недели. Если бы программа просто вслепую сбрасывала все старые результаты еще до того, как совершить запрос на обновление информации, то пользователь мог бы увидеть пустой экран — и приложение оставалось бы бесполезным до тех пор, пока не возобновится связь с сетью. На рис. 13.2 показан результат запроса по ключевому слову dogs («собаки»). При нажатии кнопки «Ввод» в поисковом поле или при нажатии кнопки обновления система порождает новый запрос.

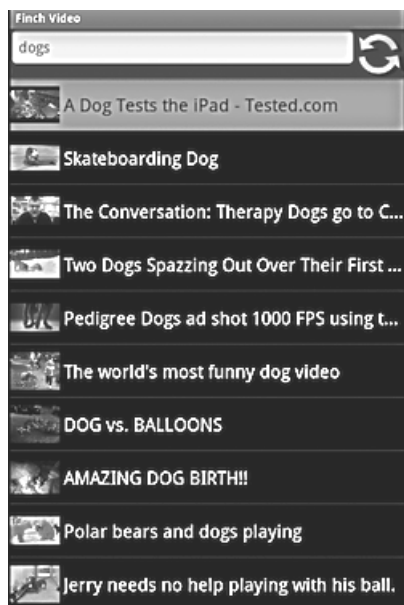


Рис. 13.2. Пример приложения Finch-видео

В нашем приложении имеется кэширующий поставщик содержимого, который запрашивает через API YouTube метаданные видео YouTube. Результаты запроса кэшируются в таблице SQL под названием `video`, в методе `query` поставщика содержимого. Поставщик содержимого использует фреймворк Finch для инициирования асинхронных REST-запросов. Пользовательский интерфейс состоит из активности, показанной на рис. 13.2, списка с поисковым полем и кнопки для обновления содержимого. Список динамически обновляется после уведомления поставщика содержимого о поступлении новых данных. Всякий раз, когда пользователь вводит поисковый запрос, а затем нажимает «Ввод», активность иницирует запрос к `FinchVideoContentProvider`, содержащий соответствующий уникальный идентификатор ресурса. Сейчас мы объясним этот пример детально.

Структура исходного кода для примера с Finch-видео при работе с YouTube

В данном разделе мы коротко рассмотрим соответствующий исходный код на языке Java, который относится к видеоприложению Finch для работы с YouTube и применяется только к простой версии нашей программы для построения списка видеороликов. Итак, нужные нам файлы находятся в двух различных каталогах: в первом лежит код приложения Finch-видео, рассмотренного в главе 12, а во втором — код библиотеки Finch Framework, на которой основан материал из главы 12. К исходным файлам, составляющим наше приложение для работы с YouTube, относятся следующие.

○ Файлы для главы 12 из каталога `$(FinchVideo)/src/`:

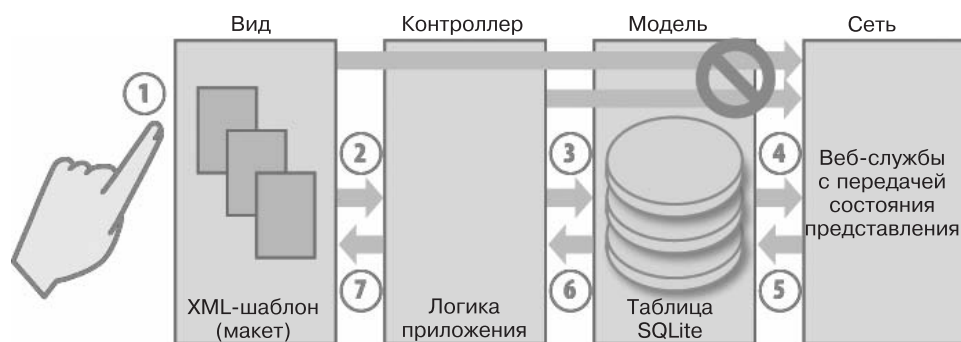
- `$(FinchVideo)/src/com/oreilly/demo/pa/finchvideo/FinchVideo.java` — класс `FinchVideo` содержит класс `Videos`, который функционально аналогичен классу `FinchVideo.SimpleVideos` из простого видеоприложения. В классе `FinchVideo.Videos` определяется еще несколько констант, в дополнение к названиям столбцов поставщика содержимого, определенных в нашей простой версии приложения для работы с YouTube. Ни в классе `FinchVideo`, ни в `Videos` нет никакого исполняемого кода;
- `$(FinchVideo)/src/com/oreilly/demo/pa/finchvideo/provider/FinchVideoContentProvider.java` — основной поставщик содержимого, дающий нам метаданные YouTube и выполняющий асинхронные RESTful-запросы к API GData сервиса YouTube;
- `$(FinchVideo)/lib-src/com/oreilly/demo/pa/finchvideo/provider/YouTubeHandler.java` — здесь осуществляется синтаксический разбор результатов, получаемых от API GData сервиса YouTube, и происходит вставка новых данных, по мере того как они прибывают.

○ Исходный код фреймворка Finch в каталоге `$(FinchFramework)/lib-src/`:

- `$(FinchFramework)/lib-src/com/finchframework/finch/rest/RESTfulContentProvider.java` — здесь содержится простой фреймворк для инициирования REST-запросов по протоколу HTTP, исходящих от поставщика содержимого Android. `FinchVideoContentProvider` дополняет этот класс, чтобы данное поведение можно было многократно использовать для асинхронного управления HTTP-запросами;
- `$(FinchFramework)/lib-src/com/finchframework/finch/rest/FileHandler.java`
`$(FinchFramework)/lib-src/com/finchframework/finch/rest/FileHandlerFactory.java` — простые фреймворки для загрузки содержимого, на которое ссылаются URI, в файловый кэш. Они обрабатывают ответ, когда приложение запрашивает уникальные идентификаторы ресурсов (в данном случае эскизов);
- `$(FinchFramework)/lib-src/com/finchframework/finch/rest/ResponseHandler.java` — здесь предоставляется простой уровень абстракции для управления содержимым, загруженным по HTTP с использованием API YouTube. Этот класс дополняется `YouTubeHandler`;
- `$(FinchFramework)/lib-src/com/finchframework/finch/rest/UriRequestTask.java` — это исполняемый объект, специально предназначенный для скачивания HTTP-содержимого. Он использует клиентский фреймворк Apache HTTP.

Пошаговая разработка поискового приложения

На рис. 13.3 схематически представлены этапы процесса, в ходе которого наш поставщик содержимого обслуживает поисковые запросы, поступающие от вида и контроллера. При этом используются сетевые запросы с передачей состояния представления. Поставщик содержимого может кэшировать получаемые из сети результаты в таблице SQLite, и лишь потом уведомлять наблюдателей, слушающих URI, которые связаны с нужными данными. Запросы могут проходить между компонентами в асинхронном режиме. Вид и контроллер не должны напрямую или синхронно инициировать собственные сетевые запросы.



Этап 1. Вид — пользовательский ввод.

Этап 2. Контроллер слушает события.

Этап 3. Контроллер инициирует управляемый запрос к поставщику содержимого (то есть к модели).

Этап 4. Поставщик содержимого инициирует REST-запрос к сервису YouTube и готовится к получению результатов.

Этап 5. YouTube отвечает, после чего инструмент синтаксического разбора вставляет результаты в SQL-кэш поставщика содержимого.

Этап 6. Поставщик содержимого направляет уведомления виду и контроллеру.

Этап 7. Вид обновляется

Рис. 13.3. Последовательность событий, в ходе которых реализуется запрос клиента на получение информации от поставщика содержимого

Оставшаяся часть главы посвящена пошаговому исследованию нашего второго видеоприложения Finch, в котором реализуется рассматриваемый паттерн написания программ для Android. Рекомендуем при чтении сверяться с этапами, схематически показанными на рис. 13.3. Обратите внимание и на то, что последовательность этапов не всегда будет соответствовать порядку, в котором мы описываем код. Чтобы не отрываться от описания кода, мы будем приводить номера этапов.

Этап 1. Пользовательский интерфейс собирает пользовательский ввод

Для получения ключевых слов поискового запроса в нашем пользовательском интерфейсе на рис. 13.2 используется обычное поле EditText.

Этап 2. Контроллер прослушивает события

FinchVideoActivity регистрирует слушатель текста, наш контроллер, получающий событие, когда пользователь нажимает клавишу «Ввод»:

```
class FinchVideoActivity {
    ...
    mSearchText.setOnEditorActionListener(
        new EditText.OnEditorActionListener() {
            public boolean onEditorAction(TextView textView,
                int actionId,
                KeyEvent keyEvent)
            {
                ...
                query();
                ...
            }
        }
    );
}
```

Этап 3. Контроллер запрашивает данные у поставщика содержимого/модели при помощи метода managedQuery

Затем контроллер вызывает метод query, это делается в ответ на пользовательский (поисковый) запрос:

```
// внутри FinchVideoActivity

...

// отправляется запрос к поставщику содержимого finch-видео
private void query() {
    if (!mSearchText.searchEmpty()) {
        String queryString =
            FinchVideo.Videos.QUERY_PARAM_NAME + "=" +
            Uri.encode(mSearchText.getText().toString());
        Uri queryUri =
            Uri.parse(FinchVideo.Videos.CONTENT_URI + "?" +
                queryString);
        Cursor c = managedQuery(queryUri, null, null, null, null);
        mAdapter.changeCursor(c);
    }
}
```

Этап 4. Реализация запроса с передачей состояния представления

Этап 4 несколько сложнее, чем предыдущие шаги описываемой последовательности. Необходимо разобраться с нашим поставщиком содержимого FinchVideoCon-

tentProvider (это поставщик с передачей состояния представления), как мы разбирались с SimpleFinchVideoContentProvider. Начнем с того, что FinchVideoContentProvider дополняет вспомогательный компонент RESTfulContentProvider, который, в свою очередь, дополняет ContentProvider:

```
FinchVideoContentProvider extend RESTfulContentProvider {
```

RESTfulContentProvider обеспечивает асинхронные операции с передачей состояния представления таким образом, что поставщик содержимого Finch может подключать собственные компоненты, предназначенные для обработки запросов и ответов. Чуть ниже мы объясним этот механизм более подробно, когда поговорим о нашем усовершенствованном методе query.

Константы и инициализация

Инициализация FinchVideoContentProvider очень напоминает процесс инициализации простого поставщика видео. Как и при работе с простым вариантом, сначала мы настраиваем механизм сопоставления URI. Нам придется решить лишь одну дополнительную задачу: обеспечить поддержку для сопоставления конкретных эскизов с содержимым. Мы не добавляем функцию сопоставления множественных эскизов с содержимым, поскольку нашей активности не требуется такая поддержка — в ней всего лишь понадобится загружать одиночные эскизы:

```
sUriMatcher.addURI(FinchVideo.AUTHORITY,  
    FinchVideo.Videos.THUMB + "/"#, THUMB_ID);
```

Создание базы данных

Создаем базу данных о видео Finch при помощи кода Java, выполняющего следующий запрос на SQL:

```
CREATE TABLE video (_ID INTEGER PRIMARY KEY AUTOINCREMENT,  
    title TEXT, description TEXT, thumb_url TEXT,  
    thumb_width TEXT, thumb_height TEXT, timestamp TEXT,  
    query_text TEXT, media_id TEXT UNIQUE);
```

Обратите внимание, что, в отличие от простой версии базы данных, в этом варианте мы добавили возможность сохранять следующие атрибуты:

- thumb_url, thumb_width, thumb_height — это URL, ширина и высота, связанные с эскизом конкретного видео;
- timestamp — при добавлении новой видеозаписи мы отмечаем точное время ее добавления;
- query_text — сохраняем в базе данных текст запроса (или ключевые слова запроса) в базе данных вместе с каждым результатом запроса;
- media_id — уникальное значение для каждого отклика с видео, получаемого от API GData. Мы не допускаем, чтобы две видеозаписи имели одинаковое значение media_id.

Метод query с сетевой функциональностью

Теперь поговорим о реализации метода `query` в классе `FinchYouTubeProvider`. В данном случае вызовы направляются в сеть для получения в качестве результатов запроса данных с YouTube. Для этого вызывается метод суперкласса, `RESTfulContentProvider.asyncQueryRequest(String queryTag, String queryUri)`. Здесь `queryTag` — это уникальная строка, позволяющая при необходимости отклонять дублирующиеся запросы, а `queryUri` — полный уникальный идентификатор ресурса, необходимый для асинхронной загрузки. В сущности, мы иницилируем запросы к следующему URI после прикрепления параметров запроса `URLEncoder.encode`, полученных из текстового поискового поля нашего приложения:

```
/** URI для запроса видео, ожидает прикрепленных ключевых слов */
private static final String QUERY_URI =
    "http://gdata.youtube.com/feeds/api/videos?" +
    "max-results=15&format=1&q=";
```



Вы без труда можете научиться сами создавать уникальные идентификаторы ресурсов, указывающие на раздел GData сайта YouTube, подходящие для использования в вашем приложении. Компания Google создала бета-версию этого вспомогательного раздела, она находится по адресу <http://gdata.youtube.com>. Открыв данную страницу у себя в браузере, вы увидите онлайн-пользовательский интерфейс, содержащий массу параметров, которые вы сами можете настраивать, чтобы создавать URI, подобные тому, что был приведен в предыдущем листинге. Мы воспользовались этим интерфейсом для выбора не более чем 15 результатов, а в качестве формата содержимого указали «мобильное видео».

Наш сетевой метод `query` выполняет обычное сопоставление URI, а потом добавляет следующие задачи, соответствующие четвертому этапу обозначенной нами последовательности — «Этап 4. Реализация запроса с передачей состояния представления»:

```
/**
 * Метод запроса поставщика содержимого, преобразующий его параметры
 * в запрос с передачей состояния представления,
 * свойственный для сервиса YouTube.
 *
 * @param uri ссылка на запрос на получение видео, строка запроса может
 * содержать "q='terms'". Ключевые слова отправляются
 * на гугловский API YouTube, где они используются
 * для поиска информации в базе видеоконтента YouTube.
 * @param projection
 * @param where не используется в данном поставщике содержимого
 * @param whereArgs не применяется в данном поставщике содержимого
 * @param sortOrder не используется в данном поставщике содержимого
 * @return курсор, содержащий результаты поискового запроса,
 * возвращенные с YouTube
 */
@Override
public Cursor query(Uri uri, String[] projection, String where,
```

```

        String[] whereArgs, String sortOrder)
{
    Cursor queryCursor;

    int match = sUriMatcher.match(uri);
    switch (match) {
        case VIDEOS:
            // Запрос передается за пределами полосы этого метода, по которой
            // сообщается другая информация, то есть не является аргументом.
            String queryText = uri.
                getQueryParameter(FinchVideo.Videos.QUERY_PARAM_NAME); ❶

            if (queryText == null) {
                // Нулевой курсор – допустимый аргумент для данного метода.
                // CursorAdapter.changeCursor(Cursor c), который интерпретирует
                // значение, полностью сбрасывая все состояние адаптера
                // так, что компонент, для которого курсор адаптирует данные,
                // не будет отображать никакого контента.
                return null;
            }

            String select = FinchVideo.Videos.QUERY_TEXT_NAME +
                " = '" + queryText + "'";

            // быстрый возврат уже сопоставленных данных
            queryCursor =
                mDb.query(VIDEOS_TABLE_NAME, projection,
                    select,
                    whereArgs,
                    null,
                    null, sortOrder); ❷

            // заставляем курсор наблюдать за сделанным запросом
            queryCursor.setNotificationUri(
                getContext().getContentResolver(), uri); ❸

        /*
         * Всегда пытаемся обновить результаты новейшими данными,
         * полученными из сети.
         *
         * Порождение асинхронного потока загрузки задачи гарантирует,
         * что процесс загрузки никак не заблокирует ни один из методов
         * поставщика содержимого и, следовательно,
         * не заблокирует поток пользовательского интерфейса.
         *
         * Пока запрос загружается, мы возвращаем клиенту курсор
         * с имеющимися данными.
         *
         * Если существующий курсор пуст, то пользовательский интерфейс не
         * отобразит никакого контента, пока не получит уведомления по URI.
         */
    }
}

```

```

        * Обновления контента, получаемые в процессе выполнения асинхронного
        * сетевого запроса, будут появляться в уже возвращенном курсоре,
        * поскольку запрос, который связан с этим курсором, будет
        * ассоциировать курсор с новыми прибывающими элементами.
        */
        if (!"".equals(queryText)) {
            asyncQueryRequest(queryText, QUERY_URI + encode(queryText)); ④
        }
        break;
    case VIDEO_ID:
    case THUMB_VIDEO_ID:
        long videoID = ContentUris.parseId(uri);
        queryCursor =
            mDb.query(VIDEOS_TABLE_NAME, projection,
                FinchVideo.Videos._ID + " = " + videoID,
                whereArgs, null, null, null);
        queryCursor.setNotificationUri(
            getContext().getContentResolver(), uri);
        break;
    case THUMB_ID:
        String uriString = uri.toString();
        int lastSlash = uriString.lastIndexOf("/");
        String mediaID = uriString.substring(lastSlash + 1);

        queryCursor =
            mDb.query(VIDEOS_TABLE_NAME, projection,
                FinchVideo.Videos.MEDIA_ID_NAME + " = " +
                    mediaID,
                whereArgs, null, null, null);
        queryCursor.setNotificationUri(
            getContext().getContentResolver(), uri);
        break;

    default:
        throw new IllegalArgumentException("unsupported uri: " +
            QUERY_URI);
    }

    return queryCursor;
}

```

Вот несколько моментов в коде, на которые следует обратить внимание.

- ① Извлекаем параметр запроса из входящего URI. Этот параметр требуется посылать методу query в самом URI, а не вместе с другими аргументами, поскольку прочие аргументы выполняют в методе query иные функции и не могут применяться для содержания в них ключевых слов запроса.
- ② Сначала ищем в базе данных уже имеющуюся там информацию, которая удовлетворяет ключевым словам запроса.
- ③ Задаем URI уведомления так, чтобы курсоры, возвращенные от метода query, получали события обновления во всех тех случаях, когда поставщик содержи-

мого обновляет информацию, отслеживаемую этими курсорами. Данное действие готовит почву для этапа 6 нашей последовательности, обеспечивающего обновление вида, когда поставщик содержимого инициирует события обновления. События обновления происходят при изменении данных, а сами данные изменяются, когда поставщик содержимого получает ответ на сделанный запрос. После прибытия уведомления происходит перерисовка пользовательского интерфейса, то есть этап 7. В данном описании этапы 6 и 7 идут «вне очереди», но мы считаем, что об этих этапах уместно рассказать именно сейчас, так как они связаны с URI уведомлений и запросами.

- 4 Порождение асинхронного потока для загрузки заданного URI запроса. Метод `asyncQueryRequest` включает в себе создание нового потока, обслуживающего каждый запрос. В нашей схеме это этап 5; асинхронный запрос порождает поток именно для того, чтобы установить сетевое соединение, а сервис YouTube возвратит ответ.

RESTfulContentProvider: вспомогательный класс для REST

Теперь рассмотрим поведения, которые `FinchVideoProvider` наследует от `RESTfulContentProvider`. Эти поведения требуются для того, чтобы выполнять запросы с передачей состояния представления. Для начала изучим поведение отдельного запроса к YouTube: как мы уже видели, запросы запускаются в асинхронном режиме из главного потока. REST-поставщик должен уметь обрабатывать особые случаи. Так, если пользователь делает запрос по ключевым словам «Смешные котята» и в то же время уже выполняется другой запрос по тем же ключевым словам, то наш поставщик содержимого сбросит второй запрос. С другой стороны, если пользователь сделал запрос по ключевому слову «Собаки», а затем, когда этот запрос еще не завершился, делает запрос по ключевому слову «Коты», то поставщик содержимого позволяет запросам по словам «Собаки» и «Коты» выполняться параллельно. Ведь позже пользователь может повторить запрос по слову «Собаки» — и тут ему пригодятся кэшированные результаты, которые уже заинтересовали его ранее.

Класс `RESTfulContentProvider` позволяет подклассу асинхронно порождать запросы, и когда данные по запросу прибывают, `RESTfulContentProvider` дает возможность производить настраиваемую обработку (`custom handling`) ответа. Для этого применяется простой подключаемый интерфейс, называемый `ResponseHandler`. Подклассы должны переопределять абстрактный метод `RESTfulContentProvider.newResponseHandler` для возврата обработчиков, специально приспособленных для синтаксического разбора данных ответа, запрашиваемых их базовым поставщиком. Каждый обработчик переопределяет метод `ResponseHandler.handleResponse` (`HttpResponse`), чтобы обеспечить настраиваемую обработку сущностей `HttpEntity`, содержащихся в переданных объектах `HttpResponse`. Например, наш поставщик содержимого применяет обработчик `YouTubeHandler` для синтаксического разбора RSS-ленты, используемой в YouTube, вставляя в базу данных о видео новые строки для каждой из считанных записей. Чуть ниже мы поговорим об этом немного подробнее.

Кроме того, класс `RESTfulContentProvider` позволяет подклассу с легкостью выполнять асинхронные запросы и отклонять дублирующиеся запросы. `RESTfulContentProvider`

сопровождает каждый запрос уникальной меткой (тегом), благодаря чему подкласс и может сбрасывать дублирующиеся запросы. Наш поставщик содержимого `FinchVideoContentProvider` применяет в качестве метки запроса ключевые слова, введенные пользователем, поскольку они уникально идентифицируют конкретный поисковый запрос.

Поставщик содержимого `FinchVideoContentProvider` переопределяет `newResponseHandler` следующим образом:

```
/**
 * Предоставляет обработчик, который может выполнить синтаксический
 * разбор содержимого RSS-ленты с ресурса YouTube *GData.
 *
 * @param requestTag уникальная метка, идентифицирующая данный запрос
 * @return объект YouTubeHandler
 */
@Override
protected ResponseHandler newResponseHandler(String requestTag) {
    return new YouTubeHandler(this, requestTag);
}
```

Теперь обсудим реализацию `RESTfulContentProvider`, это поможет нам объяснить те операции, которые он позволяет осуществлять в подклассах. Класс `UriRequestTask` реализует интерфейс `Runnable` для асинхронного исполнения запросов с передачей состояния представления. `RESTfulContentProvider` использует ассоциативный контейнер `RequestsInProgress` со строкой в качестве ключа, что гарантирует уникальность всех запросов:

```
/**
 * Включает в себя функции для выполнения асинхронных REST-запросов,
 * так, что поставщики содержимого, выполненные в виде подклассов,
 * могут использовать их для инициирования запросов,
 * продолжая применять при этом и специальные методы для интерпретации
 * контента на базе REST, в частности RSS, ATOM, JSON и т. д.
 */
public abstract class RESTfulContentProvider extends ContentProvider {
    protected FileHandlerFactory mFileHandlerFactory;
    private Map<String, UriRequestTask> mRequestsInProgress =
        new HashMap<String, UriRequestTask>();

    public RESTfulContentProvider(FileHandlerFactory fileHandlerFactory) {
        mFileHandlerFactory = fileHandlerFactory;
    }

    public abstract Uri insert(Uri uri, ContentValues cv, SQLiteDatabase db);

    private UriRequestTask getRequestTask(String queryText) {
        return mRequestsInProgress.get(queryText); ❶
    }

    /**
     * Позволяет подклассу определять базу данных, которую будет
     * использовать обработчик откликов.
     */
}
```

```

*
* @return база данных, информация о которой передается
* обработчику ответов
*/
public abstract SQLiteDatabase getDatabase();

public void requestComplete(String mQueryText) {
    synchronized (mRequestsInProgress) {
        mRequestsInProgress.remove(mQueryText); ❷
    }
}

/**
* Абстрактный метод, позволяющий подклассу определять тип обработчика,
* который должен использоваться для синтаксического разбора ответа,
* полученного на тот или иной запрос.
*
* @param requestTag уникальная метка, идентифицирующая данный запрос
* @return Обработчик ответов, созданный подклассом и применяемый
* для синтаксического разбора полученного ответа
*/
protected abstract ResponseHandler newResponseHandler(String
    requestTag);

UriRequestTask newQueryTask(String requestTag, String url) {
    UriRequestTask requestTask;

    final HttpGet get = new HttpGet(url);
    ResponseHandler handler = newResponseHandler(requestTag);
    requestTask = new UriRequestTask(requestTag, this, get, ❸
        handler, getContext());

    mRequestsInProgress.put(requestTag, requestTask);
    return requestTask;
}

/**
* Создается новый рабочий поток, предназначенный для активации сетевого
* соединения с передачей состояния представления.
*
* @param queryTag уникальная метка, идентифицирующая данный запрос
*
* @param queryUri полный уникальный идентификатор ресурса,
* к которому должен быть получен доступ в результате данного запроса
*/
public void asyncQueryRequest(String queryTag, String queryUri) {
    synchronized (mRequestsInProgress) {
        UriRequestTask requestTask = getRequestTask(queryTag);
        if (requestTask == null) {
            requestTask = newQueryTask(queryTag, queryUri); ❹
            Thread t = new Thread(requestTask);

```

```

        // позволяет параллельное исполнение других запросов
        t.start();
    }
}
...
}

```

Рассмотрим пояснения к коду.

- ❶ Метод `getRequestTask` использует `mRequestsInProgress` для доступа к любым выполняемым в текущий момент идентичным запросам. Таким образом, `asyncQueryRequest` может блокировать дублирующиеся запросы при помощи обычного оператора `if`.
- ❷ Когда запрос завершается после возврата метода `ResponseHandler.handleResponse`, `RESTfulContentProvider` удаляет задачу из своего `mRequestsInProgress`.
- ❸ `newQueryTask` создает экземпляры `UriRequestTask`, являющиеся при этом экземплярами `Runnable`. В свою очередь, `Runnable` открывает HTTP-соединение, а затем вызывает `handleResponse` применительно к подходящему обработчику.
- ❹ Наконец, в нашем коде есть уникальный запрос, создается задача для его запуска, а потом задача заключается в поток, так обеспечивается возможность ее асинхронного выполнения.

Конечно, `RESTfulContentProvider` содержит суть системы многократно используемых задач, но для полноты картины мы продемонстрируем и другие компоненты нашего фреймворка.

UriRequestTask. `UriRequestTask` содержит в себе асинхронные аспекты обработки REST-запроса. Это простой класс, обладающий полями, которые позволяют ему выполнять операции GET с передачей состояния представления в его методе `run`. Функционально такое действие относится к этапу 4 нашей последовательности — «Реализация запроса с передачей состояния представления». Как мы уже говорили, после того как получен ответ на запрос, этот ответ передается вызовом метода `ResponseHandler.handleResponse`. Предполагается, что метод `handleResponse` будет добавлять в базу данных новые записи по мере необходимости. Этот процесс будет показан в `YouTubeHandler`:

```

/**
 * Представляет собой интерфейс runnable, использующий HttpClient
 * для асинхронной загрузки заданного URI. После того как содержимое
 * из сети будет загружено, задача делегирует обработку запроса обработчику
 * ResponseHandler, специализированному для работы с контентом данной
 * разновидности.
 */
public class UriRequestTask implements Runnable {
    private HttpUriRequest mRequest;
    private ResponseHandler mHandler;

    protected Context mAppContext;

    private RESTfulContentProvider mSiteProvider;

```

```

private String mRequestTag;

private int mRawResponse = -1;
// private int mRawResponse = R.raw.map_src;

public UriRequestTask(HttpUriRequest request,
                      ResponseHandler handler, Context appContext)
{
    this(null, null, request, handler, appContext);
}

public UriRequestTask(String requestTag,
                      RESTfulContentProvider siteProvider,
                      HttpUriRequest request,
                      ResponseHandler handler, Context appContext)
{
    mRequestTag = requestTag;
    mSiteProvider = siteProvider;
    mRequest = request;
    mHandler = handler;
    mAppContext = appContext;
}

public void setRawResponse(int rawResponse) {
    mRawResponse = rawResponse;
}

/**
 * Выполняет запрос к полностью прописанному URI, в соответствии
 * с данными о протоколе, хосте и порте, указанными в конфигурации,
 * а также в соответствии с уникальным идентификатором ресурса,
 * переданным конструктору.
 */
public void run() {
    HttpResponse response;

    try {
        response = execute(mRequest);
        mHandler.handleResponse(response, getUri());
    } catch (IOException e) {
        Log.w(Finch.LOG_TAG, "exception processing asynch request", e);
    } finally {
        if (mSiteProvider != null) {
            mSiteProvider.requestComplete(mRequestTag);
        }
    }
}

private HttpResponse execute(HttpUriRequest mRequest) throws IOException {
    if (mRawResponse >= 0) {

```

```

        return new RawResponse(mAppContext, mRawResponse);
    } else {
        HttpClient client = new DefaultHttpClient();
        return client.execute(mRequest);
    }
}

public Uri getUri() {
    return Uri.parse(mRequest.getURI().toString());
}
}

```

YouTubeHandler. Поскольку используется абстрактный метод `RESTfulContentProvider.newResponseHandler`, мы наблюдали, что наш поставщик содержимого `FinchVideoContentProvider` возвращает `YouTubeHandler` для обработки RSS-лента с YouTube. `YouTubeHandler` использует для синтаксического разбора входящих данных инструмент XML Pull, экономно расходующий память. Этот инструмент итерирует запрошенные данные RSS в формате XML. В `YouTubeHandler` есть некоторые сложные моменты, но, в принципе, он просто подбирает XML-теги, необходимые для создания объекта `ContentValues`. Потом `YouTubeHandler` сможет вставить этот объект в базу данных поставщика содержимого `FinchVideoContentProvider`. Часть этапа 5 осуществляется в тот момент, когда обработчик вставляет содержимое, прошедшее синтаксический разбор, в базу данных поставщика содержимого:

```

/**
 * Синтаксический разбор данных YouTube Entity и вставка их
 * в поставщик finch-видео.
 */
public class YouTubeHandler implements ResponseHandler {
    public static final String MEDIA = "media";
    public static final String GROUP = "group";
    public static final String DESCRIPTION = "description";
    public static final String THUMBNAIL = "thumbnail";
    public static final String TITLE = "title";
    public static final String CONTENT = "content";

    public static final String WIDTH = "width";
    public static final String HEIGHT = "height";

    public static final String YT = "yt";
    public static final String DURATION = "duration";
    public static final String FORMAT = "format";

    public static final String URI = "uri";
    public static final String THUMB_URI = "thumb_uri";

    public static final String MOBILE_FORMAT = "1";

    public static final String ENTRY = "entry";

```

```

public static final String ID = "id";

private static final String FLUSH_TIME = "5 minutes";

private RESTfulContentProvider mFinchVideoProvider;

private String mQueryText;
private boolean isEntry;

public YouTubeHandler(RESTfulContentProvider restfulProvider,
                      String queryText)

{
    mFinchVideoProvider = restfulProvider;
    mQueryText = queryText;
}

/*
 * Обрабатывает ответ, получаемый от сервера YouTube GData, который
 * имеет форму RSS-ленты, содержащей ссылки на видеоролики YouTube.
 */
public void handleResponse(HttpResponse response, Uri uri)
    throws IOException
{
    try {
        int newCount = parseYoutubeEntity(response.getEntity()); ❶

        // сбрасывать старое состояние только после того,
        // как придет новое состояние
        if (newCount > 0) {
            deleteOld();
        }

    } catch (IOException e) {
        // Использовать исключение, чтобы избежать удаления старого
        // состояния, если мы не можем получить новое. Таким образом,
        // мы оставляем приложение с определенными данными,
        // с которыми можно работать при отсутствии сетевого соединения.

        // Можно повторить запрос данных в расчете на то,
        // что соединение с сетью могло восстановиться.
    }
}

private void deleteOld() {
    // Удалить все устаревшие элементы, а не только те,
    // которые соответствуют актуальному запросу.

    Cursor old = null;

    try {
        SQLiteDatabase db = mFinchVideoProvider.getDatabase();

```

```

old = db.query(FinchVideo.Videos.VIDEO, null,
    "video." + FinchVideo.Videos.TIMESTAMP +
        " < strftime('%s', 'now', '-'" + FLUSH_TIME + "'")",
    null, null, null, null);
int c = old.getCount();
if (old.getCount() > 0) {
    StringBuffer sb = new StringBuffer();
    boolean next;
    if (old.moveToNext()) {
        do {
            String ID = old.getString(FinchVideo.ID_COLUMN);
            sb.append(FinchVideo.Videos._ID);
            sb.append(" = ");
            sb.append(ID);

            // Избавляемся от ассоциированных кэшированных
            // файлов эскизов.
            mFinchVideoProvider.deleteFile(ID);

            next = old.moveToNext();
            if (next) {
                sb.append(" OR ");
            }
        } while (next);
    }
    String where = sb.toString();

    db.delete(FinchVideo.Videos.VIDEO, where, null);

    Log.d(Finch.LOG_TAG, "flushed old query results: " + c);
}
} finally {
    if (old != null) {
        old.close();
    }
}
}

private int parseYoutubeEntity(HttpEntity entity) throws IOException {
    InputStream youTubeContent = entity.getContent();
    InputStreamReader inputReader = new InputStreamReader(youTubeContent);
    int inserted = 0;

    try {
        XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
        factory.setNamespaceAware(false);
        XmlPullParser xpp = factory.newPullParser();

        xpp.setInput(inputReader);

        int eventType = xpp.getEventType();

```

```

String startName = null;
ContentValues mediaEntry = null;

// Итеративный "вытягивающий" синтаксический разбор –
// это удобный способ извлечения данных из потоков,
// так как нам не приходится держать в памяти объектную модель
// документа на этапе синтаксического разбора.

while (eventType != XmlPullParser.END_DOCUMENT) {
    if (eventType == XmlPullParser.START_DOCUMENT) {
    } else if (eventType == XmlPullParser.END_DOCUMENT) {
    } else if (eventType == XmlPullParser.START_TAG) {
        startName = xpp.getName();

        if ((startName != null)) {

            if ((ENTRY).equals(startName)) {
                mediaEntry = new ContentValues();
                mediaEntry.put(
                    FinchVideo.Videos.QUERY_TEXT_NAME,
                    mQueryText);
            }

            if ((MEDIA + ":" + CONTENT).equals(startName)) {
                int c = xpp.getAttributeCount();
                String mediaUri = null;
                boolean isMobileFormat = false;

                for (int i = 0; i < c; i++) {
                    String attrName = xpp.getAttributeName(i);
                    String attrValue = xpp.getAttributeValue(i);

                    if ((attrName != null) &&
                        URI.equals(attrName))
                    {
                        mediaUri = attrValue;
                    }

                    if ((attrName != null) && (YT + ":" +
                        FORMAT).equals(MOBILE_FORMAT))
                    {
                        isMobileFormat = true;
                    }
                }

                if (isMobileFormat && (mediaUri != null)) {
                    mediaEntry.put(URI, mediaUri);
                }
            }

            if ((MEDIA + ":" + THUMBNAIL).equals(startName)) {

```

```

int c = xpp.getAttributeCount();
for (int i = 0; i < c; i++) {
    String attrName = xpp.getAttributeName(i);
    String attrValue = xpp.getAttributeValue(i);

    if (attrName != null) {
        if ("url".equals(attrName)) {
            mediaEntry.put(
                FinchVideo.Videos.
                    THUMB_URI_NAME,
                attrValue);
        } else if (WIDTH.equals(attrName)) {
            {
                mediaEntry.put(
                    FinchVideo.Videos.
                        THUMB_WIDTH_NAME,
                        attrValue);
            }
        } else if (HEIGHT.equals(attrName)) {
            {
                mediaEntry.put(
                    FinchVideo.Videos.
                        THUMB_HEIGHT_NAME,
                        attrValue);
            }
        }
    }
}

if (ENTRY.equals(startName)) {
    isEntry = true;
}

} else if(eventType == XmlPullParser.END_TAG) {
    String endName = xpp.getName();

    if (endName != null) {
        if (ENTRY.equals(endName)) {
            isEntry = false;
        } else if (endName.equals(MEDIA + ":" + GROUP)) {
            // вставка всей группы медиа
            inserted++;

            // Напрямую применяем insert к поставщику
            // finch-видео, не используя преобразователь
            // содержимого. Мы не хотим, чтобы поставщик
            // содержимого синхронизировал эти данные
            // обратно на себя.
            SQLiteDatabase db =
                mFinchVideoProvider.getDatabase();

            String mediaID = (String) mediaEntry.get(

```

```

        FinchVideo.Videos.MEDIA_ID_NAME);

    // вставка uri эскиза
    String thumbContentUri =
        FinchVideo.Videos.THUMB_URI + "/" + mediaID;
    mediaEntry.put(FinchVideo.Videos.
        THUMB_CONTENT_URI_NAME,
        thumbContentUri);

    String cacheFileName =
        mFinchVideoProvider.getCacheName(mediaID);
    mediaEntry.put(FinchVideo.Videos._DATA,
        cacheFileName);

    Uri providerUri = mFinchVideoProvider.
        insert(FinchVideo.Videos.CONTENT_URI,
            mediaEntry, db); ❷
    if (providerUri != null) {
        String thumbUri = (String) mediaEntry.
            get(FinchVideo.Videos.THUMB_URI_NAME);

        // Можно попробовать "ленивую" загрузку
        // изображения, чтобы оно загружалось только
        // по мере просмотра страницы.
        // Более активная загрузка также может
        // улучшить производительность программы.

        mFinchVideoProvider.
            cacheUri2File(String.valueOf(ID),
                thumbUri); ❸
    }
}

} else if (eventType == XmlPullParser.TEXT) {
    // перевод строки может превратиться в дополнительное
    // текстовое событие
    String text = xpp.getText();
    if (text != null) {
        text = text.trim();
        if ((startName != null) && (!"".equals(text))) {
            if (ID.equals(startName) && isEntry) {
                int lastSlash = text.lastIndexOf("/");
                String entryId =
                    text.substring(lastSlash + 1);
                mediaEntry.put(
                    FinchVideo.Videos.MEDIA_ID_NAME,
                    entryId);
            } else if ((MEDIA + ":" + TITLE).
                equals(startName)) {
                mediaEntry.put(TITLE, text);
            }
        }
    }
}

```

```

        } else if ((MEDIA + ":" +
            DESCRIPTION).equals(startName))
        {
            mediaEntry.put(DESCRIPTION, text);
        }
    }
}
eventType = xpp.next();
}

// В качестве альтернативной схемы уведомлений уведомления можно
// осуществлять только после того, как будут вставлены все записи.

} catch (XmlPullParserException e) {
    Log.d(Ch11.LOG_TAG,
        „could not parse video feed“, e);
} catch (IOException e) {
    Log.d(Ch11.LOG_TAG,
        „could not process video stream“, e);
}

return inserted;
}
}

```

Пояснения к коду следующие.

- ❶ Наш обработчик реализует `handleResponse`, разбирая сущность YouTube HTTP в своем методе, `parseYoutubeEntity`, вставляющем новые видеоданные. Затем обработчик удаляет старые видеоданные, запрашивая элементы, которые старше, чем период задержки, и удаляет строки с данными в этом запросе.
- ❷ Обработчик закончил синтаксический разбор медийного элемента и использует содержащуюся в нем ссылку на поставщик содержимого для вставки только что разобранного объекта `ContentValues`. Обратите внимание, что этот процесс относится к этапу 5 нашей последовательности — «Обработчик ответа вставляет элементы в локальный кэш».
- ❸ Поставщик содержимого инициирует собственный асинхронный запрос после вставки новой записи с медиаинформацией, чтобы также загрузить и содержимое-эскиз. Чуть ниже мы подробнее расскажем об этой функции нашего поставщика содержимого.

insert и ResponseHandlers

Теперь рассмотрим этап 5 более подробно. Наш поставщик Finch-видео реализует `insert` во многом так же, как и простой поставщик видео. Как мы уже видели в нашем приложении, вставка видео происходит как побочный эффект метода `query`. Стоит еще раз подчеркнуть, что наш метод `insert` делится на две части. Мы хотим,

чтобы клиенты поставщика содержимого вызывали первую форму, а обработчики ответов (объекты `ResponseHandler`) — вторую. Обе эти формы показаны в следующем коде. Первая форма делегирует задачу второй. Мы разделяем `insert` потому, что обработчик ответов входит в состав поставщика содержимого и не должен соединяться сам с собой через преобразователь содержимого:

```
@Override
public Uri insert(Uri uri, ContentValues initialValues) {
    // валидация запрошенного уникального идентификатора ресурса
    if (sUriMatcher.match(uri) != VIDEOS) {
        throw new IllegalArgumentException("Unknown URI " + uri);
    }

    ContentValues values;
    if (initialValues != null) {
        values = new ContentValues(initialValues);
    } else {
        values = new ContentValues();
    }

    SQLiteDatabase db = getDatabase();
    return insert(uri, initialValues, db);
}
```

`YouTubeHandler` использует следующий метод для того, чтобы напрямую вставлять строки в простую базу данных видео. Обратите внимание, что мы не вставляем медиаэлемент, если в базе данных уже есть запись о видео с таким же `mediaID`, как у вставляемого нами элемента. Таким образом, мы избегаем попадания в базу одинаковых записей. Такое «дублирование» возможно при сведении воедино новых данных и уже имеющихся, которые, однако, не устарели:

```
public Uri insert(Uri uri, ContentValues values, SQLiteDatabase db) {
    verifyValues(values);

    // валидация запрошенного uri
    int m = sUriMatcher.match(uri);
    if (m != VIDEOS) {
        throw new IllegalArgumentException("Unknown URI " + uri);
    }

    // вставка значений в новую строку базы данных
    String mediaID = (String) values.get(FinchVideo.Videos.MEDIA_ID);

    Long rowID = mediaExists(db, mediaID);
    if (rowID == null) {
        long time = System.currentTimeMillis();
        values.put(FinchVideo.Videos.TIMESTAMP, time);
        long rowId = db.insert(VIDEOS_TABLE_NAME,
            FinchVideo.Videos.VIDEO, values);
        if (rowId >= 0) {
            Uri insertUri =
```

```

        ContentUris.withAppendedId(
            FinchVideo.Videos.CONTENT_URI, rowId);
        mContentResolver.notifyChange(insertUri, null);
        return insertUri;
    } else {
        throw new IllegalStateException("could not insert " +
            "content values: " + values);
    }
}

return ContentUris.withAppendedId(FinchVideo.Videos.CONTENT_URI, rowId);
}

```

Управление файлами: сохранение эскизов

Теперь, когда мы рассказали, как функционирует каркас нашего REST-поставщика, в заключение главы поговорим о том, как поставщик содержимого обращается с эскизами.

Выше мы рассматривали метод `ContentResolver.openInputStream`, при помощи которого поставщики содержимого могут предоставлять файлы клиентам. В нашем примере с видео *Finch* мы используем эту функцию для предоставления изображений-эскизов. Если сохранять изображения как файлы, то можно не иметь дел с блоками базы данных, которые всегда негативно влияют на производительность приложения, и можно загружать изображения только тогда, когда клиент их запрашивает. Чтобы поставщик содержимого мог предоставлять файлы, ему *необходимо* переопределить метод `ContentProvider.openFile`, открывающий дескриптор доставляемого файла. Преобразователь содержимого занимается созданием входного потока из дескриптора файла. Простейшая реализация этого метода будет вызывать `openFileHelper` для активации вспомогательного механизма, позволяющего `ContentResolver` считывать переменную `_data` для загрузки того файла, на который он ссылается. Если поставщик содержимого вообще не переопределит этот метод, то система выдаст исключение **No files supported by provider at...** (Поставщик содержимого не поддерживает никаких файлов по адресу...). В нашей простой реализации предоставляется доступ только для чтения, как показано в следующем коде:

```

/**
 * Предоставляет доступ только для чтения к файлам, которые были загружены
 * и сохранены в кэше поставщика содержимого. В частности, в данном
 * поставщике клиенты могут получать доступ к файлам загруженных
 * изображений-эскизов.
 */
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode)
    throws FileNotFoundException
{
    // поддерживаются лишь файлы, доступные только для чтения
    if (!"r".equals(mode.toLowerCase())) {
        throw new FileNotFoundException("Unsupported mode, " +

```

```

        mode + ", for uri: " + uri);
    }

    return openFileHelper(uri, mode);
}

```

Наконец, мы используем вариант реализации `ResponseHandler`, называемый `FileHandler`, для загрузки данных изображения-эскиза, на которое ссылается URL с YouTube. **Каждый URL соответствует определенной медиазаписи. Наша фабрика `FileHandlerFactory` позволяет управлять файлами из кэша, сохраненными в специальном каталоге для кэша. Мы позволяем нашей фабрике определять, где сохранять файлы:**

```

/**
 * Создаются экземпляры объектов FileHandler, которые используют общий
 * кэш-каталог. Кэш-каталог задается в конструкторе как фабрика
 * обработчиков файлов (FileHandlerFactory).
 */
public class FileHandlerFactory {
    private String mCacheDir;

    public FileHandlerFactory(String cacheDir) {
        mCacheDir = cacheDir;
        init();
    }

    private void init() {
        File cacheDir = new File(mCacheDir);
        if (!cacheDir.exists()) {
            cacheDir.mkdir();
        }
    }

    public FileHandler newFileHandler(String id) {
        return new FileHandler(mCacheDir, id);
    }

    // На самом деле не применяется, поскольку ContentResolver
    // использует поле _data.
    public File getFile(String ID) {
        String cachePath = getFileName(ID);

        File cacheFile = new File(cachePath);
        if (cacheFile.exists()) {
            return cacheFile;
        }
        return null;
    }

    public void delete(String ID) {
        String cachePath = mCacheDir + "/" + ID;

```

```
        File cacheFile = new File(cachePath);
        if (cacheFile.exists()) {
            cacheFile.delete();
        }
    }

    public String getFileName(String ID) {
        return mCacheDir + "/" + ID;
    }
}

/**
 * Записывает данные из URL в локальный файловый кэш, на который может
 * ссылаться идентификатор базы данных.
 */
public class FileHandler implements ResponseHandler {
    private String mId;
    private String mCacheDir;

    public FileHandler(String cacheDir, String id) {
        mCacheDir = cacheDir;
        mId = id;
    }

    public
    String getFileName(String ID) {
        return mCacheDir + "/" + ID;
    }

    public void handleResponse(HttpResponse response, Uri uri)
        throws IOException
    {
        InputStream urlStream = response.getEntity().getContent();
        FileOutputStream fout =
            new FileOutputStream(getFileName(mId));
        byte[] bytes = new byte[256];
        int r = 0;
        do {
            r = urlStream.read(bytes);
            if (r >= 0) {
                fout.write(bytes, 0, r);
            }
        } while (r >= 0);

        urlStream.close();
        fout.close();
    }
}
```

ЧАСТЬ IV

Продвинутые темы

Глава 14. Поиск

Глава 15. Геолокация и картография

Глава 16. Мультимедиа

Глава 17. Сенсоры, коммуникация ближнего поля, речь, жесты и доступность

Глава 18. Коммуникация, личные данные, синхронизация и социальные сети

Глава 19. Комплект для нативной разработки в Android (NDK)

В части IV мы рассмотрим API Android, важные при работе со многими приложениями, но не входящие в основной фреймворк Android, который используется практически в любых приложениях.

14 Поиск

При обсуждении Android сложно не говорить о Google. А Google — это практически синоним слова «поиск». Поисковая функция предоставила пользователям революционную возможность для получения конкретной информации на базе запроса. Для этой цели в Android предоставляется универсальный интерфейс, к которому относится Quick Search Box (Окно для быстрого поиска) и Search Bar (Панель поиска). Таким образом, идея поиска в Android становится всепроникающей. Итак, существует поисковый фреймворк, который является фреймворком пользовательского интерфейса и которым мы очень рекомендуем пользоваться.

Поисковый интерфейс

Поисковый фреймворк обеспечивает поиск по приложению. Но учитывайте, что это всего лишь фреймворк пользовательского интерфейса, не предоставляющий основу для реализации поисковой логики. Он просто содержит элементы пользовательского интерфейса, позволяющие пользователю вводить поисковый запрос и выполнять его. Эта операция, в свою очередь, вызывает заданную вами поисковую логику и возвращает соответствующие результаты. Чтобы продемонстрировать основы построения поисковой логики, а также поисковый интерфейс, мы исследуем образец поискового приложения, которое позволит пользователю выполнять поиск по сонетам Шекспира.

Основы поиска

Для обеспечения поиска приложение должно обладать несколькими важными свойствами. Требуется логика, которая будет возвращать результаты поиска. Кроме того, нужна конфигурация, допускающая возможность поиска, то есть такая конфигурация должна иметь специфические характеристики, применяемые при наполнении пользовательского интерфейса информацией и при выполнении поиска. В итоге запускается поисковая активность, которая получает запрос, и после вызова поисковой логики в этой активности выводятся результаты.

Поисковая логика

Существует несколько способов создания поисковой логики, на основе которой генерируются результаты поиска. Мы рассмотрим два способа: простой поиск по

индексу и поиск, подкрепленный применением базы данных SQLite, — `android.database.sqlite`.

Начнем с основополагающих элементов: объектов данных и интерфейса `SearchLogic`.

В этом примере, в сущности, есть один объект данных и один меньший подобъект. Поскольку мы выполняем поиск по сонетам, создадим класс `Sonnet`, содержащий название, номер сонета и строки этого сонета. Меньший подобъект — это фрагмент `SonnetFragment`. Для него нам нужно получить лишь конкретную строку того или иного сонета, не генерируя целый сонет. Подобъект используется в первую очередь для отображения результатов поиска.

```
public class Sonnet {  
    public int num;  
    public String title;  
    public String[] lines;  
}
```

```
public class SonnetFragment {  
    public int num;  
    public String line;  
}
```

В интерфейсе `SearchLogic` нам потребуется написать два метода, которые будут вызываться в ходе поиска или получения сонета через пользовательский интерфейс. Это метод `search()`, который принимает строку запроса и возвращает отсортированный массив фрагментов `SonnetFragment`, и метод `getSonnet()`, возвращающий конкретный объект `Sonnet`. На такой сонет мы ссылаемся по его номеру.

```
public interface SearchLogicInterface {  
    SonnetFragment[] search(String query);  
    Sonnet getSonnet(int i);  
}
```

Индексная поисковая логика. Чтобы подготовить такую логику, выводим все сонеты в единый необработанный файл и проводим синтаксический разбор каждой строки. Затем каждая группа строк, составляющая сонет, обрабатывается как объект `Sonnet`, и каждое слово каждой строки учитывается в более крупном индексе как ключ. Его значением является множество, каждый член которого представляет собой объект `SonnetRef`, содержащий ID сонета, в котором находится данное слово. Значение также включает массив с номерами строк, в которых содержится слово. Существуют, разумеется, и более совершенные методы, например отслеживание расположения слова (где позиция слова в строке сонета включается в качестве значения) с применением метаданных для ссылки на значение слова в сонете или на контекст, в котором оно употребляется. Каждое слово в сонете «взвешивается» таким образом, чтобы создать пословную систему ранжирования в каждом сонете. Есть и более точные методы для обработки более конкретных поисковых запросов или генерирования более точных результатов поиска. Но мы ограничимся лишь рассмотрением простейшей системы индексного поиска.

Если готовый индексный список уже доступен приложению, то после запроса конкретного слова результат возвращается очень быстро. Это объясняется тем, что вся логика, которую требуется выполнить, — найти искомое слово в индексном списке и получить ссылающиеся на него сонеты и строки, в данном случае значения из списка. В нашем примере мы используем в качестве индекса `HashMap` со словом, которое выступает в качестве ключа, и несколько объектов `SonnetRef`, содержащих номер сонета и номер строки в качестве значений.

```
// индексный список
private HashMap<String, HashSet<SonnetRef>> terminindex;
...
// добавление термина в индексный список
HashSet<SonnetRef> set = null;
if(index.containsKey(word)) {
    set = index.get(word);
} else set = new HashSet<SonnetRef>();

set.add(new SonnetRef(sons.size() - 1, i));
...
```

Для обработки нескольких терминов требуется логика, получающая каждый объект `SonnetRef` и выполняющая операцию пересечения, чтобы найти номера сонетов, в которых заданные термины встречаются вместе. В итоге возвращается результат данного пересечения.

```
public SonnetFragment[] search(String query) {
    if(query == null || query.trim().length() < 1)
        return new SonnetFragment[0];
    query = query.trim().toLowerCase();
    ArrayList<SonnetFragment> frags = new ArrayList<SonnetFragment>();
    String[] terms = query.split(" ");

    if(terms == null) terms = new String[]{query};
    ArrayList<HashSet<SonnetRef>> sets =
        new ArrayList<HashSet<SonnetRef>>();
    for(String term: terms) {
        if(terminindex.containsKey(term)) {
            // получаем каждое множество SonnetRefs для каждого
            // из указанных терминов
            sets.add((HashSet<SonnetRef>) (terminindex.get(term).clone()));
        }
    }
    if(!sets.isEmpty()) {
        HashSet<SonnetRef> main = null;
        for(HashSet<SonnetRef> set: sets) {
            if(main == null) main = set;
            else {
                // здесь выполняется операция пересечения
                main.retainAll(set);
            }
        }
    }
}
```

```

        if(main != null && !main.isEmpty()) {
            Iterator<SonnetRef> it = main.iterator();
            while(it.hasNext()) {
                SonnetRef s = it.next();
                Sonnet son = sonnets[s.num];
                frags.add(new SonnetFragment(s.num, son.lines[s.line]));
            }
        }

        return frags.isEmpty()
            ? new SonnetFragment[0]
            : frags.toArray(new SonnetFragment[frags.size()]);
    }
}

```

Поисковая логика, подкрепленная базой данных. В этом варианте реализации поиск построен на использовании SQL-запроса к тексту сонетов, хранимых в базе данных. После считывания данных сонета каждый сонет добавляется в базу данных. Столбцы содержат ссылочный номер сонета, заголовок сонета, номер строки и саму строку.

```

private SonnetsSQLiteOpenHelper sql;
...
private static class SonnetsSQLiteOpenHelper extends SQLiteOpenHelper {
    @Override
    public void onCreate(SQLiteDatabase db) {
        // создаем таблицу
        sonnetdb = db;
        sonnetdb.execSQL("CREATE TABLE "+SONNETTABLE+" ("
            +BaseColumns._ID+" INTEGER, "+SONNETNUM+
            " INTEGER,
            "+SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID+
            " TEXT, "+SONNETSTR+" TEXT, "+LINENUM+" INTEGER, "+
            LINETXT+" TEXT);");
    }

    // запрограммируем возможность добавления строки сонета
    // в базу данных
    public long addSonnet(int id, int sonnetnum, String sonnetstr,
        int linenum, String line) {
        ContentValues initialValues = new ContentValues();
        initialValues.put(BaseColumns._ID, id);
        initialValues.put(SONNETNUM, sonnetnum);
        initialValues.put(
            SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID,
            ""+sonnetnum);
        initialValues.put(LINENUM, linenum);
        initialValues.put(SONNETSTR, sonnetstr);
        initialValues.put(LINETXT, line);

        return sonnetdb.insert(SONNETTABLE, null, initialValues);
    }
}

```

```

}
...
Sonnet sonnet = new Sonnet(num, ls.toArray(new String[size]));
if(sql != null) {
    for(int i=0;i<size;i++) {
        // добавляем каждую строку в базу данных
        sql.addSonnet(id++, sonnet.num, sonnet.title, i, sonnet.lines[i]);
    }
}
...

```

После этого поисковый запрос можно выполнить при помощи оператора LIKE:

```

public Cursor searchDB(String query, String[] columns) {
    query = query.toLowerCase();
    // Здесь указывается конкретный SQL-запрос.
    // В данном случае, к строке запроса применяется LIKE.
    String selection = LINETXT + " LIKE ?";
    String[] selectionArgs = new String[] { "%"+query+"%"};
    return query(selection, selectionArgs, columns, null);
}

// Именно здесь мы выполняем запрос и возвращаем курсор.
private Cursor query(String selection, String[] selectionArgs,
                    String[] columns, String sort) {
    SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
    builder.setTables(SONNETTABLE);
    Cursor cursor = builder.query(
        sql.getReadableDatabase(),
        columns,
        selection,
        selectionArgs,
        null,
        null,
        sort);

    if (cursor == null) {
        return null;
    } else if (!cursor.moveToFirst()) {
        cursor.close();
        return null;
    }
    return cursor;
}

```

Конфигурация, обеспечивающая поиск

Как только поисковая логика будет готова, нужно заняться поисковым фреймворком. Для начала следует создать *конфигурацию с возможностью поиска* (searchable configuration). Это XML-файл, который находится в каталоге `res/xml` и, как правило, называется `searchable.xml`. Конфигурация с возможностью поиска содержит специфические атрибуты, которые в конечном итоге становятся настройками объекта `SearchableInfo`, инстанцируемого системой.

XML-файл, который описывает конфигурацию, ориентированную на поиск, должен содержать в корневом узле элемент `searchable`, а также атрибут `android:label`.

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_label"
    >
</searchable>
```

Весь синтаксис конфигурации с возможностью поиска выглядит так:

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="string resource"
    android:hint="string resource"
    android:searchMode=["queryRewriteFromData" | "queryRewriteFromText"]
    android:searchButtonText="string resource"
    android:inputType="inputType"
    android:imeOptions="imeOptions"
    android:searchSuggestAuthority="string"
    android:searchSuggestPath="string"
    android:searchSuggestSelection="string"
    android:searchSuggestIntentAction="string"
    android:searchSuggestIntentData="string"
    android:searchSuggestThreshold="int"
    android:includeInGlobalSearch=["true" | "false"]
    android:searchSettingsDescription="string resource"
    android:queryAfterZeroResults=["true" | "false"]
    android:voiceSearchMode=["showVoiceSearchButton" | "launchWebSearch" |
        "launchRecognizer"]
    android:voiceLanguageModel=["free-form" | "web_search"]
    android:voicePromptText="string resource"
    android:voiceLanguage="string"
    android:voiceMaxResults="int"
    >
<actionkey
    android:keycode="KEYCODE"
    android:queryActionMsg="string"
    android:suggestActionMsg="string"
    android:suggestActionMsgColumn="string"/>
</searchable>
```

Подробнее о конфигурации с возможностью поиска можно прочитать в разделе **Search Configuration (Конфигурация поиска)** в руководстве для разработчиков Android (Android Developers Guide).

Поисковая активность

После определения конфигурации с возможностью поиска нужно создать активность, поддерживающую такую возможность. В итоге именно эта активность будет вызывать поисковую логику и выводить результаты. Система запускает данную активность, выдавая намерение с действием `ACTION_SEARCH`, когда поиск выполняется через поисковое диалоговое окно или поисковый виджет. Запрос содержится

в намерении в виде строки `SearchManager.QUERY`. Отсюда активность может вызывать поисковую логику со строкой запроса. Логика возвращает результаты, которые затем можно отобразить. В нашем примере строка запроса будет передаваться одному из методов поисковой логики — `search()`, в ответ на что будет возвращаться массив объектов `SonnetFragment`. После этого логика активности отобразит результаты пользователю.

На первом этапе мы должны объявить поисковую активность в файле описания и указать системе, что именно к данной активности должен быть направлен поисковый запрос. Это делается путем добавления действия `android.intent.action.SEARCH` к фильтрам намерений, причем так же задается конфигурация, поддерживающая поиск.

```
<application ... >
  <activity android:name=".searchdemo.SearchActivity" >
    <intent-filter>
      <action android:name="android.intent.action.SEARCH" />
    </intent-filter>
    <meta-data android:name="android.app.searchable"
      android:resource="@xml/searchable"/>
  </activity>
  ...
</application>
```

На втором этапе требуется решить, как именно будут отображаться результаты поиска. Рекомендуется выводить результаты в форме того или иного списка. В этом примере мы будем выводить результаты в списке `ListView`. Чтобы все было проще, реализуем эту активность как расширение `ListActivity`, поскольку искомая активность предоставляет стандартный макет с `ListView`. Для доступа к данным применяются вспомогательные методы `getListView()` и `setListAdapter()`.

```
public class SearchActivity extends ListActivity {

    // объект поисковой логики
    private SearchLogicInterface SEARCHLOGIC;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.search);

        // инициализируем поисковую логику
        initializeSearchLogic();

        // обрабатываем намерения при их наличии
        handleIntent();
    }

    private void handleIntent() {
        if(getIntent() != null) {
            Intent intent = getIntent();
```

```

        if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
            query =
                intent.getStringExtra(SearchManager.QUERY).toLowerCase();
            search(query);
        }
    }
}

private void search(String query) {
    // вызываем поисковую логику
    final SonnetFragment[] sfrags = SEARCHLOGIC.search(query);

    getListView().setVisibility(View.VISIBLE);
    getListView().addHeaderView(
        View.inflate(this, R.layout.searchheader, null), null, false);

    ArrayAdapter<SonnetFragment> arr =
        new ArrayAdapter<SonnetFragment>(this, R.layout.searchrow,
                                         sfrags);
    setListAdapter(arr);

    getListView().setOnItemClickListener(new OnItemClickListener() {
        public void onItemClick(AdapterView<?> adpt, View view,
                                int pos, long id) {
            // Делаем что-либо, как только пользователь щелкает
            // на конкретном фрагменте sonnetfragment в списке.
        }
    });
}
}

```

Очень важно, чтобы в этом коде обрабатывалось намерение `ACTION_SEARCH`, передаваемое системой активности. Как только дополнение строки `SearchManager.QUERY` (`getStringExtra`) извлекается из намерения, это дополнение используется в качестве ввода для поисковой логики. Потом возвращаются результаты обработки.

В данном случае массив объектов `SonnetFragment` помещается в `ArrayAdapter`. Этот `ArrayAdapter` задается в качестве адаптера списка (`List`). Соответственно здесь выводятся результаты.

На этом заканчивается базовая работа над интерфейсом. Далее рассмотрим компоненты пользовательского интерфейса, к которым обращается пользователь для выполнения поиска, — речь идет о поисковом диалоговом окне и поисковом виджете. Поисковый виджет нужно использовать, если вы пишете программу для устройств с версией Android 3.0 (Honeycomb/API 11 или выше).

Поисковое диалоговое окно

Поисковое диалоговое окно (`search dialog`) — это компонент пользовательского интерфейса, с помощью которого пользователь может вводить текст и выполнять поиск. Этот компонент при активации появляется в верхней части экрана. Систе-

ма Android контролирует все события, происходящие в поисковом диалоговом окне. Если пользователь вводит запрос, а потом отправляет его, то система направляет намерение ACTION_SEARCH к активности, указанной в файле описания (манифесте).

Чтобы поисковое диалоговое окно могло отправлять запросы к объявленной активности с поддержкой поиска, в элементе `<meta-data>` должен содержаться атрибут `android:value`, указывающий имя класса поисковой активности. Кроме того, в активности, которая обозначена в файле описания как окно для поиска (`search dialog`), должен присутствовать атрибут `android:name` со значением `"android.app.default_searchable"`.

```
<application ... >
    <activity
        android:label="@string/app_name"
        android:name=".searchdemo.MainActivity" >

        <meta-data android:name="android.app.default_searchable"
            android:value=".searchdemo.SearchActivity" />
    </activity>
</application>
```

Как только эта ссылка на метаданные будет на месте, поисковое диалоговое окно будет активизироваться в данной активности, если пользователь нажмет на устройстве кнопку **Search (Поиск)** — при наличии таковой — либо при вызове активностью метода `onSearchRequested()`.

```
public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);

        findViewById(R.id.search).setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                onSearchRequested(); // активизирует поисковое
                                   // диалоговое окно
            }
        });
    }
}
```

Поисковое диалоговое окно всплывает в верхней части экрана. Оно никак не влияет на стек активностей. Таким образом, когда оно появляется, не вызывается никаких методов жизненного цикла — `onPause()` и т. д. Эта активность просто теряет фокус ввода, уступая его поисковому диалоговому окну. Если пользователь отменяет поиск, нажав кнопку **Back (Назад)**, то поисковое диалоговое окно закрывается и активность вновь получает фокус ввода.

Поисковый виджет

Поисковый виджет (а именно класс `SearchView`) действует только в Android 3.0 (Honeycomb/API 11) и выше. Рекомендуется использовать `SearchView` на панели действий (`ActionBar`) как вид с действием в составе разворачивающегося меню, а не размещать поисковый виджет прямо в макете вашей активности. Чтобы поместить его в `ActionBar`, создайте собственный XML-файл с меню (в этом примере он называется `search_menu.xml`) со ссылкой на `android:actionViewClass="android.widget.SearchView"` в одном из элементов и поместите XML-файл в каталог `res/menu`.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_search"
        android:icon="@android:drawable/ic_menu_search"
        android:title="@string/search"
        android:showAsAction="ifRoom|withText"
        android:actionViewClass="android.widget.SearchView"
        />
</menu>
```

Как только у вас будет готов XML-файл меню, вы сможете залить его в метод `onCreateOptionsMenu()` вашей активности. Поставьте здесь ссылку на `SearchView` и задайте метод `setSearchableInfo()` с `SearchableInfo`. Так будет представлена конфигурация с возможностью поиска.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // заполняем меню с параметрами данными из XML-файла
    MenuInflater inflater = getMenuInflater();
    // заливаем search_menu.xml
    inflater.inflate(R.menu.search_menu, menu);

    // получаем SearchView и устанавливаем конфигурацию
    // с возможностью поиска
    SearchManager searchManager =
        (SearchManager) getSystemService(Context.SEARCH_SERVICE);
    SearchView searchView =
        (SearchView) menu.findItem(R.id.menu_search).getActionView();
    searchView.setSearchableInfo(
        searchManager.getSearchableInfo(getComponentName()));

    // Не уменьшаем виджет до пиктограммы;
    // по умолчанию он должен быть развернут.
    searchView.setIconifiedByDefault(false);
    // включаем кнопку отправки
    searchView.setSubmitButtonEnabled(true);
    // обеспечиваем возможность модификации запроса
    searchView.setQueryRefinementEnabled(true);

    return true;
}
```

Варианты завершения запроса

Когда пользователь вводит запрос в поисковое окно или поисковый виджет, то в интерфейсе можно предлагать варианты завершения этого запроса (query suggestions), чтобы пользователь мог из них выбрать. При этом следует реализовать три основных этапа.

1. Система передает текст запроса определенному поставщику содержимого, в котором находятся варианты завершения запроса.
2. Поставщик содержимого возвращает курсор, который указывает список всех вариантов завершения, в зависимости от конкретного запроса.
3. Система отображает варианты завершения.

Если пользователь выберет один из вариантов, то поисковой активности будет направлено намерение со специальным действием и данными.

API предоставляют удобные средства для отображения вариантов на основании истории запросов (то есть API ориентируется при предложении вариантов на недавние запросы), а также средства для создания специальных вариантов (генерируемых логикой приложения).

Варианты завершения, основанные на недавних запросах

Класс `SearchRecentSuggestionsProvider` представляет собой основу для поставщика содержимого, хранящего историю поисковых запросов. Именно он реализует большую часть логики, необходимой для возврата верных результатов. Благодаря этому снижается объем кода и сложность конфигурации, требуемые для реализации системы по предложению вариантов завершения запросов.

Чтобы все это сделать, создайте класс `SearchRecentSuggestionsProvider` и определите в нем источник и режим, вызвав метод `setupSuggestions()`.

```
package com.oreilly.demo.android.pa.searchdemo;

import android.content.SearchRecentSuggestionsProvider;

public class CustomSearchSuggestionProvider extends
    SearchRecentSuggestionsProvider {
    public final static String AUTHORITY =
        "com.oreilly.demo.android.pa.searchdemo.CustomSearchSuggestionProvider";

    public final static int MODE = DATABASE_MODE_QUERIES;

    public CustomSearchSuggestionProvider() {
        super();
        setupSuggestions(AUTHORITY, MODE);
    }
}
```

После того как поставщик содержимого создан, он должен быть объявлен в файле описания. Самое важное в данном случае — сослаться именно на ту строку

источника, которая определена в специальном поставщике содержимого SearchRecentSuggestionsProvider.

```
<application>
    ...
    <provider android:name=".searchdemo.CustomSearchSuggestionProvider"
        android:authorities=
"com.oreilly.demo.android.pa.searchdemo.CustomSearchSuggestionProvider"
    />
    ...
</application>
```

Кроме того, на источник нужно сослаться в XML-файле, обеспечивающем поисковую конфигурацию, через атрибут `android:searchSuggestAuthority`. Вы также должны определить `android:searchSuggestSelection` со значением " ?" (не забудьте пробел перед знаком вопроса), передав запрос как аргумент выбора SQLite.

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_name"
    android:hint="@string/search_hint"
    android:voiceSearchMode="showVoiceSearchButton|launchRecognizer"
    android:searchSuggestAuthority=
"com.oreilly.demo.android.pa.searchdemo.CustomSearchSuggestionProvider"
    android:searchSuggestSelection=" ?"
>
</searchable>
```

Наконец теперь, когда `SearchRecentSuggestionsProvider` запрограммирован, объявлен и сконфигурирован, нужно добавить запросы к истории поиска. Это делается путем применения метода `saveRecentQuery()` к `SearchRecentSuggestions` с передачей запроса этому методу.

Чтобы очистить историю вызовов, сделайте `clearHistory()`.

```
SearchRecentSuggestions suggestions =
    new SearchRecentSuggestions(this,
        CustomSearchSuggestionProvider.AUTHORITY,
        CustomSearchSuggestionProvider.MODE);

suggestions.saveRecentQuery(query, null);    // сохраняем запрос

suggestions.clearHistory();                  // очищаем историю
```

Специальные варианты завершения

Чтобы создавать специальные варианты завершения запросов, формируемые на основании логики приложения, нужно попытаться задействовать поставщик содержимого, который получает запрос и возвращает курсор (`Cursor`). Информация, на которую ссылается курсор, представляет собой строки с вариантами завершения, которые система отображает пользователю. В данном случае система ожидает, что данные будут распределены по заранее заданным столбцам, как в табл. 14.1. Некоторые из них обязательны, другие — опциональны.

Таблица 14.1. Данные курсора для специальных вариантов завершения запроса

Столбец	Базовый класс	Значение	Необходим?
_ID	android.provider.BaseColumns	Уникальный целочисленный идентификатор для данной строки	Да
SUGGEST_COLUMN_TEXT_1	android.app.Search Manager	Строка с вариантом завершения	Да
SUGGEST_COLUMN_TEXT_2	android.app.Search Manager	Вторая строка текста, которую требуется отобразить	Нет
SUGGEST_COLUMN_ICON_1	android.app.Search Manager	Отрисовываемый ресурс или URI на файл пиктограммы, которая будет отображаться слева	Нет
SUGGEST_COLUMN_ICON_2	android.app.Search Manager	Отрисовываемый ресурс или URI на файл пиктограммы, которая будет отображаться справа	Нет
SUGGEST_COLUMN_INTENT_ACTION	android.app.Search Manager	Указывает действие, записанное в намерении для данного варианта. Информация должна быть получена из значения элемента android:searchSuggestIntentAction в поисковой конфигурации	Нет
SUGGEST_COLUMN_INTENT_DATA	android.app.Search Manager	Указывает информацию намерения для данного варианта завершения. Информация должна быть получена из значения элемента android:searchSuggestIntentData в поисковой конфигурации	Нет
SUGGEST_COLUMN_INTENT_DATA_ID	android.app.Search Manager	Строка пути URI, прикрепляемая к полю данных в намерении	Нет
SUGGEST_COLUMN_INTENT_EXTRA_DATA	android.app.Search Manager	Дополнительные данные, записываемые в ключе EXTRA_DATA_KEY, относящиеся к намерению	Нет
SUGGEST_COLUMN_QUERY	android.app.Search Manager	Данные, добавляемые в ключ QUERY, который относится к намерению	Нет
SUGGEST_COLUMN_SHORTCUT_ID	android.app.Search Manager	Используется только в поле для быстрого поиска. Это идентификатор, на который мы ссылаемся при необходимости сохранения сокращенного варианта завершающей строки	Нет
SUGGEST_COLUMN_SPINNER_WHILE_REFRESHING	android.app.Search Manager	Используется только в поле для быстрого поиска. Это отрисовываемый ресурс или URI файла, применяемого для отображения пиктограммы справа. Краткий вариант именно этого завершения, а не пиктограмма из столбца SUGGEST_COLUMN_ICON_2 обновляется в поле быстрого поиска	Нет

Важно отметить, что, если ваши варианты завершения запросов сохранены не в табличном формате (примером такого формата является, в частности, таблица SQLite), вам потребуется возможность оперативно привести эти данные в соответствие со столбцами, с которыми работает система. Для этого вы должны создать курсор `MatrixCursor` с необходимыми столбцами и добавить данные при помощи метода `addRow(Object[])`.

Независимо от специфики курсора метод `query()` поставщика содержимого должен возвращать курсор так, как это определено выше. Метод `query()` определяется следующим образом:

```
public Cursor query(Uri uri, String[] projection, String selection,
                   String[] selectionArgs, String sortOrder)
```

Этот метод получает от системы следующие параметры:

- `uri` — объясняется ниже в списке;
- `projection` — всегда равен нулю;
- `selection` — значение `android:searchSuggestSelection` в конфигурации с поддержкой поиска;
- `selectionArgs` — содержит поисковый запрос как единственный элемент массива, если вы объявили в поисковой конфигурации атрибут `android:searchSuggestSelection`, в противном случае — равен нулю. В сущности, это и есть термин (или термины), который ищет пользователь;
- `sortOrder` — всегда равен нулю.

Формат `uri` таков:

```
content://your.authority/optional.suggest.path/some_uri_path_that_you_defined/query
```

`your.authority` — это определенный выше атрибут `android:searchSuggestAuthority`, находящийся в XML-файле с поисковой конфигурацией. `optional.suggest.path` — это опциональный атрибут `android:searchSuggestPath`, находящийся в том же файле. Таким образом, вы сможете различать активности, способные быть потенциальными источниками запроса, если у одного и того же поставщика содержимого в вашей системе будет несколько активностей с поддержкой поиска. Элемент `some_uri_path_that_you_defined` — это определяемая вами константная строка, а `query` — сама строка запроса, задаваемого пользователем. Эта строка запроса может кодироваться URI — таким образом, может понадобиться и декодирование.

Ниже приведен пример метода `query()`.

```
public class SearchDBProvider extends ContentProvider {
    ...
    public final static String AUTHORITY =
        "com.oreilly.demo.android.pa.searchdemo.SearchDBProvider";
    ...
    // создаем сопоставитель uri (UriMatcher) на основе AUTHORITY
    private static final UriMatcher matcher = buildUriMatcher();
    ...
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
```

```

        String[] selectionArgs, String sortOrder) {
    switch (matcher.match(uri)) {
        // приходим к выводу, что это вызов с вариантом завершения
        case SEARCH_SUGGEST:
            return getSuggestions(selectionArgs[0]);
        ...
    }
}

private Cursor getSuggestions(String query) {
    query = query.toLowerCase();
    String[] columns = new String[] {
        BaseColumns._ID,
        SearchDBLogic.SONNETNUM,
        SearchDBLogic.LINETXT,
        SearchManager.SUGGEST_COLUMN_INTENT_DATA_ID
    };

    // совершаем поиск в соответствии с поисковой
    // логикой, чтобы разобраться с вариантами завершения
    return ((SearchDBLogic) SearchActivity.SEARCHLOGIC).
        searchDB(query, columns);
}
...
}

```

Кроме того, укажите в файле описания ваш поставщик содержимого:

```

<application>
    ...
    <provider android:name=".searchdemo.SearchDBProvider"
        android:authorities=
            "com.oreilly.demo.android.pa.searchdemo.SearchDBProvider"
    />
    ...
</application>

```

После этого оформите вашу поисковую конфигурацию как минимум с одним элементом `android:searchSuggestAuthority`, задав в качестве значения этого элемента определенную выше строку источника.

```

<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_name"
    android:searchSuggestAuthority=
        "com.oreilly.demo.android.pa.searchdemo.SearchDBProvider"
    >
</searchable>

```

Другая информация, например специальное намерение, также может указываться в поисковой конфигурации, для этого применяется `android:searchSuggestIntentAction`. При указании специального намерения вы позволяете поисковой

активности различать обычный поисковый запрос (идущий с намерением `android.intent.action.SEARCH`) и предложенный системой поисковый запрос. В следующем примере специальное намерение используется с `android.intent.action.VIEW`.

```
<?xml version="1.0" encoding="utf-8"?>
    <searchable xmlns:android=
        "http://schemas.android.com/apk/res/android"
        android:label="@string/app_name"
        android:searchSuggestAuthority=
            "com.oreilly.demo.android.pa.searchdemo.SearchDBProvider"
        android:searchSuggestIntentAction="android.intent.action.VIEW"
    >
</searchable>
```

В данном случае поисковая активность должна обрабатывать и `Intent.ACTION_VIEW`, и `Intent.ACTION_SEARCH`.

```
Intent intent = getIntent();
```

```
if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
    String query = intent.getStringExtra(SearchManager.QUERY).toLowerCase();
    search(query); // выполняем поиск в соответствии с поисковой логикой
} else if (Intent.ACTION_VIEW.equals(intent.getAction())) {
    Uri data = intent.getData();
    loadData(data); // делаем что-либо с этими данными
}
```

Наконец, если вы хотите предоставить варианты завершения запросов для использования в окне быстрого поиска, элемент `android:includeInGlobalSearch` в поисковой конфигурации должен быть установлен в `true`. Варианты завершения, выбираемые пользователем в окне быстрого поиска, могут автоматически предоставляться системой для быстрого доступа. Эти варианты кэшируются системой и взяты у вашего поставщика содержимого, поэтому к ним можно быстро получить доступ. Как только вы открываете в вашем приложении доступ к вариантам завершения запроса в окне быстрого поиска, система ранжирования определяет, как варианты завершения конкретного запроса будут предоставляться пользователю. Это может зависеть от того, у какого количества других приложений есть результаты этого запроса и как часто пользователь выбирает ваш результат по сравнению с таким результатом из других приложений. Невозможно с точностью сказать, как именно будут ранжироваться ваши варианты завершения и будут ли вообще ваши варианты предлагаться для завершения конкретного запроса. Вообще, при предоставлении качественных результатов запроса более вероятно, что именно варианты из вашего приложения окажутся на высоких позициях. И наоборот: если приложение предлагает неточные варианты завершения запроса, они ранжируются ниже или вовсе не отображаются.

15 Геолокация и картография

С тех пор как в мобильные телефоны начали встраивать автономные GPS-приемники, разработчики предвидели наступление новой эпохи геолокационных (location-based) приложений. Функция распознавания местоположения (location awareness) позволяет создавать мобильные приложения нового поколения. Например, если ваша программа подыскивает для пользователя интересные его рестораны, то явным преимуществом было бы ограничить поиск областью, которая находится в зоне досягаемости того или иного пользователя. Еще лучше, если вы сможете посмотреть на карте, где находятся рестораны, а также если программа покажет, как туда лучше добраться на машине или пешком. Если вы ищете дополнительный заработок, как в программе **MJAndroid**, рассмотренной нами в разделе «API базы данных на примере MJAndroid» главы 9, то было бы очень хорошо графически отметить на карте места, в которых предполагается работать.

Навигационные функции — типичная черта лишь первого поколения геолокационных сервисов (LBS). Приложения, позволяющие пользователям, например, делиться с друзьями информацией о собственном местоположении, как Google Latitude, либо присваивать географическим точкам различные степени важности, как Foursquare, стали появляться как грибы после дождя. Геолокационные сервисы, без преувеличения, находятся на взлете, и, как вы увидите, система Google Android предлагает обширные возможности, значительно упрощающие разработку приложений такого типа.

С экономической точки зрения геолокационные приложения — это серьезный фактор в мобильной телефонии. Значительная часть прибылей от продажи мобильных приложений относится к программам именно этого класса, кроме того, их доля на рынке продолжает быстро расти. Поскольку такие приложения опираются на способность мобильной сети находить устройства, в ней работающие, и работа этих программ основана на взаимодействии мобильности с определением местоположения, геолокационные приложения имеют для мобильной телефонии не менее фундаментальное значение, чем сама связь.

Часто в приложениях функция распознавания местоположения комбинируется с поисковой функцией. Где мои контакты? Где товары или услуги, которые я ищу? Где находятся люди, с которыми у нас есть общие интересы?

В этой главе мы поговорим о том, как программа **MJAndroid** использует систему Android для получения ответов на эти вопросы.

Геолокационные сервисы

На мобильных телефонах по отдельности или в комбинациях используются взаимосвязанные методы, позволяющие узнать, где находится телефон.

- Идентификатор соты (Cell ID) — независимо от того, говорите вы с кем-нибудь или нет, ваш мобильный телефон, пока он включен, постоянно «общается» с ближайшими сотовыми вышками. Аппарат делает это для того, чтобы была возможность немедленно ответить на вызов, если кто-то вам позвонит. Поэтому каждые несколько секунд телефон «пингует» сотовую вышку, с которой он связывался в последний раз, чтобы сообщить, что он все еще находится в зоне ее действия, и чтобы записать сетевые параметры — в частности, точное время, актуальную силу сигнала (восходящего и нисходящего) и т. д.

Если вы перемещаетесь, то ваш телефон может инициировать передачу обслуживания (так называемый «хэндовер») на другую сотовую вышку. Этот процесс протекает в фоновом режиме и не требует вашего участия. Каждая сотовая вышка в мире имеет уникальный идентификатор, называемый, что логично, идентификатором соты. Кроме того, каждой сотовой вышке известна широта и долгота, под которыми она расположена. Поэтому мобильному телефону достаточно просто приблизительно узнать о том, где вы находитесь. Для этого телефону требуется отметить географическое расположение идентификатора соты, в которой он в данный момент находится. Размеры сот в сотовых сетях варьируются в зависимости от того, насколько активный сетевой трафик ожидается в конкретном районе. В США, например, одна сота имеет радиус от 800 метров (в городах) до восьми километров и более (открытые пространства).

- Триангуляция — большую часть времени сотовый телефон находится в зоне действия более чем одной сотовой вышки. В мобильных технологиях 2G и последующих поколений сотовая вышка умеет определить, с какого направления приходит сигнал. Если телефон находится в зоне действия двух или трех сотовых вышек, то вместе они могут выполнять триангуляцию местоположения телефона. Затем в сетях некоторых операторов телефон может запрашивать у сети информацию о том, где он находится. Это звучит немного старомодно, но такая техника определения местоположения может быть очень точной и не требует устанавливать на сотовом телефоне какое-либо дополнительное оборудование.
- GPS — спутниковая система глобального позиционирования (Global Positioning System, GPS) в наши дни является вездесущей. Она работает и в навигационных системах автомобилей, и в портативных навигаторах, и в мобильных телефонах. Приятно отметить, что при использовании GPS на вашем мобильном телефоне его местоположение можно определить очень точно. В частности, можно определить высоту над уровнем моря, что важно для некоторых приложений. У GPS есть свои недостатки, но тем не менее популярность этой системы только растет. А недостатки ее таковы:
 - дополнительная стоимость — GPS-радио и процессоры стоят совсем недорого, но все же, если общая стоимость компонентов мобильного телефона увеличится хотя бы на десять долларов, это будет довольно ощутимо;

- сокращение срока службы аккумулятора — уже достигнуты замечательные прорывы в снижении энергоемкости радиоустройств и процессоров GPS, но они по-прежнему сильно расходуют заряд батареи. На большинстве телефонов, оснащенных функцией GPS, пользователь также может включать и отключать эту функцию. Если работа вашего приложения зависит от точности глобального позиционирования, то стоит учитывать, что приложение само должно проверять, включена ли на устройстве система GPS, и уведомлять пользователя, если эта система отключена;
- недостаточная безотказность — порой отказывают любые системы, но функционирование GPS особенно зависит от того, сможет ли мобильное устройство связываться со спутниками, которые летают на орбите. Если вы находитесь в подвале высотного здания и вокруг вас — сплошной железобетон, то вы, пожалуй, не сможете пользоваться GPS.

Можно не сомневаться, что на всех телефонах Android будет использоваться один или несколько из перечисленных методов геолокации. В частности, на самых новых телефонах с Android применяются они все. Итак, поговорим подробнее об использовании геолокационных возможностей.

Работа с картами

Самым прославленным продуктом Google является его поисковик, но немногим уступают ему в известности и знаменитые карты. Работая над созданием Android, сотрудники Google без труда разглядели потенциал геолокационных сервисов, а также поняли, как хорошо они впишутся в систему для работы с географическими картами. Большинство геолокационных сервисов в конечном счете отображают карту. У Google давно уже была в арсенале технология отображения и обновления интерактивных карт, а также бизнес-процессы, позволявшие другим пользоваться этими картами и добавлять картографические функции на собственные сайты. По-прежнему требуется совершить значительный прорыв, чтобы картографические функции стали доступны для разработчиков мобильных приложений, но Google, несомненно, уже взялась за решение этой проблемы в Android.

Активность для работы с картами Google

В числе приложений, предлагаемых вместе с операционной системой Android, есть и программа Google Maps. Если приложение допускает работу с картами, то программу Google Maps можно запустить из него так же, как и любую другую активность.

1. Создать намерение Intent (`new Intent(String action, Uri uri)`), сообщающее, что вам требуется отобразить карту. Необходимые параметры таковы:
 - действие (`action`), для которого следует указать `ACTION_VIEW`;
 - уникальный идентификатор ресурса (`Uri`), для которого нужно задать одну из трех следующих схем, вставив в нее свои данные:

```

geo: latitude, longitude
geo: latitude, longitude ?z= zoom
geo: 0,0?q my_street_address
geo: 0,0?q business_near_city

```

2. Вызвать `startActivity(Intent intent)`, воспользовавшись только что созданным намерением.

Вот пример, в котором создается карта:

```

Intent intent = new Intent(ACTION_VIEW, "geo:37.422006,-122.084095");
startActivity(intent);

```

Это совсем не сложно, а в вашем распоряжении оказывается весь потенциал карт Google, однако таким способом нельзя по-настоящему интегрировать карту в ваше приложение. **Google Maps — это самостоятельное приложение, и вы не сможете изменить его пользовательский интерфейс или наложить на карту дополнительный графический слой, где можно было бы указать объекты, интересующие ваших пользователей.** Для работы с картографическими возможностями Android предлагает более удобные пакеты.

MapView и MapActivity

В приложение MJAndroid, которое мы разрабатывали в главе 9, требуется добавить слои, на которых будут отображаться местоположения организаций, находящихся поблизости от вас и предлагающих дополнительный заработок. Итак, мы не будем работать с картами Google, а вместо этого воспользуемся видом MapView, который можем наложить на карту и снабдить любой нужной нам графикой. На каждую активность может приходиться только один MapView, и данная активность должна дополнять MapActivity. Как видите, можно «малой кровью» приобрести мощные географические функции, которыми вид MapView обогащает ваше приложение.

Для работы с MapView требуется выполнить пару важнейших предпосылок, о которых мы вкратце упоминали, когда обсуждали инициализацию MJAndroid в главе 9.

- Включить в программу библиотеку, в которую входит MapView.

MapView не входит в состав библиотек Android, которые устанавливаются по умолчанию. Для работы с MapView вам потребуется указать в файле `AndroidManifest.xml`, что вы пользуетесь дополнительной библиотекой:

```

<application android:icon="@drawable/icon2">
    <uses-library android:name="com.google.android.maps" />

```

Строку `uses-library` можно вставить не в любом месте файла `AndroidManifest.xml`. Она должна быть в определении тега `<application>` и вне определения тега `<activity>`.

- Подписывание приложения и получение ключа к картографическому API от Google. Если вы используете в своем приложении MapView, то фактически для рисования карты в вашей программе применяются данные, полученные от карт

Google. По юридическим причинам Google обязана отслеживать, кто пользуется картографическими данными. Google не касается то, что именно за операции с картами осуществляет ваше приложение, но вы обязаны зарегистрироваться в Google для получения ключа к API и согласиться с соответствующими условиями использования. Таким образом, вы сообщаете Google о том, что применяете картографические данные, а также информацию о маршрутах, предоставляемую сервисом Google Maps. В разделе «Подписывание приложения» главы 4 было рассказано, как подписать приложение и получить ключ к API.



Не забывайте, что программы, использующие MapView, требуется подписывать. Чтобы вы могли без проблем опробовать пример с программой MJAndroid из этой книги, мы предлагаем именно такой файл APK, который описан в разделе «Подписывание приложения» главы 4.

Работа с MapView

В MapView заключено довольно много сложного картографического кода, вы можете совершенно бесплатно пользоваться этим видом в своих приложениях, которые пишете для Android. Ниже перечислены некоторые вещи, которые можно делать при помощи MapView, причем для этого требуется выполнить минимальный объем собственного программирования.

- Отобразить карту улиц любого района в мире, с новейшей картографической информацией, предоставляемой Google.
- Изменить вид отображения карты, чтобы предоставить:
 - просмотр изображений улиц — фотографии, сделанные на уровне едущего автомобиля;
 - вид со спутника — фотографию местности, полученную путем аэрофото-съемки;
 - вид трафика — информацию о загруженности дорог, обновляемую в реальном времени и наложенную на карту с видом со спутника.
- Перемещать карту под управлением программы.
- Наносить собственную графику на слои, накладываемых на карту.
- Откликаться на события, возникающие, когда пользователь касается карты на экране.

Инициализация MapView и MapLocationOverlay

Карта в MicroJobs предусматривает два режима работы.

- При запуске и когда мы выбираем Current Location (Актуальное местоположение) в блоке с изменяемым значением (Spinner), мы хотим отобразить карту того места, в котором находимся в данный момент, а также хотим, чтобы карта

изменялась по мере того, как мы движемся. Для такой карты будем использовать класс `MyLocationOverlay`.

- При выборе конкретного местоположения из раскрывающегося списка мы хотим отобразить карту этого места, отключить геолокационные обновления и не отслеживать движение.

Рассмотрим в классе `MicroJobs.java` код, который занимается инициализацией `MapView`, а также изучим класс `MyLocationOverlay`, отслеживающий наше актуальное местоположение:

```
@Override
public void onCreate(Bundle savedInstanceState) {

    // код опущен...
    mvMap = (MapView) findViewById(R.id.mapmain); ❶

    // получение контроллера карты
    final MapController mc = mvMap.getController(); ❷

    mMyLocationOverlay = new MyLocationOverlay(this, mvMap); ❸
    mMyLocationOverlay.enableMyLocation();
    mMyLocationOverlay.runOnFirstFix( ❹
        new Runnable() {
            @Override
            public void run() {
                mc.animateTo(mMyLocationOverlay.getMyLocation()); ❺
                mc.setZoom(16);
            }
        });

    Drawable marker = getResources().getDrawable(
        R.drawable.android_tiny_image); ❻
    marker.setBounds(0, 0, marker.getIntrinsicWidth(),
        marker.getIntrinsicHeight());
    mvMap.getOverlays().add(new MJJobsOverlay(marker));

    mvMap.setClickable(true); ❼
    mvMap.setEnabled(true);
    mvMap.setSatellite(false);
    mvMap.setTraffic(false);

    // начинаем работу с обычным масштабом
    mc.setZoom(16); ❽

    // код опущен...
}

/** необходим метод, указывающий, будем ли мы отображать маршруты */
@Override
protected boolean isRouteDisplayed() { return false; } ❾
```

Рассмотрим пояснения к коду.

- ❶ Сначала находим `MapView` в файле компоновки `main.xml` так же, как находим и любой другой вид, и присваиваем его переменной `mvMap` типа `MapView`, чтобы при необходимости на этот вид можно было сослаться.
- ❷ Получаем описатель для `MapController`, связанного с `MapView`. Его будем использовать для панорамирования (анимирования) карты, увеличения, уменьшения, перехода от одного вида к другому и т. д.
- ❸ Для работы с `MyLocationOverlay` создаем новый экземпляр и даем ему имя `mMyLocationOverlay`.
- ❹ Работу с `mMyLocationOverlay` мы начинаем с того, что определяем метод, который Android вызовет, когда мы получим наше первое изменение данных геолокации от поставщика геолокационной информации.
- ❺ Метод `runOnFirstFix` перемещает карту в точку нашего актуального местоположения (оно задается при помощи `mMyLocationOverlay.getLocation()`) и ставит такую степень увеличения, которой будет достаточно, чтобы мы могли рассмотреть, какие вакансии предлагаются поблизости.
- ❻ Идентифицируем маркер, который решили применить с `mMyLocationOverlay` в качестве отметки для имеющихся вакансий. Мы воспользуемся изображением, сохраненным в нашем каталоге `res/drawable`, это изображение называется `android_tiny_image`. Здесь мы видим полюбившегося нам зеленого робота. Мы определяем границы `Drawable` и добавляем накладываемый слой с этими маркерами в список других слоев, заданных для `MapView.mvMap`.
- ❼ Теперь стоит задать несколько исходных атрибутов для `mvMap`, об этом мы поговорим ниже в данном разделе. Большинство из этих атрибутов пользователь может изменить при помощи кнопок меню.
- ❽ Затем, следуя философии «убежденных перестраховщиков», просто на случай, если у нас в распоряжении вдруг не окажется поставщика геолокационной информации, который запустит `runOnFirstFix`, вновь задаем здесь уровень увеличения.
- ❾ Наконец, `MapView` требует от нас переопределить метод `isRouteDisplayed()`, чтобы указать, будем ли мы отображать на нашей карте информацию о маршрутах. В нашем приложении мы этого делать не собираемся, поэтому здесь возвращается `false`.

В `MyLocationOverlay` заключена масса кода для работы с геолокацией и картографированием. В ходе единственного вызова к конструктору мы выполняем следующее.

- Приказываем Android определить, какие поставщики геолокации доступны вокруг нас (GPS, идентификатор соты, триангуляция).
- Подключаемся к лучшим из этих поставщиков.
- Приказываем поставщику геолокации периодически предоставлять нам обновления информации о местоположении, по мере того как телефон будет двигаться вместе с нами.

- Подключаемся к служебным подпрограммам, которые будут автоматически перемещать нашу карту так, чтобы отражать все происходящие изменения местоположения.

`MyLocationOverlay` также позволяет нам разместить в `MapView` и картушку компаса, и ее тоже можно было бы обновлять, но в программе **MJAndroid** мы этой функцией пользоваться не будем.

Картографические атрибуты, задаваемые в коде, таковы.

- `setClickable` — мы хотим, чтобы пользователь мог коснуться пальцем определенной вакансии на экране и получить подробную информацию о данной работе. Поэтому для этого атрибута мы зададим значение `true`.
- `setEnabled` — наследуется от `android.view.View`. Google не дает точного ответа на вопрос, что это означает в случае с `MapView`, но предположительно атрибут обеспечивает работу стандартных картографических функций — увеличения и уменьшения, панорамирования и т. д.
- `setSatellite` — выставив такой флаг, мы добавляем вид со спутника со сводной карты. Если не поставить этот индикатор, то вид со спутника удаляется. Попробуем обойтись на карте без спутниковой информации.
- `setTraffic` — аналогично, выставив или убрав этот флаг, мы добавляем на карту актуальную информацию о дорожном трафике или соответственно удаляем ее. Опять же попробуем начать работать с картой без указания такой информации.

Изменение масштаба на картах Android

Карты Android оснащены поддержкой увеличения и уменьшения масштаба. Клавиша **I** позволяет сделать масштаб на карте крупнее, а клавиша **O** — мельче. Кроме того, изменения масштаба на карте могут управляться программой при помощи контроллера `MapController`.

Для работы с масштабом определено несколько методов, все они используют `MapController`. На картах Android предусмотрен 21 уровень масштабирования. На уровне масштабирования 1 экватор Земли имеет длину 256 пикселей. На каждом последующем уровне масштабирования соответствующая величина удваивается. Google предупреждает, что для некоторых регионов мира отсутствуют карты с наивысшим разрешением. Все методы масштабирования ограничивают уровни масштабирования диапазоном от 1 до 21, даже если вы требуете выйти за границы этого диапазона.

Существуют следующие методы для управления масштабированием:

- `zoomIn` — увеличение на один уровень;
- `zoomOut` — уменьшение на один уровень;
- `setZoom(int zoomlevel)` — масштабирование до заданного уровня в диапазоне от 1 до 21;
- `zoomInFixing(int xpixel, int ypixel)`, `zoomOutFixing(int xpixel, int ypixel)` — увеличение на один уровень. При этом заданная точка сохраняет фиксир-

- рованное место на экране. Обычно при масштабировании в сторону увеличения или уменьшения, единственная фиксированная точка — это центр экрана. Данные подпрограммы позволяют сделать фиксированной любую точку на карте;
- `zoomToSpan(int latSpanE6, int longSpanE6)` — программа пытается подобрать такой масштаб, чтобы заданный участок карты целиком умещался на экране. Фактически программа выбирает уровень увеличения, который ближе всего соответствует запрошенному участку. Параметры широты и долготы данного участка являются целыми числами, их значение в 106 раз превышает истинное значение в градусах. Например, участок в 2,5° широты на 1° долготы будет выражен как `zoomToSpan(2500000, 1000000)`.

Приостановление и возобновление работы MapActivity

Давайте ненадолго сосредоточимся на картографических активностях и подумаем, как нам сэкономить энергию батареи. Оказывается, в Android это довольно просто.

На мобильных платформах срок действия батареи — это все и вся, и мы, в отличие от приложения, отображаемого в данный момент, хотим сделать все возможное, чтобы наша программа потребляла минимум энергии. Вспомните наш разговор о жизненном цикле Android (раздел «Визуализация жизненных циклов» главы 10). Тогда мы говорили о том, что, когда одна активность (например, `MicroJobs`) запускает другую активность (например, `MicroJobsList`), новая активность занимает экран, а активность, которая делала вызов, смещается в стек активностей, которые ожидают запуска. В этот момент Android вызывает подпрограмму `onPause` в той активности, которая делала вызов, чтобы эта активность могла подготовиться к гибернации. Тогда в `MicroJobs.java` (или в любой `MapActivity`, использующей геолокационные обновления) нам потребуется отключить эти обновления. Таким образом мы хотя бы сэкономим циклы, которые потратились бы на обновление, и можем позволить телефону потратить еще меньше энергии, если переведем поставщик геолокации в состояние покоя, в котором расходуется совсем мало ресурсов.

Когда вызванная активность (в нашем случае — `MicroJobsList`) завершает работу, а активность, делавшая вызов, поднимается вверх из стека и занимает экран, фреймворк вызывает метод `onResume` в вызывающей активности. В `MapActivity` при активации этого метода мы хотим вновь включить обновления информации о местоположении.

В `MicroJobs` методы `onPause` и `onResume` очень просты:

```
/**
 * @see com.google.android.maps.MapActivity#onPause()
 */
@Override
public void onPause() {
    super.onPause();
```

```

        mMyLocationOverlay.disableMyLocation();
    }

    /**
     * @see com.google.android.maps.MapActivity#onResume()
     */
    @Override
    public void onResume() {
        super.onResume();
        mMyLocationOverlay.enableMyLocation();
    }

```

Обратите внимание: если бы в состав нашего `MyLocationOverlay` входила картушка компаса, ее тоже потребовалось бы сначала отключить, а потом снова активизировать. Иначе система впустую тратила бы циклы и заряд батареи, обновляя направление на картушке компаса, которая не видна на экране.

Управление картой при помощи клавиш меню

Мы хотим дать пользователю возможность включать вид со спутника, отображение дорожного движения и панорамы улиц. Кроме того, добавим в меню еще несколько кнопок, которые будут отвечать за масштабирование или, например, еще какой-нибудь способ перейти к списку вакансий.

В Android имеется богатейший набор возможностей для работы с меню, в том числе три типа меню: список параметров, контекстное меню и подменю. Все эти меню обладают собственными возможностями, кнопками с пиктограммами и прочими продвинутыми возможностями. Мы ограничимся использованием обычных кнопок меню с текстовыми надписями (как выполняли это выше, в разделе «Меню и панель действий» главы 6). Здесь нам потребуется сделать две вещи:

- создать меню с кнопками, которое будет отображаться;
- отслеживать события меню и в ответ на них запускать соответствующие действия.

Следующий код создает меню для `MicroJobs.java`:

```

/**
 * задаем меню для этой страницы
 *
 * @see android.app.Activity#onCreateOptionsMenu(android.view.Menu)
 */
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    boolean supRetVal = super.onCreateOptionsMenu(menu);
    menu.add(Menu.NONE, 0, Menu.NONE, getString(R.string.map_menu_zoom_in));
    menu.add(Menu.NONE, 1, Menu.NONE, getString(R.string.map_menu_zoom_out));
    menu.add(Menu.NONE, 2, Menu.NONE, getString(R.string.map_menu_set_satellite));
    menu.add(Menu.NONE, 3, Menu.NONE, getString(R.string.map_menu_set_map));
    menu.add(Menu.NONE, 4, Menu.NONE, getString(R.string.map_menu_set_traffic));
    menu.add(Menu.NONE, 5, Menu.NONE, getString(R.string.map_menu_show_list));
}

```

```
    return supRetVal;
}
```

Мы создаем кнопки меню, переопределяя метод `onCreateOptionsMenu`, в котором получили параметр меню для меню активности. После того как мы исправно предоставили суперклассу возможность сделать то, что он должен сделать, просто добавляем в меню элементы (кнопки) при помощи `menu.add`. Версия `menu.add`, которую мы выбрали, принимает четыре параметра.

- `int groupid` — Android позволяет группировать элементы меню так, чтобы можно было быстро изменить все меню целиком. В `MicroJobs` нам такая возможность не требуется — об этом мы сообщаем при помощи `Menu.NONE`.
- `int itemid` — нам нужен уникальный идентификатор для данного элемента меню, чтобы позже мы могли отслеживать, был ли выбран этот элемент.
- `int order` — определенный во втором параметре `itemid` не обеспечивает расположения элементов по порядку. Если порядок размещения элементов нам важен, его нужно задавать в этом параметре. Поскольку нас порядок элементов не волнует, здесь мы снова ставим `Menu.NONE`.
- `int titleRes` — это идентификатор строкового ресурса, который мы собираемся использовать в качестве названия кнопки. Обратите внимание, что это `Integer` (целое число), а не `String` (строка). Поэтому строки меню требуется заранее определить в файле `string.xml`, который находится в каталоге `res`. Напоминаем, что Android занимается компиляцией строк из `res/strings.xml` в файл `.java` (`R.java`), который присваивает целое число каждой строке. Метод `getString` получает это число для вас (несмотря на название, метод возвращает именно целое число, а не строку).

Для отслеживания событий меню мы переопределяем метод `onOptionsItemSelected`:

```
/**
 * @see android.app.Activity#onOptionsItemSelected(android.view.MenuItem)
 */
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case 0:
            // увеличение
            zoomIn();
            return true;
        case 1:
            // уменьшение
            zoomOut();
            return true;
        case 2:
            // переключение на вид со спутника
            mMap.setSatellite(!mMap.isSatellite());
            return true;
        case 3:
            // переключение на уличную панораму с координатами,
            // соответствующими актуальному центру карты
    }
```

```

String uri =
    "google.streetview:cbll="+curlocation[0]+",
    "+curlocation[1]+"&cbp=1,0,,0,1.0&mz="
    "+mvMap.getZoomLevel();
Intent streetView = new Intent(
    android.content.Intent.ACTION_VIEW,
    Uri.parse(uri)
);
startActivity(streetView);
return true;
case 4:
    // переключение в режим просмотра дорожного трафика
    mvMap.setTraffic(!mvMap.isTraffic());
    return true;
case 5:
    // показ активности со списком вакансий
    startActivity(new Intent(MicroJobs.this,
        MicroJobsList.class));
    return true;
}
return false;
}

```

Мы пользуемся MenuItem, а переключатель предусматривает вариант для каждой кнопки, которую мы определили в меню. Мы уже встречали код, подобный тому, что содержится в каждом из вариантов.

Управление картой с клавиатуры

Некоторые пользователи предпочитают управлять картой с клавиатуры (обычно это один щелчок, в отличие от двух щелчков, вызывающих событие меню). Активируя такое поведение, мы также увидим, как в принципе нужно отвечать на события KeyPad. Итак, мы добавили определенный код для увеличения и уменьшения масштаба, а также для выхода из текущей активности:

```

/**
 * @see android.app.Activity#onKeyDown(int, android.view.KeyEvent)
 */
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    switch (keyCode) {
        case KeyEvent.KEYCODE_DPAD_UP: // увеличить масштаб
            zoomIn();
            return true;
        case KeyEvent.KEYCODE_DPAD_DOWN: // уменьшить масштаб
            zoomOut();
            return true;
        case KeyEvent.KEYCODE_BACK: // возврат (то есть выход из приложения)
            finish();
            return true;
        default:

```

```

        return false;
    }
}

```

Для отслеживания событий нажатия клавиши (**key-down**) мы просто **переопределяем** `onKeyDown` (как это описано в подразделе «Слушание событий клавиатуры» раздела «Множественные указатели и жесты» главы 6) и предоставляем переключатель для различных интересующих нас клавиш. Кроме кодов клавиш, которые мы и так ожидаем встретить на устройстве (`KEYCODE_A`, ... `KEYCODE_Z`; а также `KEYCODE_SPACE`, `KEYCODE_SHIFT_LEFT` и `KEYCODE_SHIFT_RIGHT`), в Android содержатся такие коды клавиш, которых может и не быть на том или ином телефоне (`KEYCODE_CAMERA` и `KEYCODE_VOLUME_UP`). Полный набор кодов клавиш приводится по адресу <http://code.google.com/android/reference/android/view/KeyEvent.html>.

Геолокация без использования карт

Что делать, если вашей активности требуется доступ к геолокационной информации, но в этой активности отсутствует `MapView`? При использовании `MapView Android` значительно все упрощает, позволяя применять `My LocationOverlay`. Но если вам все же не нужна карта, то все равно будет не слишком сложно получить геолокационную информацию. Код из этого раздела не входит в состав программы `MJAndroid`, но показывает, как можно получить геолокационную информацию, не пользуясь `MapView`.

Рассмотрим очень простое приложение, состоящее всего из одной активности. В этой программе актуальное местоположение отображается в поле `TextView`.

Файлы описания и компоновки

Ниже приведен файл описания этой программы — `AndroidManifest.xml`. В редакторе нам потребовалось внести в файл единственное изменение — добавить `uses-permission` для `android.permission.ACCESS_FINE_LOCATION` (в предпоследней строке файла). Нам всегда требуется такое право доступа, чтобы получать информацию о местоположении от нашего геолокационного поставщика, использующего систему GPS:

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.microjobsinc.dloc"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity android:name=".Main"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

```

```

        </activity>
    </application>

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION">
    </uses-permission>
</manifest>

```

Мы использовали очень простой файл компоновки, в состав которого входит четыре `TextView`: по одной метке и по одному текстовому полю для значений широты и долготы:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView
        android:id="@+id/lblLatitude"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Latitude:"
    />
    <TextView
        android:id="@+id/tvLatitude"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
    />
    <TextView
        android:id="@+id/lblLongitude"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Longitude:"
    />
    <TextView
        android:id="@+id/tvLongitude"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
    />
</LinearLayout>

```

Подключение к поставщику геолокации и получение обновлений

Для начала рассмотрим активность, которая просто подключается к `LocationProvider` системы глобального позиционирования, получает информацию о нашем актуальном местоположении и отображает ее (без обновлений):

```

package com.oreilly.demo.pa.microJobs;

import android.app.Activity;

```

```
import android.content.Context;
import android.location.Location;
import android.location.LocationManager;
import android.os.Bundle;
import android.widget.TextView;

public class Main extends Activity {
    /** Вызывается при первом создании активности. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // находим текстовые виды
        TextView tvLatitude = (TextView)findViewById(R.id.tvLatitude);
        TextView tvLongitude = (TextView)findViewById(R.id.tvLongitude);

        // получаем описатель для LocationManager
        LocationManager lm = (LocationManager)
            getSystemService(Context.LOCATION_SERVICE); ❶

        // подключаемся к геолокационной службе GPS
        Location loc = lm.getLastKnownLocation("gps"); ❷

        // заполняем текстовые виды
        tvLatitude.setText(Double.toString(loc.getLatitude())); ❸
        tvLongitude.setText(Double.toString(loc.getLongitude()));
    }
}
```

Процедура совершенно незамысловата. Рассмотрим пояснения к коду.

- ❶ Соединение с LocationManager при помощи `getSystemService(Context.LOCATION_SERVICE)`.
- ❷ Запрашиваем LocationManager о том, где будет использоваться `getLastKnownLocation("provider")`.
- ❸ Получаем широту и долготу — эти значения возвращаются от Location и используются при необходимости.

Еще нам нужно получать периодические обновления геолокационной информации от LocationManager так, чтобы мы могли отслеживать собственное движение. Для этого необходимо добавить подпрограмму-слушатель (**listener routine**) и приказать LocationManager вызывать эту подпрограмму, когда появится обновление информации.

Приложение получает доступ к геолокационным обновлениям, поступающим от LocationManager, через класс `DisPlOcListener`. Поэтому создадим экземпляр этого класса в методе `onCreate` нашей главной активности. Нам потребуется переопределить методы в классе `DisPlOcListener`, чтобы код соответствовал определению интерфейса `LocationListener`, но в нашем приложении работать с этим интерфейсом не придется, поэтому данные определения останутся пустыми. Вот полная реализация класса:

```
package com.oreilly.demo.pa.MicroJobs;

import android.app.Activity;
import android.content.Context;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
import android.widget.TextView;

public class Main extends Activity {
    private LocationManager lm;
    private LocationListener locListenD;
    public TextView tvLatitude;
    public TextView tvLongitude;

    /** Вызывается при первом создании активности. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // находим текстовые виды
        tvLatitude = (TextView)findViewById(R.id.tvLatitude);
        tvLongitude = (TextView)findViewById(R.id.tvLongitude);

        // получаем описатель для LocationManager
        LocationManager lm =
            (LocationManager) getSystemService(Context.LOCATION_SERVICE);

        // подключаемся к геолокационной службе GPS
        Location loc = lm.getLastKnownLocation("gps");

        // заполняем текстовые виды
        tvLatitude.setText(Double.toString(loc.getLatitude()));
        tvLongitude.setText(Double.toString(loc.getLongitude()));

        // приказываем Location Manager присылать нам обновления
        // геолокационной информации
        locListenD = new DispLocListener();
        lm.requestLocationUpdates("gps", 30000L, 10.0f, locListenD);
    }

    private class DispLocListener implements LocationListener {

        @Override
        public void onLocationChanged(Location location) {
            // обновляем текстовые виды
            tvLatitude.setText(Double.toString(location.getLatitude()));
            tvLongitude.setText(Double.toString(location.getLongitude()));
        }
    }
}
```

```

@Override
public void onProviderDisabled(String provider) {
}

@Override
public void onProviderEnabled(String provider) {
}

@Override
public void onStatusChanged(String provider, int status,
                             Bundle extras) {
}
}
}

```

Наш метод `onCreate` создает экземпляр класса `DispLocListener` и приказывает, чтобы `LocationManager` обновлял его при необходимости посредством `requestLocationUpdates`. Этот метод принимает четыре параметра.

- `String provider` — указывает, какое местоположение использовать поставщику. В данном случае предполагается, что у нас доступна система GPS.
- `long minTime` — минимальное время между обновлениями, указывается в миллисекундах. `LocationManager` будет делать обновления не чаще чем с таким интервалом. Здесь можно настроить приложение так, чтобы оно не слишком расходовало заряд батареи: чем чаще происходят обновления, тем больше тратится энергии.
- `float minDistance` — минимальное расстояние в метрах, после преодоления которого инициируется обновление. `LocationManager` обновится лишь при условии, что с момента последнего обновления мы прошли минимум такое расстояние.
- `LocationListener listener` — название метода слушателя, который следует вызывать при наличии обновлений. Слушатель — это только что созданный нами экземпляр `DispLocListener`.

Наконец, мы хотим добавить код методов `onPause` и `onResume`, чтобы отключать обновления местоположения, когда карта не отображается на экране устройства, и снова включать, когда карта появляется на экране:

```

/**
 * Отключить обновления местоположения, если включена пауза.
 */
@Override
public void onPause() {
    super.onPause();
    lm.removeUpdates(locListener);
}

/**
 * Возобновить обновления местоположения, когда работа продолжается.
 */
@Override
public void onResume() {

```

```
super.onResume();  
lm.requestLocationUpdates("gps", 30000L, 10.0f, locListenD);  
}
```

Обновление эмулированной геолокации

При разработке и отладке приложения, подобного описанному в предыдущем разделе, мы, как правило, пользуемся эмулятором. Было бы неплохо (возможно, даже существенно), если бы мы могли обновлять текущую геолокацию, которую эмулятор использует при выполнении вашего кода. Такой поставщик имитируемой геолокации может быть очень сложен, но в Android предусмотрено несколько встроенных способов для обновления эмулированной геолокации:

- географическая программа (geo), встроенная в оболочку Android;
- разовые обновления, выполняемые при помощи Dalvik Debug Monitor Server;
- направление движения, последовательно обновляемое при помощи Dalvik Debug Monitor Server.

Рассмотрим эти компоненты.

Использование geo для обновления геолокации

Утилита geo встроена в образ Android, работающий в эмуляторе. Она обладает рядом возможностей, например может добавлять фиктивное местоположение (mock location). Для этого нужно подключиться по Telnet к консоли эмулятора и использовать команду `geo fix`, как показано ниже.

Команду `geo fix` можно использовать для отправки Android геолокационной информации. Это делается путем подключения к консоли эмулированного устройства по протоколу Telnet. Затем LocationProvider будет использовать эти данные как актуальное местоположение:

```
telnet localhost 5554  
Android Console: type 'help' for a list of commands  
OK  
geo fix -122.842232 38.411908 0  
OK
```

`geo fix` принимает три параметра:

- longitude — долгота, указывается в десятичных дробях;
- latitude — широта, также задается в десятичных дробях;
- altitude — высота, указывается в метрах.

Использование DDMS для обновления геолокации

В главе 1 мы обсуждали службу Dalvik Debug Monitor Service (DDMS). Здесь мы поговорим о двух функциях этого инструмента, связанных с обновлениями геолокации. Область Emulator Control (Управление эмулятором) на экране DDMS обеспечивает несколько способов управления работающим эмулятором. После переключения в режим DDMS (нажмите DDMS в верхнем правом углу окна программы Eclipse) должна открыться область Emulator Control (Управление эмулятором), ко-

торая отобразится в середине окна DDMS по левому краю (рис. 15.1). Возможно, потребуется немного прокрутить эту область, чтобы просмотреть все инструменты управления, связанные с Location Controls (Управление геолокационной информацией).

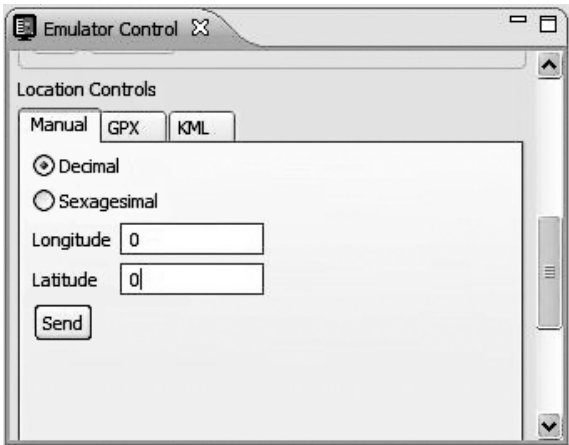


Рис. 15.1. Область Emulator Control (Управление эмулятором) в инструменте DDMS

Чтобы сообщить эмулятору однократное обновление геолокационной информации, просто укажите в соответствующих полях значения широты и долготы и нажмите Send (Отправить).

Если выбрать вкладку GPX или KML, то можно будет загрузить файл в формате GPX или KML, описывающий путь (рис. 15.2). Здесь мы уже загрузили файл OR.kml, имеющийся на сайте этой книги. Он отслеживает путь поблизости от головного офиса издательства O'Reilly, расположенного в городе Севастополь, штат Калифорния.

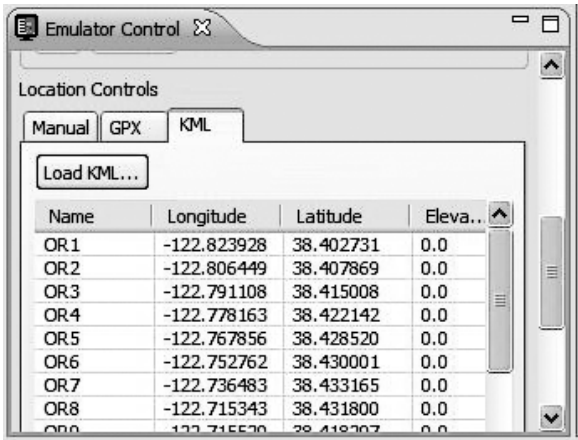


Рис. 15.2. Эмулятор DDMS с обновлениями геолокационной информации в формате KML

Файлы путей в формате GPX можно создавать с помощью разнообразных навигационных программ с поддержкой GPS, а файлы KML — используя Google Earth и многие другие навигационные программы. Файл OR.kml был сгенерирован путем нанесения нескольких меток (placemark) и конкатенации их в единый файл. Вот фрагмент файла OR.kml:

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">
<Document>
  <name>OR1.kml</name>
  <StyleMap id="msn_ylw-pushpin">
    <Pair>
      <key>normal</key>
      <styleUrl>#sn_ylw-pushpin</styleUrl>
    </Pair>
    <Pair>
      <key>highlight</key>
      <styleUrl>#sh_ylw-pushpin</styleUrl>
    </Pair>
  </StyleMap>
  <Style id="sh_ylw-pushpin">
    <IconStyle>
      <scale>1.3</scale>
      <Icon>
        <href>http://maps.google.com/mapfiles/kml/pushpin/ylw-
          pushpin.png</href>
      </Icon>
      <hotSpot x="20" y="2" xunits="pixels" yunits="pixels"/>
    </IconStyle>
    <ListStyle>
    </ListStyle>
  </Style>
  <Style id="sn_ylw-pushpin">
    <IconStyle>
      <scale>1.1</scale>
      <Icon>
        <href>http://maps.google.com/mapfiles/kml/pushpin/ylw-
          pushpin.png</href>
      </Icon>
      <hotSpot x="20" y="2" xunits="pixels" yunits="pixels"/>
    </IconStyle>
    <ListStyle>
    </ListStyle>
  </Style>
  <Placemark>
    <name>OR1</name>
    <LookAt>
      <longitude>-122.7583711698369</longitude>
      <latitude>38.38922415809942</latitude>
      <altitude>0</altitude>
      <range>14591.7166300043</range>
```

```

        <tilt>0</tilt>
        <heading>0.04087372005871314</heading>
        <altitudeMode>relativeToGround</altitudeMode>
    </LookAt>
    <styleUrl>#msn_ylw-pushpin</styleUrl>
    <Point>
        <coordinates>-
            122.8239277647483,38.40273084940345,0</coordinates>
    </Point>
</Placemark>
<Placemark>
    <name>OR2</name>
    <LookAt>
        <longitude>-122.7677364592949</longitude>
        <latitude>38.3819544049429</latitude>
        <altitude>0</altitude>
        <range>11881.3330990845</range>
        <tilt>0</tilt>
        <heading>-8.006283077460853e-010</heading>
        <altitudeMode>relativeToGround</altitudeMode>
    </LookAt>
    <styleUrl>#msn_ylw-pushpin</styleUrl>
    <Point>
        <coordinates>-
            122.8064486052584,38.40786910573772,0</coordinates>
    </Point>
</Placemark>
<Placemark>
    <name>OR3</name>
    <LookAt>
        <longitude>-122.7677364592949</longitude>
        <latitude>38.3819544049429</latitude>
        <altitude>0</altitude>
        <range>11881.3330990845</range>
        <tilt>0</tilt>
        <heading>-8.006283077460853e-010</heading>
        <altitudeMode>relativeToGround</altitudeMode>
    </LookAt>
    <styleUrl>#msn_ylw-pushpin</styleUrl>
    <Point>
        <coordinates>-
            122.7911077944045,38.41500788727795,0</coordinates>
    </Point>
</Placemark>
...

```

StreetView

До версии Android 2.3. в библиотеке Google Map API существовал метод `setStreetView(boolean)`, применявшийся с `MapView`. Но сегодня этот метод считается устаревшим,

и Google рекомендует использовать приложение StreetView. Приложение Google StreetView имеется на большинстве телефонов с Android. Если оно не установлено, то его можно скачать на рынке Android Market.

Для работы с приложением StreetView, как и с другими приложениями, сгенерируйте намерение с соответствующим URI, а затем воспользуйтесь им для запуска активности. Вот URI для запуска приложения StreetView (обратите внимание: параметр `cbll` является необходимым, а `cbp` и `mz` — опциональными):

```
google.streetview:cbll=lat,lng&cbp=1,yaw,.pitch,zoom&mz=mapZoom
```

В состав этого URI входят следующие переменные:

- `lat` — широта;
- `lng` — долгота;
- `yaw` — горизонтальная панорама с просмотром из центра, начиная с севера в градусах; градусная мера возрастает по часовой стрелке (две запятые в конце следует оставлять для обеспечения обратной совместимости);
- `pitch` — вертикальная панорама, с просмотром из центра, градусная мера изменяется от -90 до 90 (сверху вниз);
- `zoom` — масштабирование панорамы; 1.0 — исходная величина, 2.0 — $2\times$, 3.0 — $4\times$;
- `mapZoom` — степень масштабирования для самой карты.

Пример подготовки и запуска намерения `streetView`:

```
String uri = "google.streetview:cbll=42.352299,-71.063979&cbp=1,0,,0,1.0&mz=12";  
Intent streetView = new Intent(android.content.Intent.ACTION_VIEW, Uri.parse(uri));  
startActivity(streetView);
```

16 Мультимедиа

В современном мире конвергентных технологий мобильный телефон имеет множество функций, кроме обычных голосовых вызовов. Мультимедийные возможности, то есть воспроизведение и запись аудио и видео, — одна из таких важных сфер, и с этим согласятся многие пользователи. Просто осмотритесь в толпе — и вы обязательно заметите несколько человек, которые пользуются на мобильном телефоне различными программами и делятся с друзьями записями, которые сами и сделали. Android предоставляет API, обеспечивающие простой доступ к этим возможностям. В том числе эти API позволяют встраивать мультимедийные возможности в само приложение и управлять ими.

Аудио и видео

Android поддерживает воспроизведение большинства популярных форматов аудио и видео. Кроме того, система позволяет записывать медийную информацию в нескольких форматах. Записи сохраняются в файлах, при необходимости их можно переносить в долговременное хранилище медийной информации. MediaStore — это специальный поставщик содержимого в Android, обеспечивающий сохранение и совместное использование медийных данных, например изображений, видео и аудио. После того как метаданные, ассоциированные с медиафайлами, оказываются в этом поставщике содержимого, их могут использовать другие приложения.

На момент написания книги большинство устройств Android, имеющих доступ к рынку, поддерживают следующие аудио- и видеоформаты. Обратите внимание на то, что производители устройств могут добавлять и поддержку других форматов, не перечисленных здесь.

○ Аудио:

- AAC LC/LTP *;
- HE-AACv1 (AAC+);
- HE-AACv2 (улучшенный AAC+);
- AMR-NB *;
- AMR-WB *;
- MP3;
- FLAC (Android 3.1+);

- MIDI;
 - Ogg Vorbis;
 - PCM/WAVE.
- Видео:
- H.263 *;
 - H.264 AVC * (Android 3.0+);
 - MPEG-4 SP;
 - VP8 (Android 2.3.3+).

Звездочка (*) указывает форматы, для которых поддерживается кодировка. Для всех остальных форматов поддерживается только декодировка.

Более подробная актуальная информация приводится на сайте разработчиков Android по адресу <http://developer.android.com/guide/appendix/media-formats.html>.

Воспроизведение аудио и видео

Android предоставляет стандартное средство для воспроизведения аудио и видео: класс `MediaPlayer`. При работе с аудиоконтентом этот класс позволяет воспроизводить и необработанные данные. Такая возможность очень полезна в сложных приложениях, где аудио генерируется динамически.

В ходе жизненного цикла `MediaPlayer` проходит несколько этапов:

- бездействие (`idle`) — инстанцирован `MediaPlayer`;
- инициализирован (`initialized`) — установлен источник медийной информации;
- приготовление (`preparing`) — `MediaPlayer` готовит медиаисточник к воспроизведению;
- приготовлен (`prepared`) — `MediaPlayer` готов к воспроизведению;
- запущен (`started`) — воспроизведение идет;
- приостановлен (`paused`) — воспроизведение приостановлено;
- воспроизведение завершено (`playback completed`) — воспроизведение исходной информации завершено (и его можно начать заново);
- остановлен (`stopped`) — `MediaPlayer` не готов к воспроизведению исходной информации;
- конец (`end`) — `MediaPlayer` больше не работает, и связанные с ним ресурсы освобождаются.

Более глубокое представление об этих состояниях позволяет составить диаграмму состояний, приведенная на сайте разработчиков Android по адресу <http://developer.android.com/reference/android/media/MediaPlayer.html#StateDiagram>. Перед тем как начать работу с `MediaPlayer`, будет полезно рассмотреть серию этапов, происходящих в вашем приложении.

1. Создание экземпляра `MediaPlayer` при помощи метода `create()` (состояние бездействия).
2. Инициализация `MediaPlayer` с медийным источником, который следует воспроизвести (инициализированное состояние).

3. Подготовка MediaPlayer к воспроизведению при помощи метода `prepare()` (состояние приготовления и готовности).
4. Воспроизведение MediaPlayer при помощи метода `start()` (рабочее состояние).
5. В ходе воспроизведения по желанию можно приостановить, остановить окончательно или повторно воспроизвести MediaPlayer (рабочее состояние, состояния паузы, завершения воспроизведения и остановки работы).
6. После того как работа будет завершена, обязательно высвободите ресурсы, ассоциированные с MediaPlayer. Для этого вызывается метод `release()` (конечное состояние).

В следующих подразделах эти этапы рассматриваются более подробно.

Воспроизведение аудио

Для воспроизведения аудио можно воспользоваться одним из двух методов — MediaPlayer и AudioTrack. MediaPlayer — это стандартный, простой способ воспроизведения. Его данные должны быть потоковыми или находиться в файле. AudioTrack, напротив, обеспечивает прямой доступ к необработанным аудиоданным, расположенным в памяти.

Воспроизведение аудио при помощи MediaPlayer

Начиная работать с MediaPlayer, необходимо определить, будем ли мы использовать файл, расположенный в каталоге с ресурсами приложения. Если мы собираемся сделать именно так, то в MediaPlayer есть удобный статический метод, который задает источник данных и готовит плеер к работе:

```
MediaPlayer mediaPlayer = MediaPlayer.create(this, R.raw.example);
```

Если вы не используете ресурс приложения, то есть, например, ссылаетесь на аудиофайл, находящийся в файловой системе (карта памяти и т. п.) или на сайте (допустим, по адресу <http://SomeServer/SomeAudioFile.mp3>), то вам придется вручную задать источник данных и вызвать его. Данные можно взять по уникальному идентификатору, сделав такой вызов:

```
setDataSource(context, uri)
```

Контекст, который задается в первом аргументе, позволяет MediaPlayer получить доступ к ресурсам самого приложения и, как следствие, дает возможность получить данные по URI. Подойдет как контекст приложения, так и контекст активности.

Другой вариант — указать абсолютный путь к файлу при помощи:

```
setDataSource(path)
```

API версии 9 позволяет добавить к плееру некоторые дополнительные эффекты (например, реверберацию). Можете задавать любые звуковые эффекты на этапе установки источника данных до вызова метода `prepare()`:

```
MediaPlayer mediaPlayer = new MediaPlayer();
```

```
// Uri mediaReference = "http://someUriToaMediaFile.mp3";
```

```
// mediaPlayer.setDataSource(this, mediaReference);
```

```
// использование абсолютного пути  
mediaPlayer.setDataSource("/sdcard/somefile.mp3");
```

```
// подготовка медиаплеера  
mediaPlayer.prepare();
```

После того как `MediaPlayer` будет подготовлен, можно начинать воспроизведение:

```
mediaPlayer.start();
```

Во время воспроизведения плеер можно поставить на паузу или остановить вообще. Чтобы возобновить воспроизведение из состояния паузы, достаточно просто снова вызвать `start()`. Когда `MediaPlayer` остановлен вообще, его можно вновь запустить только после повторной активации методом `reset()`, который заново выполняет инициализацию с источником данных, как это было показано выше, и задействует метод `prepare()`. Рассмотрим следующий код:

```
mediaPlayer.pause(); // пауза  
mediaPlayer.start(); // возобновление воспроизведения после паузы
```

```
mediaPlayer.stop(); // остановка
```

```
...
```

```
// для продолжения воспроизведения нужно выполнить перезапуск  
mediaPlayer.reset();  
// теперь для возобновления воспроизведения медиаплеер должен повторно  
// пройти инициализацию
```

Пока `MediaPlayer` проигрывает аудио, вы можете отслеживать, в какой точке файла сейчас происходит воспроизведение, при помощи метода `getCurrentPosition()`. Этот метод возвращает количество времени, которое прошло с начала воспроизведения, в миллисекундах:

```
mediaPlayer.getCurrentPosition();
```

После того как `MediaPlayer` больше не будет вам нужен, обязательно освободите связанные с ним ресурсы, чтобы система снова могла ими пользоваться:

```
mediaPlayer.release();
```

Воспроизведение аудио при помощи `AudioTrack`

`AudioTrack` обеспечивает значительно более прямой метод воспроизведения аудио. В следующем примере показаны параметры, необходимые, чтобы настроить `AudioTrack`:

```
File mediafile = new File(mediaFilePath);  
short[] audio = new short[(int) (mediafile.length()/2)];
```

```
// считывание файла и заполнение audio[]
```

```
AudioTrack audiotrack = new AudioTrack(
```

```
// тип потока
AudioManager.STREAM_MUSIC,
// частота
11025,
// конфигурация канала — моно, стерео и т. д.
AudioFormat.CHANNEL_CONFIGURATION_MONO,
// кодировка аудио
AudioFormat.ENCODING_PCM_16BIT,
// длина
audio.length,
// режим
AudioTrack.MODE_STREAM
);
```

Метод `AudioTrack` позволяет задать тип потока аудио (музыка, звонок, будильник, голосовой вызов и т. д.), частоту образца в герцах (44100, 22050, 11025), конфигурацию аудио (моно или стерео), формат и кодировку аудио, длину аудио, выраженную в байтах, а также режим (статический или потоковый). Экземпляр класса `AudioTrack` в Android достаточно один раз сконфигурировать, и он автоматически будет распознавать, как взаимодействовать с оборудованием устройства, обеспечивая максимально удобную работу с медиа.

Для воспроизведения аудио выполняется метод `play()`, и данные считываются с устройства в буфер:

```
// начало воспроизведения
audiotrack.play();

// считывание аудио с устройства в буфер
audiotrack.write(audio, 0, audio.length);
```

Чтобы приостановить воспроизведение, используйте метод `pause()`:

```
// пауза
audiotrack.pause();
```

Чтобы остановить воспроизведение, переведите дорожку в состояние «остановлено». Если эта дорожка вам больше не нужна, освободите ее. В противном случае, если потребуется вновь воспроизвести аудио, сначала его нужно будет повторно инициализировать:

```
// остановить
audiotrack.stop();

// освободить все ресурсы
audiotrack.release();
```

Воспроизведение видео

Воспроизведение видео в отличие от воспроизведения аудио осуществляется только при помощи `MediaPlayer`. Видеоэквивалента для `AudioTrack` не существует. Видео работает с `MediaPlayer` примерно так же, как и аудио, но вы должны дополнительно указывать вид (называемый поверхностью), на котором видео будет отображаться. В Android предоставляется удобный элемент управления, в котором уже есть

собственная поверхность: вид `VideoView`. Ниже мы покажем, как им пользоваться. В этом примере будет описано добавление опционального контроллера, который позволяет пользователю управлять воспроизведением при помощи простого интерфейса. В этом интерфейсе есть кнопки для запуска, остановки и приостановления воспроизведения, а также имеется ползунок, позволяющий переходить вперед или назад по мере воспроизведения видео:

```
// создание вида (в данном случае он уже включен в файл разметки формы)
VideoView videoview = (VideoView) findViewById(R.id.videoview);
videoview.setKeepScreenOn(true);

// используется при потоковом воспроизведении
if (videouri != null) videoview.setVideoURI(videouri);
// абсолютный путь, если мы работаем с файлом
else videoview.setVideoPath(videopath);

// добавим элемент управления, который поможет нам контролировать
// воспроизведение
mediacontroller = new MediaController(this);
mediacontroller.setAnchorView(videoview);
videoview.setMediaController(mediacontroller);
if (videoview.canSeekForward())
videoview.seekTo(videoview.getDuration()/2);

// запуск воспроизведения
videoview.start();
```

Запись аудио и видео

Стандартный класс, поддерживающий запись, называется `MediaRecorder`. Он во многом напоминает `MediaPlayer` и в течение жизненного цикла проходит через различные состояния. Эти состояния перечислены ниже (подробнее о них рассказывает диаграмма состояний, приведенная на сайте разработчиков Android по адресу <http://developer.android.com/reference/android/media/MediaRecorder.html>):

- инициализация (`initialize`) — инстанцирование класса `MediaRecorder`;
- инициализирован (`initialized`) — `MediaRecorder` готов к использованию;
- источник данных сконфигурирован (`DataSource configured`) — источник медиа (где будет размещаться вывод) сконфигурирован;
- подготовлен (`prepared`) — `MediaRecorder` подготовлен к записи;
- запись (`recording`) — идет запись;
- освобождено (`released`) — все ресурсы высвобождены.

Для использования `MediaRecorder` в файле описания необходимо задать несколько прав доступа:

- чтобы разрешить видеозапись, активируйте `RECORD_VIDEO` и `CAMERA`:

```
<uses-permission android:name="android.permission.RECORD_VIDEO"/>
<uses-permission android:name="android.permission.CAMERA"/>
```

- для записи аудио активируйте RECORD_AUDIO:

```
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
```

Запись аудио

Существует три метода для записи аудио. Стандартный метод называется `MediaRecorder`. Наиболее простой метод связан с использованием намерения — `Intent`. Метод `AudioRecorder` можно использовать для записи непосредственно из аппаратных буферов.

Запись аудио при помощи `MediaRecorder`

Сначала инициализируем `MediaRecorder`. Затем задаем информацию об источнике данных (источник ввода аудио, формат вывода, тип кодировки, где именно будет записываться файл и т. д.). Начиная с версии 8, можно также задавать скорость цифрового потока (битрейт) и частоту дискретизации (`sampling rate`). Когда все это будет сделано, вызываем метод `prepare()`:

```
// инициализация MediaRecorder
MediaRecorder mediarecorder = new MediaRecorder();

// конфигурирование источника данных
// источник ввода аудио
mediarecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
// формат вывода
mediarecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
// кодировка
mediarecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
// использование абсолютного пути к файлу, в котором сохранен вывод
mediarecorder.setOutputFile("/sdcard/audiorecordexample.3gpp");

// подготовка к записи
mediarecorder.prepare();
```

Затем, когда потребуется начать запись, вызываем метод `start()`:

```
mediarecorder.start();
```

Когда запись необходимо остановить, вызываем метод `stop()`. Если после этого вы собираетесь продолжить запись, вызовите `reset()`, чтобы принудительно перевести `MediaRecorder` в исходное состояние бездействия. Затем переконфигурируйте источник данных, чтобы снова подготовить `MediaRecorder`:

```
mediarecorder.stop();
...
mediarecorder.reset();
```

Когда `MediaRecorder` уже не нужен, убедитесь, что вы его высвободили:

```
mediarecorder.release();
```

В следующем примере показано небольшое удобное приложение, использующее разработанный нами код, чтобы предоставить пользователю кнопку **Record** (Запись).

При нажатии этой кнопки метод `record` выполняется применительно к файлу, путь к которому уже проставлен. После этого становится видна кнопка **Stop** (Стоп), а кнопка **Record** (Запись) становится невидимой. После нажатия кнопки **Stop** (Стоп) вызывается метод `stopRecord` и кнопка **Record** (Запись) вновь становится видимой:

```
public class AudioRecorder extends Activity {
    private MediaRecorder mediarecorder;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.audiorecorderlayout);

        ImageButton recordbutton = (ImageButton) findViewById(R.id.record);
        recordbutton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                record("/sdcard/audiorecordexample.3gpp");
            }
        });

        ImageButton stopbutton = (ImageButton) findViewById(R.id.stop);
        stopbutton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                stopRecord();
            }
        });
    }

    private void record(String filePath) {
        try {
            File mediafile = new File(filePath);
            if(mediafile.exists()) {
                mediafile.delete();
            }
            mediafile = null;

            // кнопка записи исчезает
            ImageButton button = (ImageButton)
                findViewById(R.id.record);
            button.setVisibility(View.GONE);
            // появляется кнопка "Стоп"
            ImageButton stopbutton = (ImageButton)
                findViewById(R.id.stop);
            stopbutton.setVisibility(View.VISIBLE);

            // настраиваем средство записи медиа
            if(mediarecorder == null) mediarecorder =
                new MediaRecorder();
            mediarecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
            mediarecorder.setOutputFormat(
                MediaRecorder.OutputFormat.THREE_GPP);
            mediarecorder.setAudioEncoder(
```

```

        MediaRecorder.AudioEncoder.AMR_NB);
mediarecorder.setOutputFile(filePath);

        // готовим средство записи медиа
mediarecorder.prepare();
        // запускаем средство записи медиа
mediarecorder.start();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private void stopRecord() {
    // останавливаем средство записи медиа
mediarecorder.stop();
    // заново устанавливаем средство записи медиа
mediarecorder.reset();

    // появляется кнопка записи
ImageButton button = (ImageButton)
        findViewById(R.id.record);
button.setVisibility(View.VISIBLE);
    // исчезает кнопка "Стоп"
ImageButton stopbutton = (ImageButton)
        findViewById(R.id.stop);
stopbutton.setVisibility(View.GONE);
}
}

```

Запись аудио с применением намерений

Проще всего записывать аудио с помощью Intent. Просто создайте намерение (intent) `MediaStore.Audio.Media.RECORD_SOUND_ACTION` и запустите его методом `startActivityForResult()`, запуск происходит из активности `Activity`. Так запускается стандартный инструмент записи аудио, который имеется на большинстве устройств с Android. Далее записывается определенное аудио:

```

Intent intent = new Intent(MediaStore.Audio.Media.RECORD_SOUND_ACTION);
startActivityForResult(intent, 1); // намерение и requestCode (код запроса) 1

```

После того как запись завершится и инструмент аудиозаписи закончит работу, ваша активность `Activity`, сделавшая вызов к `startActivityForResult()`, будет снова перенесена на передний план. Когда это произойдет, метод `onActivityResult()`, относящийся к вашей активности `Activity`, будет запущен с указанным вами кодом запроса (`requestCode`), в данном случае — 1, кодом результата (OK или ошибка) и с намерением, несущим уникальный идентификатор ресурса, который указывает на записанный аудиофайл:

```

protected void onActivityResult(int requestCode, int resultCode, Intent
                                intent) {
    // Это наш код запроса?

```

```

    if (requestCode == 1) {
        // Дает ли этот код запроса результат OK?
        if (resultCode == RESULT_OK) {
            // Давайте получим uri.
            Uri audioUri = intent.getData();
            // Давайте воспроизведем информацию, на которую ссылается
            // uri, или что-нибудь с ней сделаем.
            playAudio(audioUri);
        }
    }
}

```

Запись аудио с применением AudioRecorder

Параллельно с AudioTrack, AudioRecorder предоставляет гораздо более прямой способ записи:

```
short[] buffer = new short[10000];
```

```

recorder = new AudioRecord( // источник, из которого следует записывать
    MediaRecorder.AudioSource.MIC,
                                // частота
    11025,
                                // конфигурация канала — моно, стерео и т. д.
    AudioFormat.CHANNEL_CONFIGURATION_MONO,
                                // кодировка аудио
    AudioFormat.ENCODING_PCM_16BIT,
                                // размер буфера
    buffer.length
);

```

Метод AudioRecord указывает тип источника, с которого будет вестись запись (микрофон, камкордер¹, голосовой вызов), частоту дискретизации в герцах (44100, 22050 или 11025), конфигурацию аудио (моно или стерео), формат/кодировку аудио и длину буфера в байтах. Обратите внимание на то, что именно размер этого буфера определяет, как долго AudioRecord сможет вести запись до того, как начнет «наслаивать» данные, которые пока еще не считаны. Данные должны считываться с аудиооборудования более мелкими фрагментами, чем общий размер буфера записи. Метод AudioRecord в Android достаточно сконфигурировать один раз, после чего он автоматически будет распознавать, как взаимодействовать с оборудованием конкретного устройства. Поэтому работа с программой будет протекать без всяких проблем.

Чтобы начать запись, переведите AudioRecord в состояние Record (Запись) и начинайте циклически считывать данные из аппаратного буфера:

```

recorder.startRecording();
while(recordablestate) {
    try {
        // считываем данные до достижения размера буфера

```

¹ Микрофон, направленный в ту же сторону, что и объектив камеры.

```

int readBytes = recorder.read(buffer, 0, buffer.length);

// осуществляем какие-либо операции со считанными байтами
} catch (Exception t) {
    recordablestate = false;
}
}

```

Чтобы остановить запись, переведите `AudioRecord` в состояние **Stop** (Стоп). Если вы больше не собираетесь ничего записывать, то не забудьте высвободить все ресурсы, которые были вовлечены в процесс записи. В противном случае можно вызвать метод `startRecording()`, чтобы вновь начать запись:

```

// остановка записи
recorder.stop();

// высвобождение ресурсов, использовавшихся при записи
recorder.release();

```

Запись видео

Видео можно записывать двумя способами: с помощью `MediaRecorder` или посредством `Intent`. Запись необработанной информации не поддерживается, как и в случае с аудио.

Запись видео с применением `MediaRecorder`

Процесс записи видео с применением `MediaRecorder` во многом напоминает аналогичный процесс с аудио. Инициализируется `MediaRecorder`, подготавливается источник данных, потом `MediaRecorder` запускается. Можно предложить пользователю окно предварительного просмотра, чтобы он мог отслеживать снимаемое видео. Для этого, как и для воспроизведения аудио, предоставляется поверхность — о ней мы говорили выше. Обычно применяется вид `VideoView`:

```

// инициализация
MediaRecorder mediarecorder = new MediaRecorder();

// установка источника данных
mediarecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
mediarecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
mediarecorder.setOutputFormat(MediaRecorder.OutputFormat.DEFAULT);
mediarecorder.setAudioEncoder(MediaRecorder.AudioEncoder.DEFAULT);
mediarecorder.setVideoEncoder(MediaRecorder.VideoEncoder.DEFAULT);
mediarecorder.setOutputFile("/sdcard/someexamplevideo.mp4");

// Указание поверхности, на которой будет отображаться информация
// для предварительного просмотра. В данном случае используется
// вид VideoView.
videoview = (VideoView) findViewById(R.id.videosurface);
SurfaceHolder holder = videoview.getHolder();

```

```
mediarecorder.setPreviewDisplay(holder.getSurface());
```

```
// подготовка  
mediarecorder.prepare();
```

```
// начало записи  
mediarecorder.start();
```

Запись видео с применением намерений

Запись видео с использованием намерений напоминает аналогичный процесс с аудио. В таком случае применяется намерение `MediaStore.ACTION_VIDEO_CAPTURE`, результирующие данные — это URI видеофайла.

Сохраненный медийный контент

Даже когда медийная информация сохраняется в файле (как в случае записи), такой медийный файл не становится автоматически доступным другим приложениям. Чтобы файл стал доступным, его необходимо поместить в `MediaStore` (хранилище для медийного контента). `MediaStore` — это поставщик содержимого, специально предназначенный для хранения и получения медийных данных (изображений, видео, аудио) на устройстве. Для сохранения ссылки на файл создайте объект `ContentValues` и вставьте его в подходящий поставщик содержимого `MediaStore`. В следующем примере происходит вставка аудиофайла с соответствующими метаданными, например с заголовком и именем исполнителя:

```
// генерирование ContentValues и добавление соответствующих  
// значений метаданных
```

```
ContentValues content = new ContentValues();
```

```
//ОЧЕНЬ ВАЖНО! Ссылка должна представлять собой абсолютный путь к данным.  
content.put(MediaStore.MediaColumns.DATA, "/sdcard/AudioExample.3gpp");  
content.put(MediaStore.MediaColumns.TITLE, "AudioRecordExample");  
content.put(MediaStore.MediaColumns.MIME_TYPE, "audio/amr");  
content.put(MediaStore.Audio.Media.ARTIST, "Me");  
content.put(MediaStore.Audio.Media.IS_MUSIC, true);
```

```
// получение преобразователя содержимого  
ContentResolver resolve = getContentResolver();
```

```
// вставка в преобразователь содержимого  
Uri uri = resolve.insert(MediaStore.Audio.Media.EXTERNAL_CONTENT_URI, content);
```

```
// сообщение о вставке; оповещаются все элементы, которых это касается  
sendBroadcast(new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE, uri));
```

17 Сенсоры, коммуникация ближнего поля, речь, жесты и доступность

Благодаря нынешнему техническому прогрессу как пользователь, так и окружающая среда могут взаимодействовать с устройством разнообразными способами. Речь идет и о внешних сенсорах, которые способны определять, когда устройство изменяет ориентацию в пространстве, и о сенсорных экранах (тачскринах), воспринимающих сложные жесты, которые могут инициировать событие на устройстве. Android предоставляет такие API, которые позволяют разработчику получать доступ к сенсорам, а пользователю — взаимодействовать с устройством различными интерактивными способами. В этой главе мы исследуем некоторые подобные API и поговорим о сенсорах, NFC (коммуникации ближнего поля), библиотеке жестов и доступности устройства.

Сенсоры

Функционал современного смартфона не ограничивается лишь возможностями, связанными с той или иной входящей или исходящей коммуникацией. После оснащения телефонов внешними сенсорами, которые способны сообщать информацию о той среде, где находится телефон, аппарат стал более мощным и полезным как для пользователя, так и для разработчика. Начиная с версии Android 1.5 (API уровня 3), в телефоне доступен стандартный набор сенсоров. К физическим сенсорам относятся в том числе акселерометр, измеряющий ускорение по различным осям, гироскоп, вычисляющий вращательное изменение вокруг определенных осей, сенсоры магнитного поля, отслеживающие силу магнитных полей по ряду осей, световые сенсоры, измеряющие силу окружающего света, сенсоры близости, определяющие, насколько близко устройство находится к внешнему объекту, температурные сенсоры, измеряющие температуру окружающей среды, а также сенсоры давления, действующие подобно барометру. Величина, непосредственно измеренная каждым из сенсоров, считается результатом «сырых измерений», поэтому ассоциативный сенсор можно считать сенсором необработанных данных (*raw sensor*). Некоторые сенсоры позволяют комбинировать измерения или собирать их

результаты, и расчеты могут производиться над собранными величинами, чтобы получить результаты более сложных измерений. Например, при интеграции величин вращательных изменений, фиксируемых гироскопом, и измерения времени, за которое эти изменения происходят, можно рассчитать вектор вращения. Часто сложные измерения такого рода выполняются комбинированными сенсорами.

Для доступа к набору сенсоров Android предоставляет удобную системную службу, называемую `SensorManager`. Доступ к этой службе осуществляется при помощи метода `getSystemService()`, относящегося к классу `Context`, с аргументом `Context.SENSOR_SERVICE`. Затем посредством `SensorManager` можно обратиться к конкретному сенсору с помощью метода `getDefaultSensor()`.

Правда, иногда может быть возвращен комбинированный сенсор, поэтому, если вы хотите получить доступ к сенсору необработанной информации и к данным, которые с ним ассоциированы, используйте метод `getSensorList()`:

```
SensorManager mngr =  
    (SensorManager) context.getSystemService(Context.SENSOR_SERVICE);  
// получение стандартного акселерометра  
Sensor accel = mngr.getDefaultSensor (Sensor.TYPE_ACCELEROMETER);  
// получение необработанных данных акселерометра  
List<Sensor> list = mngr.getSensorList(Sensor.TYPE_ACCELEROMETER);
```

После получения сенсора или набора сенсоров вы можете активировать их и приступить к получению их данных, зарегистрировав слушатель, который будет принимать такие данные. Данные начнут прибывать с такой частотой, которую вы зададите в качестве аргумента. Эта частота может иметь значение `SENSOR_DELAY_NORMAL`, `SENSOR_DELAY_UI` (частота, подходящая для базовых взаимодействий с пользовательским интерфейсом), `SENSOR_DELAY_GAME` (высокая частота, которая подойдет для большинства игр), `SENSOR_DELAY_FASTEST` (подача данных сразу же по мере их поступления). Кроме того, задержку между событиями можно указать в миллисекундах:

```
SensorEventListener listener = new SensorEventListener() {  
    @Override  
    public void onAccuracyChanged(Sensor sensor, int accuracy) { }  
  
    @Override  
    public void onSensorChanged(SensorEvent event) { }  
};  
  
// регистрация слушателя  
mngr.registerListener(listener, sensor, SensorManager.SENSOR_DELAY_UI);
```

Два метода `SensorEventListener` — `onAccuracyChanged()` и `onSensorChanged()` — вызываются, когда оказываются доступны данные от интересующего нас сенсора. Метод `onAccuracyChanged()` вызывается всякий раз, когда у сенсора изменяется уровень точности или доля погрешности. Метод `onSensorChanged()`, пожалуй, более интересен, тем, что те данные, которые передает ему сенсор, оказываются обернуты в объект `SensorEvent`.

Исключительно важно отменить регистрацию слушателя и, таким образом, деактивировать сенсор, когда он вам больше не нужен (например, когда работа

активности приостановлена). В противном случае приложение продолжит расходовать ресурсы и впустую тратить энергию. Система не сделает этого за вас, даже если экран будет отключен:

```
mngr.unregisterListener(listener);
```

Пока сенсор включен, объект `SensorEvent` передается слушателю при помощи метода `onSensorChanged()`. Именно значения этого объекта `SensorEvent` — фактор, специфичный для сенсоров каждого типа.

Положение

Координатная система телефона основана на экране и на той ориентации телефона, которая задана по умолчанию. Оси X , Y и Z расположены так, как показано на рис. 17.1, и действуют следующим образом:

- ось X — расположена по горизонтали, положительные значения находятся справа, а отрицательные — слева;
- ось Y — расположена по вертикали, положительные значения идут вверх, а отрицательные — вниз;
- ось Z — положительные значения идут из экрана, вперед, а отрицательные — за экран, назад (начало координат по оси Z находится на плоскости экрана).

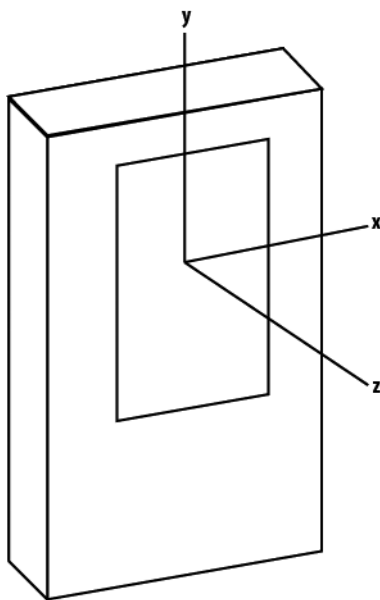


Рис. 17.1. Координатная система телефона

Когда пользователь перемещает телефон, оси перемещаются вместе с ним и не меняются местами.

Точность и флуктуации показаний различных сенсоров зависят от качества оборудования. Зачастую приходится избавляться от значительного дрожания и шума (например, путем использования фильтра нижних частот). Проектирование и создание такого фильтра — задача, которую приходится решать разработчику.

Акселерометр

Акселерометр измеряет ускорение, прилагаемое к устройству, и возвращает значения по трем осям (значение [0] по оси *X*, значение [1] по оси *Y* и значение [2] по оси *Z*). Значения измеряются в единицах системы СИ (м/с^2). Необходимо отметить, что сила тяготения не вычитается из возвращаемых значений. Следовательно, когда устройство лежит на столе (например, экраном вверх), значение [2] составит $9,81 \text{ м/с}^2$.

Поскольку очень часто приходится либо вычитать из общего показателя ускорения силу гравитации, либо определять ускорение по различным осям, Android 2.3 (API уровня 9) также поддерживает сенсор линейного ускорения и сенсор гравитации, которые рассматриваются далее в этой главе.

Гироскоп

Гироскоп измеряет угловую скорость или частоту вращения вдоль трех осей. Все значения исчисляются в рад/с. Вращение против часовой стрелки имеет положительные значения. Наблюдатель, который смотрит на экран устройства как обычно — когда экран расположен в точке (0; 0; 100) в координатной системе устройства, — должен сообщить, что если устройство вращается против часовой стрелки, то его вращение является положительным. Поскольку речь идет об угловой скорости, для расчета угла необходимо интегрировать значения в течение некоторого периода времени:

```
private static final float NS2S = 1.0f / 1000000000.0f;
private float timestamp;
private float[] angle;

@Override
public void onSensorChanged(SensorEvent event) {
    float gyrox = event.values[0];
    float gyroy = event.values[1];
    float gyroz = event.values[2];

    // Здесь мы интегрируем значения во времени, чтобы определить угол
    // вращения вдоль каждой оси.
    if (timestamp != 0) {
        final float dT = (event.timestamp - timestamp) * NS2S;
        angle[0] += gyrox * dT;
        angle[1] += gyroy * dT;
        angle[2] += gyroz * dT;
    }

    timestamp = event.timestamp;
}
```

Поскольку это стандартный набор проблем, Android 2.3 (API уровня 9) поддерживает сенсор для определения вектора вращения. Об этом показателе мы поговорим в следующем пункте.

Вектор вращения

Вектор вращения в Android 2.3 и выше представляет ориентацию устройства как комбинацию угла и оси, под которыми устройство повернулось на угол Θ вокруг оси $\langle x, y, z \rangle$. Хотя все это можно рассчитать при помощи гироскопа, многим разработчикам приходится выполнять такие операции довольно часто. И чтобы упростить процедуру использования, Google внедрил концепцию вращения.

Есть три элемента вектора вращения — $\langle x \cdot \sin(\Theta/2), y \cdot \sin(\Theta/2) \text{ и } z \cdot \sin(\Theta/2) \rangle$. Соответственно магнитуда вектора вращения равна $\sin(\Theta/2)$, а направление вектора вращения равно направлению оси вращения. Три элемента вектора вращения равны последним трем компонентам единичного кватерниона $\langle \cos(\Theta/2), x \cdot \sin(\Theta/2), y \cdot \sin(\Theta/2) \text{ и } z \cdot \sin(\Theta/2) \rangle$. Элементы вектора вращения являются безразмерными величинами.

Линейное ускорение

В Android 2.3 (API уровня 9) поддерживаются сенсоры еще одного типа, призванные упростить типичные вычисления, связанные с применением акселерометра. Пересылаемое значение — это трехмерный вектор, указывающий ускорение вдоль каждой оси устройства, не учитывая гравитации. Это означает, что результирующие значения равны линейному ускорению по каждой из осей минус воздействие гравитации на эту ось. Таким образом, становится проще исключить эффект воздействия гравитационной постоянной, которая непрерывно воздействует на телефон, используемый на Земле. Все значения измеряются в м/с^2 .

Гравитация

Значения, получаемые от этого сенсора, составляют трехмерный вектор, указывающий направление и магнитуду гравитации. Этот сенсор также относится к Android 2.3 (API уровня 9) и обеспечивает стандартные вычисления. Все значения измеряются в м/с^2 .

Другие сенсоры

В Android также поддерживаются следующие сенсоры.

- Сенсор света — предоставляет массив из одного значения (значение [0]), выражающий уровень окружающего света в единицах системы СИ — люксах (lx).
- Сенсор магнетизма — измеряет силу окружающих магнитных полей в микро-теслах (μT) по осям X , Y и Z .
- Сенсор давления — встречается на некоторых устройствах. Если такой сенсор имеется, то его значения исчисляются в килопаскалях (kPa).
- Близость — измеряет массив из одного значения (значение [0]), выражающий расстояние от сенсора, измеряемое в сантиметрах. В некоторых случаях сенсор близости может выполнять только бинарное измерение: «близко» (0) или «далеко» (1). В таком случае расстояние, большее или равное значению метода

`getMaximumRange()`, который относится к сенсору, возвратит результат «далеко», а все остальные значения — результат «близко».

- Температура — это еще один сенсор, который встречается на устройствах нечасто. Значения выражаются в градусах по стоградусной шкале Цельсия (°C). В Android 4.0 (API 14) и выше считается устаревшим.
- Температура окружающей среды — также встречается на устройствах нечасто. Измеряет температуру окружающего воздуха. Значения выражаются в градусах по стоградусной шкале Цельсия (°C). Поддерживается в Android 4.0 (API 14) и выше.
- Относительная влажность — встречается на устройствах нечасто. Измеряет относительную влажность окружающего воздуха. Значения даются в процентах (%). Поддерживается в Android 4.0 (API 14) и выше.

Коммуникация ближнего поля (NFC)

Коммуникация ближнего поля — это технология ближкодействующей (до 20 см), высокочастотной беспроводной коммуникации. Это стандарт, дополняющий более крупный стандарт радиочастотной идентификации (РЧИД), комбинирующий интерфейс смарт-карты и считывателя в единое устройство. Данный стандарт изначально разрабатывался для использования с мобильными телефонами, поэтому он весьма заинтересовал производителей, искавших способы бесконтактной передачи данных (например, при использовании кредитных карточек). Стандарт позволяет использовать коммуникацию ближнего поля несколькими способами.

- Эмуляция карты. Устройство — карточка бесконтактного считывания, поэтому с него могут получать информацию другие считыватели.
- Режим считывания. Устройство может считывать метки РЧИД (радиочастотная идентификация).
- Одноранговый режим. Между двумя устройствами устанавливается прямая и обратная коммуникация для обмена данными.

В Android 2.3 (API уровня 9) Google реализовал функциональность режима считывания при коммуникации ближнего поля. Начиная с Android 2.3.3 (API уровня 10), также появилась возможность записи данных в метку NFC и обмена данными в одноранговом режиме (P2P).

Метки коммуникации ближнего поля состоят из данных, закодированных в формате для обмена данными при коммуникации ближнего поля (NDEF). Этот формат регламентируется в спецификации NFC Forum Type 2. Каждое NDEF-сообщение состоит из одной или более записей NDEF. **Официальная техническая спецификация** коммуникации ближнего поля расположена по адресу <http://www.nfc-forum.org/>. Для разработки и тестирования считывающего приложения, использующего коммуникацию ближнего поля, настоятельно рекомендуется приобрести NFC-совместимое устройство (например, Nexus S, см. <http://www.google.com/phone/detail/nexus-s>) и использовать NFC-совместимую метку.

Для применения функций коммуникации ближнего поля в вашем приложении необходимо объявить в файле описания следующее право доступа:

```
<uses-permission android:name="android.permission.NFC" />
```

Чтобы допустить установку приложения только на те устройства, которые поддерживают NFC, добавьте в файл описания и следующую строку:

```
<uses-feature android:name="android.hardware.nfc" />
```

Считывание метки

Режим считывания предназначен для получения уведомлений при сканировании РЧИД/NFC метки. В Android 2.3 (API уровня 9) единственный способ реализации такой функции — создать активность, которая будет слушать намерения `android.nfc.action.TAG_DISCOVERED`, которые широковещательным способом сообщают о том, что в данный момент считывается метка. В Android 2.3.3 (API уровня 10) предоставляются более обширные средства для получения таких уведомлений в соответствии с процессом, показанным на рис. 17.2.

В Android 2.3.3 (API уровня 10) и выше при обнаружении метки NFC объект метки (`Parcelable`) помещается в намерение (`Intent`) как `EXTRA_TAG`. Затем система начинает отслеживать логический поток, чтобы найти активность, которой лучше всего будет направить намерение. Такой механизм разработан для того, чтобы обеспечить высокую вероятность направления метки к подходящей активности без вывода пользователю диалогового окна для выбора активности (то есть сделать это прозрачным образом) и, таким образом, воспрепятствовать обрыву соединения между меткой и устройством. Этот обрыв может быть спровоцирован ненужным вмешательством пользователя.

Первое, что необходимо проверить, — узнать, есть ли на переднем плане какая-нибудь активность, которая вызвала метод `enableForegroundDispatch()`. Если это так, то намерение передается активности и на этом процесс завершается. Если же нет, то система проверяет первую запись `NdefRecord` в первом `NdefMessage` данных тега. Если `NdefRecord` — это уникальный идентификатор ресурса (URI), рассылка **Smart Poster** или **информация о типе MIME**, то система проверяет наличие активности, зарегистрированной на получение намерения `ACTION_NDEF_DISCOVERED` (`android.nfc.action.NDEF_DISCOVERED`) с нужным типом данных. Если такое намерение существует, то подходящая для его обработки активность (чем точнее совпадение, тем лучше) получает намерение — и на этом все завершается. Если и эти условия не выполняются, то система ищет активность, зарегистрированную на получение `ACTION_TECH_DISCOVERED` и подходящую для специфического набора технологий, соответствующего метке (опять же чем точнее совпадение, тем лучше). Если совпадение имеется, то намерение передается такой активности — и решение найдено.

Тем не менее, если не найдется активность, которая прошла бы предыдущие проверки, намерение в итоге будет передано как действие `ACTION_TAG_DISCOVERED`, и именно так происходила бы обработка метки в версии **Android 2.3 (API уровня 9)**.

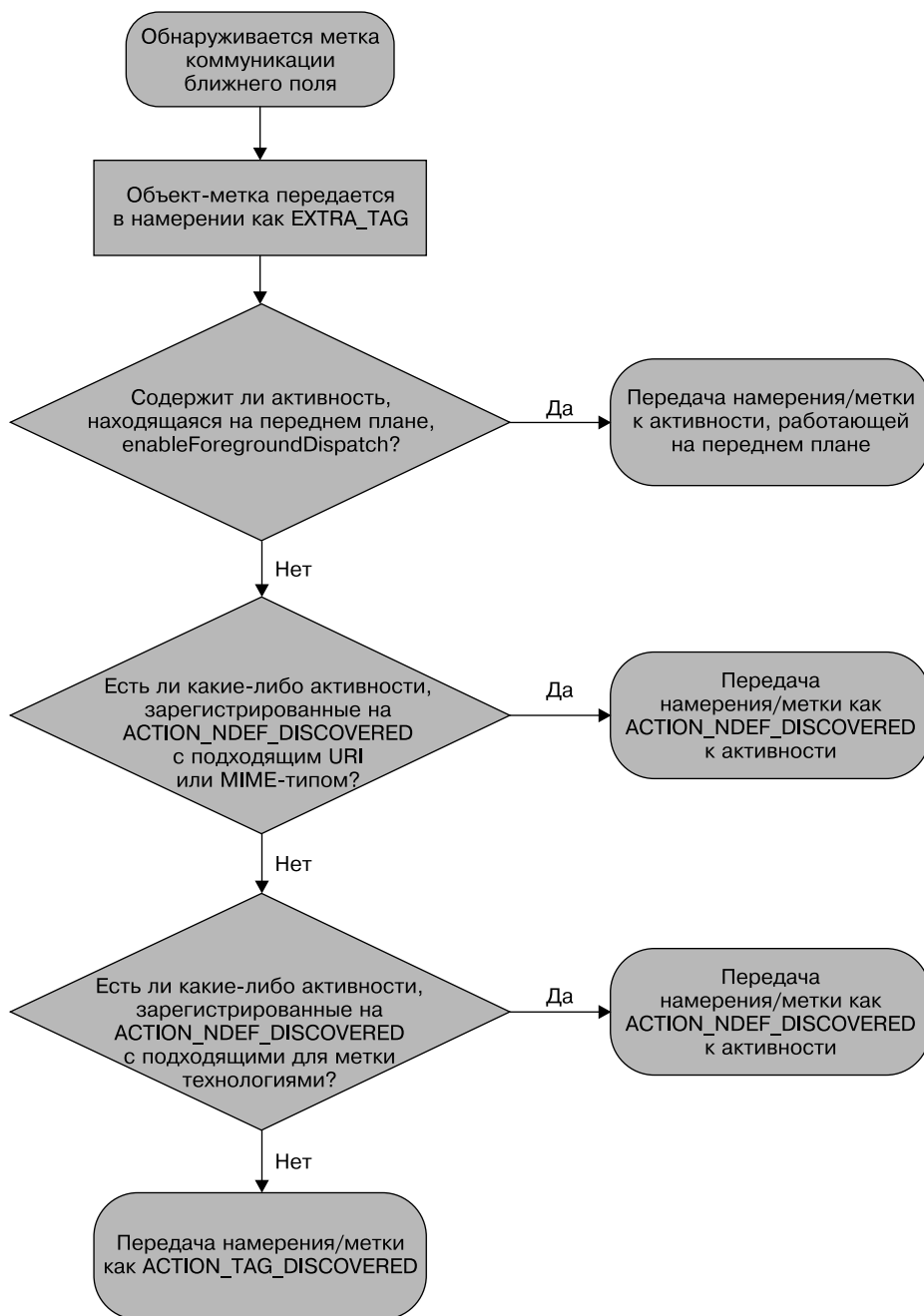


Рис. 17.2. Поток меток коммуникации ближнего поля в Android 2.3.3 (API уровня 10)

Чтобы гарантировать, что именно активность, которая находится на переднем плане, была первой в очереди на получение метки, нужно получать адаптер устрой-

ства коммуникации ближнего поля и вызвать `enableForegroundDispatch` со ссылкой на контекст активности. Конкретный адаптер устройства коммуникации ближнего поля представлен классом `NfcAdapter`. Чтобы получить точный адаптер конкретного устройства, необходимо запустить `getDefaultAdapter()` в версии Android 2.3 (API уровня 9) или `getDefaultAdapter(context)` в Android 2.3.3 (API уровня 10):

```
NfcAdapter adapter = NfcAdapter.getDefaultAdapter();

// --- только для API 10
// NfcAdapter adapter = NfcAdapter.getDefaultAdapter(context);

if(adapter != null) {
    // true при активации, false при ее отсутствии
    boolean enabled = adapter.isEnabled();
}
```

Когда адаптер устройства коммуникации ближнего поля получен, нужно создать намерение `PendingIntent` и передать его методу `enableForegroundDispatch()`. Этот метод должен вызываться из основного потока и только в том случае, если интересующая нас активность находится на переднем плане (после вызова метода `onResume()`):

```
PendingIntent intent =
    PendingIntent.getActivity(this, 0,
        new Intent(this,
            getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP),
        0);

NfcAdapter.getDefaultAdapter(this).enableForegroundDispatch(this, intent,
    null, null);
```

Исключительно важно то, что, когда активность уходит с переднего плана (при вызове метода `onPause()`), вы должны вызвать метод `disableForegroundDispatch()`:

```
@Override
protected void onPause() {
    super.onPause();
    if(NfcAdapter.getDefaultAdapter(this) != null)
        NfcAdapter.getDefaultAdapter(this).disableForegroundDispatch(this);
}
```

Если активность зарегистрирована на `ACTION_NDEF_DISCOVERED`, эта активность должна иметь фильтр намерений (`intent-filter`) `android.nfc.action.NDEF_DISCOVERED` и все фильтры для конкретных (специфичных) данных в файле описания:

```
<activity android:name=".NFC233">
    <!-- слушание android.nfc.action.NDEF_DISCOVERED -->
    <intent-filter>
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
        <data android:mimeType="text/*" />
    </intent-filter>
</activity>
```

Это же касается случая с `TECH_DISCOVERED` (в следующий пример также включен ресурс метаданных, описывающий специфическую технологию. Этот ресурс указан в метке коммуникации ближнего поля, и эту технологию мы будем рассматривать более подробно. Например, речь может идти об информации в формате NDEF):

```
<activity android:name=".NFC233">
    <intent-filter>
        <action android:name="android.nfc.action.TECH_DISCOVERED" />
    </intent-filter>

    <meta-data android:name="android.nfc.action.TECH_DISCOVERED"
        android:resource="@xml/nfcfilter"
    />
</activity>

<?xml version="1.0" encoding="utf-8"?>
    <!-- Берем все, что использует NfcF или содержит NDEF
        в качестве полезной информации. -->
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <tech-list>
        <tech>android.nfc.tech.NfcF</tech>
    </tech-list>

    <tech-list>
        <tech>android.nfc.tech.NfcA</tech>
        <tech>android.nfc.tech.MifareClassic</tech>
        <tech>android.nfc.tech.Ndef</tech>
    </tech-list>
</resources>
```

Регистрация на намерение `ACTION_TAG_DISCOVERED` будет записываться в файле описания вот так:

```
<!-- эта информация отобразится в виде диалогового окна во время
    сканирования метки nfc -->
<activity android:name=".NFC" android:theme="@android:style/Theme.Dialog">
    <intent-filter>
        <action android:name="android.nfc.action.TAG_DISCOVERED"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

Когда метка считана, система ширококестельным образом передает намерение, причем его информационное наполнение (полезная нагрузка) выступает в качестве ассоциированных данных. В **Android 2.3.3 (API уровня 10)** здесь также содержится объект `Tag` (в качестве `EXTRA_TAG`). Этот объект `Tag` предоставляет средства для получения конкретной `TagTechnology` и для выполнения сложных операций (например, ввода-вывода). Необходимо учитывать, что массивы (`Array`), передаваемые этому классу и возвращаемые им, не клонируются, поэтому будьте осторожны и не модифицируйте их:

```
Tag tag = (Tag) intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
```

В Android 2.3 (API уровня 9) и выше идентификатор (ID) метки заключен внутри намерения и зашифрован строкой `android.nfc.extra.ID` (`NfcAdapter.EXTRA_ID`) как массив байтов:

```
byte[] byte_id = intent.getByteArrayExtra(NfcAdapter.EXTRA_ID);
```

Данные упакованы в виде массива `Parcelable`-объектов (`NdefMessage`), зашифрованных строкой `android.nfc.extra.NDEF_MESSAGES` (`NfcAdapter.EXTRA_NDEF_MESSAGES`):

```
Parcelable[] msgs =
    intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
NdefMessage[] nmsgs = new NdefMessage[msgs.length];
for(int i=0;i<msgs.length;i++) {
    nmsgs[i] = (NdefMessage) msgs[i];
}
```

Внутри каждого сообщения `NdefMessage` находится массив из `NdefRecord`. В этой записи всегда будет присутствовать 3-битный **TNF (type name format, формат имени типа)**, тип записи, уникальный идентификатор и полезная нагрузка. Подробнее этот состав рассматривается в документе `NdefRecord` по адресу <http://developer.android.com/reference/android/nfc/NdefRecord.html>.

В настоящее время известно несколько таких типов, четыре наиболее распространенных из них — `TEXT`, `URI`, `SMART_POSTER` и `ABSOLUTE_URI` — мы рассмотрим ниже:

```
// перечисление интересующих нас типов:
private static enum NFCType {
    UNKNOWN, TEXT, URI, SMART_POSTER, ABSOLUTE_URI
}

private NFCType getTagType(final NdefMessage msg) {
    if(msg == null) return null;
    // просто собираем первый распознаваемый элемент:

    for (NdefRecord record : msg.getRecords()) {
        if(record.getTnf() == NdefRecord.TNF_WELL_KNOWN) {
            if(Arrays.equals(record.getType(), NdefRecord.RTD_TEXT)) {
                return NFCType.TEXT;
            }
            if(Arrays.equals(record.getType(), NdefRecord.RTD_URI)) {
                return NFCType.URI;
            }
            if(Arrays.equals(record.getType(), NdefRecord.RTD_SMART_POSTER)) {
                return NFCType.SMART_POSTER;
            }
        } else if(record.getTnf() == NdefRecord.TNF_ABSOLUTE_URI) {
            return NFCType.ABSOLUTE_URI;
        }
    }
    return null;
}
```

Для считывания полезной нагрузки типа `NdefRecord.RTD_TEXT` первый байт этой информации определяет статус и соответственно тип кодировки текстовой полезной нагрузки:

```
/*
 * Первый байт полезной нагрузки содержит поле "кодировки байта состояния"
 * в соответствии с документом с форума NFC "Text Record Type Definition",
 * раздел 3.2.1.
 *
 * Bit_7 поле кодировки текста
 * if Bit_7 == 0 то текст имеет кодировку UTF-8
 * else if Bit_7 == 1 то текст имеет кодировку UTF16
 * Bit_6 в настоящее время всегда равно 0 (зарезервировано
 * для использования в будущем)
 * Биты с 5 до 0 – это длина языкового кода IANA.
 */
private String getText(final byte[] payload) {
    if(payload == null) return null;
    try {
        String textEncoding = ((payload[0] & 0200) == 0) ? "UTF-8" : "UTF-16";
        int languageCodeLength = payload[0] & 0077;
        return new String(payload, languageCodeLength + 1,
            payload.length - languageCodeLength - 1, textEncoding);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

При считывании полезной нагрузки стандартного уникального идентификатора ресурса (тип `NdefRecord.RTD_URI`), первый байт полезной нагрузки определяет префикс URI:

```
/**
 * документ с форума NFC "URI Record Type Definition"
 *
 * Преобразование префикса происходит на основании раздела 3.2.2
 * из документа с форума NFC "URI Record
 * Type Definition".
 */
private String convertUriPrefix(final byte prefix) {
    if(prefix == (byte) 0x00) return "";
    else if(prefix == (byte) 0x01) return "http://www.";
    else if(prefix == (byte) 0x02) return "https://www.";
    else if(prefix == (byte) 0x03) return "http://";
    else if(prefix == (byte) 0x04) return "https://";
    else if(prefix == (byte) 0x05) return "tel:";
    else if(prefix == (byte) 0x06) return "mailto:";
    else if(prefix == (byte) 0x07) return "ftp://anonymous:anonymous@";
    else if(prefix == (byte) 0x08) return "ftp://ftp.";
    else if(prefix == (byte) 0x09) return "https://";
}
```

```

else if(prefix == (byte) 0x0A) return "sftp://";
else if(prefix == (byte) 0x0B) return "smb://";
else if(prefix == (byte) 0x0C) return "nfs://";
else if(prefix == (byte) 0x0D) return "ftp://";
else if(prefix == (byte) 0x0E) return "dav://";
else if(prefix == (byte) 0x0F) return "news:";
else if(prefix == (byte) 0x10) return "telnet://";
else if(prefix == (byte) 0x11) return "imap:";
else if(prefix == (byte) 0x12) return "rtsp://";
else if(prefix == (byte) 0x13) return "urn:";
else if(prefix == (byte) 0x14) return "pop:";
else if(prefix == (byte) 0x15) return "sip:";
else if(prefix == (byte) 0x16) return "sips:";
else if(prefix == (byte) 0x17) return "tftp:";
else if(prefix == (byte) 0x18) return "btsp://";
else if(prefix == (byte) 0x19) return "bt12cap://";
else if(prefix == (byte) 0x1A) return "btgoep://";
else if(prefix == (byte) 0x1B) return "tcpobex://";
else if(prefix == (byte) 0x1C) return "irdaobex://";
else if(prefix == (byte) 0x1D) return "file://";
else if(prefix == (byte) 0x1E) return "urn:epc:id:";
else if(prefix == (byte) 0x1F) return "urn:epc:tag:";
else if(prefix == (byte) 0x20) return "urn:epc:pat:";
else if(prefix == (byte) 0x21) return "urn:epc:raw:";
else if(prefix == (byte) 0x22) return "urn:epc:";
else if(prefix == (byte) 0x23) return "urn:nfc:";
return null;
}

```

При указании абсолютного URI (тип `NdefRecord.TNF_ABSOLUTE_URI`) вся полезная нагрузка имеет кодировку UTF-8 и составляет весь URI:

```

if(record.getTnf() == NdefRecord.TNF_ABSOLUTE_URI) {
    String uri = new String(record.getPayload(), Charset.forName("UTF-8"));
}

```

Особый тип Smart Poster (`NdefRecord.RTD_SMART_POSTER`) состоит из множества подзаписей текста или данных URI (или абсолютных URI):

```

private void getTagData(final NdefMessage msg) {
    if(Arrays.equals(record.getType(), NdefRecord.RTD_SMART_POSTER)) {
        try {
            // выделяем подзаписи
            NdefMessage subrecords = new NdefMessage(record.getPayload());
            // получаем подзаписи
            String fulldata = getSubRecordData(subrecords);
            System.out.println("SmartPoster: "+fulldata);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```
// метод для получения информации подзаписей
private String getSubRecordData(final NdefRecord[] records) {
    if(records == null || records.length < 1) return null;
    String data = "";
    for(NdefRecord record : records) {
        if(record.getTnf() == NdefRecord.TNF_WELL_KNOWN) {
            if(Arrays.equals(record.getType(), NdefRecord.RTD_TEXT)) {
                data += getText(record.getPayload()) + "\n";
            }
            if(Arrays.equals(record.getType(), NdefRecord.RTD_URI)) {
                data += getURI(record.getPayload()) + "\n";
            } else {
                data += "OTHER KNOWN DATA\n";
            }
        } else if(record.getTnf() == NdefRecord.TNF_ABSOLUTE_URI) {
            data += getAbsoluteURI(record.getPayload()) + "\n";
        } else data += "OTHER UNKNOWN DATA\n";
    }
    return data;
}
```

Запись в метку

Уже в Android 2.3.3 (API уровня 10) появилась возможность записи данных в метку. Для этого используется объект Tag, который применяется для получения в метке подходящей TagTechnology. Метки коммуникации ближнего поля работают на основе независимо разработанных технологий и предлагают широкий спектр возможностей. Реализации TagTechnology обеспечивают доступ к этим различным технологиям и возможностям. В нашем случае технология NDEF требуется для получения и изменения в метке информации NdefRecords и NdefMessage:

```
// получение метки из намерения
Tag mytag = (Tag) intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);

// получение Ndef (TagTechnology) из метки
Ndef ndefref = Ndef.get(mytag);
```

При осуществлении операций ввода-вывода с применением TagTechnology необходимо соблюдать следующие требования.

- Перед использованием любой последующей операции ввода-вывода нужно вызывать метод connect().
- Операции ввода-вывода могут блокироваться, и их никогда не следует вызывать в основном потоке приложения.
- Единоновременно можно устанавливать соединение только с одной TagTechnology. Другие вызовы connect() будут выдавать исключение IOException.
- Метод close() следует вызывать после завершения операций ввода-вывода с TagTechnology. Этот метод будет отменять все прочие заблокированные операции ввода-вывода в других потоках (в том числе connect()), выдавая исключение IOException.

Итак, чтобы записать данные в метку, метод `connect()` вызывается из потока, работающего отдельно от главного потока. Когда это сделано, `isConnected()` следует проверить, чтобы убедиться, что соединение установлено. Если оно установлено, то можно вызвать метод `writeNdefMessage()` с созданным сообщением `NdefMessage` (содержащим не менее одной записи `NdefRecord`). После того как данные записаны, вызывается `close()` для завершения процесса.

Полный код для внесения текстовой записи в метку с использованием ссылки на NDEF TagTechnology таков:

// передаем ссылку на Ndef TagTechnology и текст, который хотим закодировать

```
private void writeTag(final Ndef ndefref, final String text) {
    if(ndefref == null || text == null || !ndefref.isWritable()) {
        return;
    }

    (new Thread() {
        public void run() {
            try {
                Message.obtain(mgsToaster, 0,
                    "Tag writing attempt started").sendToTarget();
                int count = 0;
                if(!ndefref.isConnected()) {
                    ndefref.connect();
                }
                while(!ndefref.isConnected()) {
                    if(count > 6000) {
                        throw new Exception("Unable to connect to tag");
                    }
                    count++;
                    sleep(10);
                }
                ndefref.writeNdefMessage(msg);
                Message.obtain(mgsToaster, 0,
                    "Tag write successful!").sendToTarget();
            } catch (Exception t) {
                t.printStackTrace();
                Message.obtain(mgsToaster, 0,
                    "Tag writing failed! - "+t.getMessage()).sendToTarget();
            } finally {
                // игнорируем ошибку закрытия...
                try { ndefref.close(); }
                catch (IOException e) { }
            }
        }
    }).start();
}
```

// создаем новую запись NdefRecord

```
private NdefRecord newTextRecord(String text) {
```

```

byte[] langBytes = Locale.ENGLISH.
    getLanguage().
    getBytes(Charset.forName("US-ASCII"));

byte[] textBytes = text.getBytes(Charset.forName("UTF-8"));

char status = (char) (langBytes.length);

byte[] data = new byte[1 + langBytes.length + textBytes.length];
data[0] = (byte) status;
System.arraycopy(langBytes, 0, data, 1, langBytes.length);
System.arraycopy(textBytes, 0, data, 1 + langBytes.length,
    textBytes.length);

return new NdefRecord(NdefRecord.TNF_WELL_KNOWN,
    NdefRecord.RTD_TEXT,
    new byte[0],
    data);
}

```

Одноранговый режим и Beam

Одноранговый режим (peer-to-peer mode, P2P) активируется в Android 2.3.3 (API уровня 10), когда одно устройство должно передавать данные другому по технологии коммуникации ближнего поля, причем второе устройство способно принимать данные, получаемые при такой коммуникации. Отправляющее устройство также может получать информацию с принимающего устройства, в этом и заключается одноранговая коммуникация. В API 10 это делалось методом выдвижения на передний план (foreground push). Но в более поздних версиях API (API 14, Android 4.0+) этот метод устарел. Вместо него используется более новый API для перевода на передний план, называемый Beam. В этом подразделе мы опишем оба метода.

API уровней 10–13

В API 10 для осуществления одноранговой коммуникации по NFC используется метод `enableForegroundNdefPush()` класса `NfcAdapter`. В результате активность для передачи сообщения `NdefMessage` (если она находится на переднем плане) может передать его другому устройству, приспособленному для коммуникации ближнего поля и поддерживающему протокол отправки NDEF `com.android.npp`. Метод `enableForegroundNdefPush()` следует вызывать из основного потока (как и в случае с `onResume()`). Этот метод должен отключаться, когда активность переходит на задний план (отключение происходит в методе `onPause()`):

```

@Override
public void onResume() {
    super.onResume();

    NdefRecord[] rec = new NdefRecord[1];
    rec[0] = new TextRecord("NFC Foreground Push Message");
    NdefMessage msg = new NdefMessage(rec);
}

```

```

    NfcAdapter.getDefaultAdapter(this).enableForegroundNdefPush(this, msg);
}

// создание новой NdefRecord
private NdefRecord newTextRecord(String text) {
    byte[] langBytes = Locale.ENGLISH.
        getLanguage().
        getBytes(Charset.forName("US-ASCII"));

    byte[] textBytes = text.getBytes(Charset.forName("UTF-8"));

    char status = (char) (langBytes.length);

    byte[] data = new byte[1 + langBytes.length + textBytes.length];
    data[0] = (byte) status;
    System.arraycopy(langBytes, 0, data, 1, langBytes.length);
    System.arraycopy(textBytes, 0, data, 1 + langBytes.length,
        textBytes.length);

    return new NdefRecord(NdefRecord.TNF_WELL_KNOWN,
        NdefRecord.RTD_TEXT,
        new byte[0],
        data);
}

```

Когда метод `enableForegroundNdefPush()` активен, стандартная диспетчеризация меток отключена. Лишь активность, находящаяся на переднем плане, может получать найденную по метке информацию с помощью метода `enableForegroundDispatch()`. Так гарантируется, что другие активности и службы не перехватят NFC-метку, сканируемую в настоящий момент. Поэтому данные получит та активность, которая работает на переднем плане.

Важно отметить, что, когда активность уходит с переднего плана (`onPause()`), вызывается метод `disableForegroundNdefPush()`:

```

@Override
protected void onPause() {
    super.onPause();
    if(NfcAdapter.getDefaultAdapter(this) != null) {
        NfcAdapter.getDefaultAdapter(this).disableForegroundNdefPush(this);
    }
}

```

Beam: API уровня 14 и выше

Для работы с Android Beam экран соответствующего устройства должен быть разблокирован, а на том устройстве, где инициируется такой «луч» (от англ. beam. — *Примеч. пер.*), рассматриваемая активность должна быть на переднем плане.

В API уровня 14 и выше механизм Android Beam обеспечивается путем использования двух методов `NfcAdapter`: `setNdefPushMessage()` и `setNdefPushMessageCallback()`. Метод `setNdefPushMessage()` принимает `NdefMessage` в качестве аргумента и сразу же отправляет это сообщение. В свою очередь, метод `setNdefPushMessageCallback()`

действует асинхронно и задействует интерфейс `NfcAdapter.CreateNdefMessageCallback`. Относящийся к этому интерфейсу метод `createNdefMessage()` вызывается, когда устройство попадает в диапазон коммуникации ближнего поля другого устройства. Если используются одновременно оба метода `pushMessage`, метод `setNdefPushMessageCallback()` имеет приоритет.

```
// здесь мы используем обратный вызов для отправки сообщения через NfcAdapter
NfcAdapter nfcadapter = NfcAdapter.getDefaultAdapter(this);
```

```
// здесь генерируется обратный вызов
CreateNdefMessageCallback nfccallback = new CreateNdefMessageCallback() {
```

```
    @Override
    public NdefMessage createNdefMessage(NfcEvent event) {
        String text = "Beaming via callback";
        byte[] mimeBytes =
            "application/com.oreilly.demo.android.pa.sensordemo".
        getBytes(Charset.forName("US-ASCII")); NdefRecord mimeRecord =
            new Ndef
            Record(NdefRecord.TNF_MIME_MEDIA, mimeBytes, new byte[0],
                text.getBytes());

        NdefMessage msg = new NdefMessage(new NdefRecord[] {mimeRecord});

        return msg;
    }
};
```

```
nfcadapter.setNdefPushMessageCallback(nfccallback, this);
```

```
// здесь мы просто напрямую выталкиваем сообщение
```

```
String directtext = "Beaming Directly";
byte[] directMimeBytes =
    "application/com.oreilly.demo.android.pa.sensordemo".
getBytes(Charset.forName("US-ASCII")); NdefRecord directMimeRecord =
    new NdefRecord
(NdefRecord.TNF_MIME_MEDIA, directMimeBytes, new byte[0],
    directtext.getBytes());
NdefMessage directmsg = new NdefMessage(new NdefRecord[]
    {directMimeRecord});
```

```
nfcadapter.setNdefPushMessage(directmsg, this);
```

В предыдущем примере запись Android Application Record (AAR) не используется. Поэтому в разделе файла описания для определения активности будет включен следующий фильтр намерений:

```
<activity android:name=".NFC40">
    <intent-filter>
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
```

```

        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType=
            "application/com.oreilly.demo.android.pa.sensordemo"/>
    </intent-filter>
</activity>

```

Настоятельно рекомендуем именно запись AAR, а не фильтр намерений, если вы собираетесь обрабатывать коммуникацию ближнего поля на уровне приложения (AAR не работает на уровне активности из-за ограничения, связанного с именем пакета). Таким образом, другие приложения не мешают обработке конкретного NFC-действия.

```
// генерируем NdefMessage с записью AAR
```

```

NdefMessage msg = new NdefMessage(new NdefRecord[] {mimeRecord,
    NdefRecord.createApplicationRecord(
        "com.oreilly.demo.android.pa.sensordemo")});

```

```
nfcadapter.setNdefPushMessage(msg, this);
```

Ввод жестов

В современном мире устройств с сенсорными экранами существует отличный способ сделать работу с устройством и простой и интересной. Для этого следует использовать сложные жесты (например, несколько движений пальцем по экрану в различных направлениях). Начиная с Android 1.6 (API уровня 4), мы можем пользоваться API для обработки жестов. Простейший способ добавить в приложение возможность ввода жестов при работе с этим API — использовать `android.gesture.GestureOverlayView`:

```
<!-- пример использования GestureOverlayView в файле layout.xml -->
```

```

<android.gesture.GestureOverlayView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gestures"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gestureStrokeType="multiple"
    android:eventsInterceptionEnabled="true">
</android.gesture.GestureOverlayView>

```

`GestureOverlayView` — это особый тип `FrameLayout`, который можно накладывать на другие виджеты либо который может содержать в себе другие виджеты. Данный макет способен запоминать штрихи, проводимые по экрану, а также отображать цветную линию (по умолчанию ее цвет — желтый), соответствующую траектории движения того или иного штриха. Интерфейс `GestureOverlayView.OnGesturePerformedListener` предоставляется для того, чтобы устройство могло реагировать на осуществляемые жесты:

```

GestureOverlayView gestures = (GestureOverlayView) findViewById(R.id.gestures);
gestures.addOnGesturePerformedListener(

```

```

        new GestureOverlayView.OnGesturePerformedListener() {
            @Override
            public void onGesturePerformed(GestureOverlayView overlay,
                                           Gesture gesture) {
                // пока ничего не делаем
            }
        });

```

После того как жест выполнен, вы можете посмотреть в библиотеке Gesture, распознается ли он. Информацию из библиотеки **Gesture можно считать разными способами**, при помощи статических методов класса GestureLibraries. После того как библиотека загружена (загружается GestureStore), выполненный жест можно передать и проанализировать при помощи метода recognize. Этот метод возвращает список объектов Prediction, в каждом из которых указываются очки и имя, причем очки отражают степень сходства выполненного жеста с тем или иным жестом, содержащимся в библиотеке:

```

final GestureLibrary library = GestureLibraries.fromFile("/Some/File/Path");
library.load(); // загрузка библиотеки

GestureOverlayView gestures = (GestureOverlayView)
    findViewById(R.id.gestures);
gestures.addOnGesturePerformedListener(
    new GestureOverlayView.OnGesturePerformedListener() {
        @Override
        public void onGesturePerformed(GestureOverlayView overlay,
                                       Gesture gesture) {
            // распознавание
            ArrayList<Prediction> predictions = library.recognize(gesture);
            if (predictions.size() > 0) {
                for(Prediction prediction: predictions) {
                    // очков достаточно много, и мы констатируем совпадение
                    if (prediction.score > 1.0) {
                        // отображаем всплывающее сообщение с названием жеста
                        Toast.makeText(this,
                                    prediction.name, Toast.LENGTH_SHORT).show();
                    }
                }
            }
        }
    });

```

Базовая анатомия Gesture состоит из множества объектов GestureStroke, и каждый объект GestureStroke составляется из множества объектов GesturePoint. Объект GesturePoint создается на базе координат x и y , а также из одной временной отметки (timestamp), указывающей, когда была сгенерирована точка. Когда Gesture сохраняется в GestureStore (в библиотеке GestureLibrary), он кодируется именем (String).

Добавить жест в библиотеку совсем не сложно. Вы сообщаете имя для ассоциирования жеста, а также объект Gesture, а потом сохраняете его в библиотеке. Обратите внимание: библиотека должна считываться с внешнего источника файлов

(например, с карты памяти или закрытого (приватного) файла), чтобы библиотека могла изменяться и, следовательно, выполнять функцию хранилища жестов. Библиотека, считываемая с необработанного ресурса, доступна только для чтения (используется `GestureLibraries.fromRawResource(context, resId)`):

```
public void saveGesture(String name, Gesture gesture) {
    library.addGesture(name, gesture);
    library.save();
}
```

Доступность

Начиная с Android 1.6 (API уровня 4), предоставляется специальный API, обеспечивающий возможность более широкого использования приложений Android слабовидящими и слепыми пользователями. Центральным элементом этого API, называемого интерфейсом доступности, является `AccessibilityService`, абстрактный класс, работающий в фоновом режиме.

В конечном итоге использование `AccessibilityService` подразумевает, что вам его придется дополнять. Следовательно, в файле описания потребуется объявить еще один сервис. Но требуется не просто сделать объявление. Кроме того, сервисы такого типа также должны обрабатывать особые намерения (`android.accessibilityservice.AccessibilityService`):

```
<service android:name=".Accessibility">
    <intent-filter>
        <action android:name=
            "android.accessibilityservice.AccessibilityService" />
    </intent-filter>
</service>
```

При создании класса `AccessibilityService` необходимо объявить тип отклика и тип события. Для этого генерируется объект `AccessibilityServiceInfo`, задающий различные переменные, а потом этот объект передается методу `setServiceInfo()`. Обратите внимание на то, что система будет собирать информацию только после выполнения привязки к классу/объекту:

```
AccessibilityServiceInfo info = new AccessibilityServiceInfo();
info.eventTypes = AccessibilityEvent.TYPES_ALL_MASK;
// задержка (мс) после новейшего события заданного типа перед тем,
// как вывести уведомление
info.notificationTimeout = 50;
info.feedbackType = AccessibilityServiceInfo.FEEDBACK_GENERIC |
    AccessibilityServiceInfo.FEEDBACK_AUDIBLE |
    AccessibilityServiceInfo.FEEDBACK_HAPTIC |
    AccessibilityServiceInfo.FEEDBACK_SPOKEN |
    AccessibilityServiceInfo.FEEDBACK_VISUAL;
info.packageNames = new String[1];
// обработать только этот пакет
info.packageNames[0] = getPackageName();
setServiceInfo(info);
```

После того как служба запущена и привязка выполнена, события будут приниматься и передаваться методу `onAccessibilityEvent()`:

```
@Override
public void onAccessibilityEvent(AccessibilityEvent event) {
    // здесь мы проверяем, произошло ли событие "щелчок"
    if(event.getEventType() == AccessibilityEvent.TYPE_VIEW_CLICKED) {
        // делаем что-либо со щелчком
    }
}
```

На данном этапе существуют различные варианты реагирования на событие. Обычно используется служба `VibratorService`, передающая осязаемый сигнал вместе с **голосовой или иной звуковой информацией**. **Вибратор** — это служба системного уровня, получаемая посредством контекстного метода `getSystemService()`. После того как объект `Vibrator` получен, можно применить при реагировании на событие определенную последовательность вибраций:

```
// получение Vibrator
Vibrator vibrate = (Vibrator) getSystemService(Service.VIBRATOR_SERVICE);
// последовательность вибраций
long[] pattern = new long[] { 0L, 100L };
// вибрация
vibrate.vibrate(pattern, -1)
```

В Android предоставляется движок `TextToSpeech`, которым можно пользоваться для передачи речи. Для его использования требуется инстанцировать класс `android.speech.tts.TextToSpeech`, инициализирующий движок `TextToSpeech`. После инициализации, чтобы передать речь, к этому классу необходимо применить метод `TextToSpeech`. Можно вызывать ряд методов и параметров, например задать настройки локали, высоты звука или скорости речи. Обязательно вызывайте метод `shutdown`, когда экземпляр `TextToSpeech` уже не нужен. Это делается, чтобы высвободить ресурсы:

```
TextToSpeech tts = new TextToSpeech(thisContext,
    new TextToSpeech.OnInitListener() {
        @Override
        public void onInit(int status) {
            // уведомление, выдаваемое при инициализации движка
            // TextToSpeech Engine
        }
    });

// говорим "щелк"
tts.speak("Click", 2, null);
// экземпляр нам больше не нужен, поэтому мы от него избавляемся
// и высвобождаем ресурсы
tts.shutdown();
```

Вопросы, связанные с доступностью, подробнее рассматриваются в свободном проекте `Eyes-Free` (<http://code.google.com/p/eyes-free>).

18 Коммуникация, личные данные, синхронизация и социальные сети

Одним из основных типов данных, которые сохраняются и используются в Android (причем многократно), является контактная информация. Это различные информационные фрагменты, связанные с контактом, — имя, телефонный номер, адрес электронной почты и т. д. В Android 2.0 (API уровня 5) концепция контактной информации была существенно расширена (появилась возможность доступа к нескольким учетным записям и поддержка агрегации похожих контактов). В предыдущих главах мы рассмотрели вопросы использования поставщиков содержимого и классов базы данных в Android, поэтому здесь мы не будем возвращаться к этому подготовительному материалу. Мы подробно поговорим о поставщике содержимого `ContactsContract`.

Контакты учетной записи

Для доступа к контактам учетной записи в файле описания следует прописать такие права доступа:

```
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.WRITE_CONTACTS" />
```

Внутри активности можно использовать метод `managedQuery` для запроса данных `ContactsContract.Contacts` и получения курсора для последующей работы:

```
private Cursor getContacts() {
    Uri uri = ContactsContract.Contacts.CONTENT_URI;

    String[] projection = new String[] {
        ContactsContract.Contacts._ID,
        ContactsContract.Contacts.LOOKUP_KEY,
        ContactsContract.Contacts.DISPLAY_NAME
    };

    String selection = null;
```

```
String[] selectionArgs = null;
String sortOrder = ContactsContract.Contacts.DISPLAY_NAME +
    " COLLATE LOCALIZED ASC";
return managedQuery(uri, projection, selection, selectionArgs,
    sortOrder);
}
```

Подробная информация о столбцах и константах, имеющихсся в классе `ContactsContract.Contacts`, приводится в документации для разработчиков по адресу <http://developer.android.com/reference/android/provider/ContactsContract.Contacts.html>.

Имея курсор, мы можем загрузить его внутри `SimpleCursorAdapter` и обеспечить отображение конкретных полей с данными, необходимых нам. В этом случае речь идет об отображаемом имени контакта:

```
String[] fields = new String[] {
    ContactsContract.Data.DISPLAY_NAME
};
SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
    R.layout.contact,
    cursor,
    fields,
    new int[] {R.id.name});

// получаем listview
ListView contactlist = (ListView) findViewById(R.id.contactlist);
// настраиваем адаптер и обеспечиваем отображение в нем
contactlist.setAdapter(adapter);
```

Вот макет, в котором содержится `ListView` (ссылка на него — `R.id.contactlist`):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#fff"
    >
    <ListView android:id="@+id/contactlist"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
    />
</LinearLayout>
```

Вот макет для контактной информации (ссылка на него — `R.layout.contact`), используемый с `SimpleCursorAdapter`:

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="#fff"
    >
    <TextView android:id="@+id/name"
        android:layout_width="fill_parent"
```

```

        android:layout_height="wrap_content"
        android:textColor="#000"
        android:textSize="25sp"
        android:padding="5dp"
    />
</LinearLayout>

```

Здесь мы удаляем контакт, сообщая курсор и позицию в курсоре, которую требуется удалить:

```

private void deleteContact(Cursor cursor, int position) {
    cursor.moveToPosition(position);
    long id = cursor.getLong(0);
    String lookupkey = cursor.getString(1);
    Uri uri = ContactsContract.Contacts.getLookupUri(id, lookupkey);

    String[] selectionArgs = null;
    String where = null;
    ContentResolver cr = getContentResolver();
    cr.delete(uri, where, selectionArgs);
}

```

Чтобы добавить контакт в этот пример, мы создаем коллекцию операций ContentProviderOperation и пакетно применяем их. Обратите внимание: сначала мы вставляем новый контакт, а потом добавляем телефонные номера, если они нам доступны (в данном случае такая информация у нас есть). Для осуществления вставки мы создаем операцию ContentProviderOperation отдельно для каждой вставки. При этом создается ContentProviderOperation.Builder с методом SimpleCursorContentProviderOperation.newInsert(), а потом выполняется построение при помощи метода build():

```

String accountNameWeWant = "SpecialAccount";

String phone = "8885551234";
String name = "Bob";

String accountname = null;
String accounttype = null;

Account[] accounts = AccountManager.get(this).getAccounts();

// Находим нужную нам учетную запись. Если она не находится,
// используем 'null' — значение, задаваемое по умолчанию.
for(Account account : accounts) {
    if(account.equals(accountNameWeWant)) {
        accountname = account.name;
        accounttype = account.type;
        break;
    }
}

ArrayList<ContentProviderOperation> ops =
    new ArrayList<ContentProviderOperation>();

```

```
ops.add(ContentProviderOperation.newInsert(
    ContactsContract.RawContacts.CONTENT_URI)
    .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, accountname)
    .withValue(ContactsContract.RawContacts.ACCOUNT_NAME, accounttype)
    .build());

// создаем новый контакт
ops.add(
    ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
    .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0)
    .withValue(ContactsContract.Data.MIMETYPE,
        ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE)
    .withValue(
        ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME, name)
    .build());

// если нам известен телефонный номер, добавляем его
if(phone.getText() != null
    && phone.getText().toString().trim().length() > 0) {
    ops.add(ContentProviderOperation.newInsert(
        ContactsContract.Data.CONTENT_URI)
        .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, 0)
        .withValue(ContactsContract.Data.MIMETYPE,
            ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE)
        .withValue(ContactsContract.CommonDataKinds.Phone.NUMBER,
            phone)
        .withValue(ContactsContract.CommonDataKinds.Phone.TYPE,
            ContactsContract.CommonDataKinds.Phone.TYPE_HOME)
        .build());
}

try {
    getContentResolver().applyBatch(ContactsContract.AUTHORITY, ops);
} catch (Exception e) {
    e.printStackTrace();
}
```

Аутентификация и синхронизация

В Android 2.0 (API уровня 5) и выше появилась возможность писать собственные поставщики синхронизации для интегрирования с системными контактами, календарями и т. д. В таком случае синхронизация с удаленной службой является, к сожалению, слишком рискованной авантюрой, поскольку малейший сбой на любом этапе этого процесса попросту вызовет крах системы Android и спровоцирует перезагрузку (причем практически невозможно будет понять, что именно пошло неправильно). Но по мере развития Android синхронизация также становится проще и надежнее. На данный момент этот процесс протекает в два этапа: аутентификация (аутентификатор учетных записей) и синхронизация (поставщик синхронизации).

Прежде чем приступить к детальному рассмотрению двух этих этапов, отметим, что примеры, которые мы здесь приводим, состоят из двух компонентов — серверной части и клиентской части на самом устройстве Android. Серверная часть, которой мы пользуемся, — это простая серверная служба, принимающая специфические запросы GET и выдающая ответ в формате JSON. Релевантный URI запроса GET, а также пример ответа мы приводим в каждой из рассматриваемых частей. В исходном коде, прилагаемом к этой книге, для полноты картины приведен весь серверный код.

Следует также отметить, что в приводимом нами примере мы решили синхронизироваться с контактами учетной записи. Но синхронизироваться можно не только с ними. Возможна синхронизация с любым поставщиком содержимого, к которому у вас есть доступ, и даже с сохраненными данными, специфичными для конкретного приложения.

Аутентификация

Чтобы клиент мог пройти аутентификацию на удаленном сервере при помощи системы аутентификации учетных записей, присутствующей в Android, необходимы три элемента.

- Служба, которая запускается намерением `android.accounts.AccountAuthenticator` и возвращающая в своем методе `onBind` подкласс `AbstractAccountAuthenticator`.
- Активность, приглашающая пользователя ввести свои учетные данные.
- XML-файл, описывающий, как должна выглядеть учетная запись, которая отображается пользователю.

Поговорим сначала о службе. В файле описания нам необходимо активировать `android.permission.AUTHENTICATE_ACCOUNTS`:

```
<uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
```

Потом служба должна быть охарактеризована в файле описания. Обратите внимание: намерение `android.accounts.AccountAuthenticator` включается в описатель `intent-filter`. Кроме того, в файле описания содержится информация о ресурсе для `AccountAuthenticator`:

```
<service android:name=".sync.authsync.AuthenticationService">
  <intent-filter>
    <action android:name="android.accounts.AccountAuthenticator" />
  </intent-filter>
  <meta-data android:name="android.accounts.AccountAuthenticator"
    android:resource="@xml/authenticator" />
</service>
```

Далее следует ресурс, который мы указали в файле описания. В частности, он описывает тип `accountType`, отличающий аутентификатор от других аутентификаторов, которые используют определение данной учетной записи. Аккуратно работайте с XML-документом (например, нельзя напрямую присваивать строку `android:label` или допускать отсутствие отрисовываемого объекта), поскольку в Android произойдет крах системы в тот самый момент, когда вы попытаетесь добавить новую учетную запись (из настроек учетной записи и синхронизации):

```
<?xml version="1.0" encoding="utf-8"?>

<account-authenticator xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:accountType="com.oreilly.demo.pa.ch17.sync"
    android:icon="@drawable/icon"
    android:smallIcon="@drawable/icon"
    android:label="@string/authlabel"
/>
```

Теперь, когда информация о службе есть в файле описания, можно заняться самой службой. Обратите внимание: метод `onBind()` возвращает класс `Authenticator`. Этот класс дополняет класс `AbstractAccountAuthenticator`:

```
package com.oreilly.demo.android.pa.clientserver.client.sync.authsync;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class AuthenticationService extends Service {
    private static final Object lock = new Object();
    private Authenticator auth;

    @Override
    public void onCreate() {
        synchronized (lock) {
            if (auth == null) {
                auth = new Authenticator(this);
            }
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return auth.getIBinder();
    }
}
```

Прежде чем перейти к рассмотрению полного исходного кода класса `Authenticator`, следует упомянуть один важный метод класса `AbstractAccountAuthenticator`. Он называется `addAccount()`. Этот метод вызывается именно в тот момент, когда на экране **Add Account** (Добавить учетную запись) мы выбираем кнопку, соответствующую нашей учетной записи. Активность `LoginActivity` (созданная нами, запрашивает у пользователя данные для входа) описывается в намерении (`Intent`), которое находится в возвращаемом пакете (`Bundle`). Очень важен ключ `AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE`, входящий в состав намерения, поскольку в нем находится объект `AccountAuthenticatorResponse`. Этот объект нужен, чтобы доставить обратно на устройство ключи учетных записей, после того как пользователь успешно пройдет сертификацию с удаленной службой.

```

public class Authenticator extends AbstractAccountAuthenticator {

    public Bundle addAccount(AccountAuthenticatorResponse response,
        String accountType, String authTokenType,
        String[] requiredFeatures, Bundle options) {

        Intent intent = new Intent(context, LoginActivity.class);
        intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,
            response);
        Bundle bundle = new Bundle();
        bundle.putParcelable(AccountManager.KEY_INTENT, intent);
        return bundle;
    }
}

```

Вот полный код активности `Authenticator`, дополняющей класс `AbstractAccountAuthenticator`:

```

package com.oreilly.demo.android.pa.clientserver.client.sync.authsync;

import com.oreilly.demo.android.pa.clientserver.client.sync.LoginActivity;
import android.accounts.AbstractAccountAuthenticator;
import android.accounts.Account;
import android.accounts.AccountAuthenticatorResponse;
import android.accounts.AccountManager;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;

public class Authenticator extends AbstractAccountAuthenticator {
    public static final String AUTHTOKEN_TYPE
        = "com.oreilly.demo.android.pa.clientserver.client.sync";
    public static final String ACCOUNT_TYPE
        = "com.oreilly.demo.android.pa.clientserver.client.sync";

    private final Context context;

    public Authenticator(Context context) {
        super(context);
        this.context = context;
    }

    @Override
    public Bundle addAccount(AccountAuthenticatorResponse response,
        String accountType, String authTokenType,
        String[] requiredFeatures, Bundle options) {

        Intent intent = new Intent(context, LoginActivity.class);
        intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,
            response);
        Bundle bundle = new Bundle();

```

```

        bundle.putParcelable(AccountManager.KEY_INTENT, intent);
        return bundle;
    }

    @Override
    public Bundle confirmCredentials(AccountAuthenticatorResponse response,
        Account account, Bundle options) {
        return null;
    }

    @Override
    public Bundle editProperties(AccountAuthenticatorResponse response,
        String accountType) {
        return null;
    }

    @Override
    public Bundle getAuthToken(AccountAuthenticatorResponse response,
        Account account, String authTokenType, Bundle loginOptions) {
        return null;
    }

    @Override
    public String getAuthTokenLabel(String authTokenType) {
        return null;
    }

    @Override
    public Bundle hasFeatures(AccountAuthenticatorResponse response,
        Account account, String[] features) {
        return null;
    }

    @Override
    public Bundle updateCredentials(AccountAuthenticatorResponse response,
        Account account, String authTokenType, Bundle loginOptions) {
        return null;
    }
}

```

В данном случае у удаленного сервера есть метод API входа в систему (доступный посредством HTTP URI), принимающий имя пользователя и пароль в качестве переменных. Если вход в систему пройдет успешно, ответ вернется со строкой в формате JSON и в этой строке будет содержаться метка (token):

```
uri: http://<serverBaseUrl>:<port>/login?username=<name>&password=<pass>
```

```
response: { "token" : „someAuthenticationToken" }
```

Активность `LoginActivity`, требующая от пользователя ввести имя и пароль к учетной записи, затем устанавливает контакт с удаленным сервером. После того

как будет возвращена ожидаемая строка в формате JSON, вызывается метод `handleLoginResponse()`, передающий важную информацию об учетной записи обратно к `AccountManager`:

```
package com.oreilly.demo.android.pa.clientserver.client.sync;

import org.json.JSONObject;

import com.oreilly.demo.android.pa.clientserver.client.R;
import com.oreilly.demo.android.pa.clientserver.client.sync.authsync.Authenticator;
import android.accounts.Account;
import android.accounts.AccountAuthenticatorActivity;
import android.accounts.AccountManager;
import android.app.Dialog;
import android.app.ProgressDialog;
import android.content.ContentResolver;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.provider.ContactsContract;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.EditText;
import android.widget.Toast;

public class LoginActivity extends AccountAuthenticatorActivity {
    public static final String PARAM_AUTHTOKEN_TYPE = "authtokenType";
    public static final String PARAM_USERNAME = "username";
    public static final String PARAM_PASSWORD = "password";

    private String username;
    private String password;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getVars();
        setupView();
    }

    @Override
    protected Dialog onCreateDialog(int id) {
        final ProgressDialog dialog = new ProgressDialog(this);
        dialog.setMessage("Attempting to login");
        dialog.setIndeterminate(true);
        dialog.setCancelable(false);
        return dialog;
    }

    private void getVars() {
```

```

        username = getIntent().getStringExtra(PARAM_USERNAME);
    }

    private void setupView() {
        setContentView(R.layout.login);

        findViewById(R.id.login).setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                login();
            }
        });

        if(username != null) {
            ((EditText) findViewById(R.id.username)).setText(username);
        }
    }

    private void login() {
        if(((EditText) findViewById(R.id.username)).getText() == null ||
            ((EditText) findViewById(R.id.username)).getText().toString().
                trim().length()
                < 1) {
            Toast.makeText(this, "Please enter a Username",
                Toast.LENGTH_SHORT).show();
            return;
        }
        if(((EditText) findViewById(R.id.password)).getText() == null ||
            ((EditText) findViewById(R.id.password)).getText().toString().
                trim().length()
                < 1) {
            Toast.makeText(this, "Please enter a Password",
                Toast.LENGTH_SHORT).show();
            return;
        }

        username = ((EditText)
            findViewById(R.id.username)).getText().toString();
        password = ((EditText)
            findViewById(R.id.password)).getText().toString();

        showDialog(0);

        Handler loginHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                if(msg.what == NetworkUtil.ERR) {
                    dismissDialog(0);
                    Toast.makeText(LoginActivity.this, "Login Failed: "+
                        msg.obj, Toast.LENGTH_SHORT).show();
                } else if(msg.what == NetworkUtil.OK) {

```

```

        handleLoginResponse((JSONObject) msg.obj);
    }
}

NetworkUtil.login(getString(R.string.baseurl),
    username, password, loginHandler);
}

private void handleLoginResponse(JSONObject resp) {
    dismissDialog(0);

    final Account account = new Account(username,
        Authenticator.ACCOUNT_TYPE);

    if (getIntent().getStringExtra(PARAM_USERNAME) == null) {
        AccountManager.get(this).addAccountExplicitly(account, password,
            null);
        ContentResolver.setSyncAutomatically(account,
            ContactsContract.AUTHORITY, true);
    } else {
        AccountManager.get(this).setPassword(account, password);
    }

    Intent intent = new Intent();
    intent.putExtra(AccountManager.KEY_ACCOUNT_NAME, username);
    intent.putExtra(AccountManager.KEY_ACCOUNT_TYPE,
        Authenticator.ACCOUNT_TYPE);
    if (resp.has("token")) {
        intent.putExtra(AccountManager.KEY_AUTHTOKEN,
            resp.optString("token"));
    }
    setAccountAuthenticatorResult(intent.getExtras());
    setResult(RESULT_OK, intent);
    finish();
}
}

```

XML макета активности LoginActivity таков:

```

<?xml version="1.0" encoding="utf-8" ?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background="#fff">
    <ScrollView
        android:layout_width="fill_parent"
        android:layout_height="0dip"
        android:layout_weight="1">
        <LinearLayout

```

```

        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:orientation="vertical"
        android:paddingTop="5dip"
        android:paddingBottom="13dip"
        android:paddingLeft="20dip"
        android:paddingRight="20dip">
        <EditText
            android:id="@+id/username"
            android:singleLine="true"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:minWidth="250dip"
            android:scrollHorizontally="true"
            android:capitalize="none"
            android:hint="Username"
            android:autoText="false" />
        <EditText
            android:id="@+id/password"
            android:singleLine="true"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:minWidth="250dip"
            android:scrollHorizontally="true"
            android:capitalize="none"
            android:autoText="false"
            android:password="true"
            android:hint="Password"
            android:inputType="textPassword" />
    </LinearLayout>
</ScrollView>
<FrameLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background="#fff"
    android:minHeight="54dip"
    android:paddingTop="4dip"
    android:paddingLeft="2dip"
    android:paddingRight="2dip">
    <Button
        android:id="@+id/login"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:minWidth="100dip"
        android:text="Login" />
</FrameLayout>
</LinearLayout>

```

Итак, учетная запись создана, и мы можем использовать ее для синхронизации данных.

Синхронизация

Синхронизация данных определенной учетной записи опять же состоит из трех частей: службы, регистрируемой для слушания намерения `android.content.SyncAdapter` и возвращающей методу `onBind()` дополненный класс `AbstractThreadedSyncAdapter`; описателя XML, сообщающего информацию о структуре тех данных, которые будут синхронизироваться и просматриваться; и, наконец, класса, дополняющего `AbstractThreadedSyncAdapter`, — он и занимается самой синхронизацией.

В нашем примере мы собираемся синхронизировать контактную информацию с учетной записью, описанной в предыдущем разделе. Обратите внимание: синхронизировать можно не только контактную информацию. Можно синхронизироваться с любым поставщиком содержимого, к которому у вас есть доступ, и даже со специфичными для конкретного приложения сохраненными данными.

В файле описания прописываются следующие права доступа:

```
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.WRITE_CONTACTS" />
<uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
<uses-permission android:name="android.permission.USE_CREDENTIALS" />
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WRITE_SETTINGS" />
<uses-permission android:name="android.permission.WRITE_SECURE_SETTINGS" />
<uses-permission android:name="android.permission.READ_SYNC_STATS" />
<uses-permission android:name="android.permission.READ_SYNC_SETTINGS" />
<uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS" />
```

Теперь опишем службу, с которой мы собираемся работать. Обратите внимание на то, что здесь присутствует намерение `android.content.SyncAdapter`, а также указывается структура контактных данных и сам адаптер `SyncAdapter`:

```
<service android:name=".sync.authsync.SyncService">
  <intent-filter>
    <action android:name="android.content.SyncAdapter" />
  </intent-filter>
  <meta-data android:name="android.content.SyncAdapter"
    android:resource="@xml/syncadapter" />
  <meta-data android:name="android.provider.CONTACTS_STRUCTURE"
    android:resource="@xml/contacts" />
</service>
```

В XML-ресурсе для `sync-adapter` обратите внимание на описатель `accountType`. Содержимое, с которым мы собираемся работать, — это информация о контактах Android:

```
<?xml version="1.0" encoding="utf-8"?>

<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"
  android:contentAuthority="com.android.contacts"
  android:accountType="com.oreilly.demo.android.pa.clientserver.client.sync"
/>
```

Вот XML для описателя контактов. Обратите внимание на названия различных столбцов, описанных нами:

```
<?xml version="1.0" encoding="utf-8"?>
<ContactsSource xmlns:android="http://schemas.android.com/apk/res/android">

    <ContactsDataKind
        android:mimeType=
            "vnd.android.cursor.item/
                vnd.com.oreilly.demo.android.pa.clientserver.sync.profile"
        android:icon="@drawable/icon"
        android:summaryColumn="data2"
        android:detailColumn="data3"
        android:detailSocialSummary="true" />

</ContactsSource>
```

Созданная нами служба SyncService возвращает класс SyncAdapter. Это созданный нами класс, дополняющий AbstractThreadedSyncAdapter:

```
package com.oreilly.demo.android.pa.clientserver.client.sync.authsync;
```

```
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class SyncService extends Service {
    private static final Object lock = new Object();
    private static SyncAdapter adapter = null;

    @Override
    public void onCreate() {
        synchronized (lock) {
            if (adapter == null) {
                adapter = new SyncAdapter(getApplicationContext(), true);
            }
        }
    }

    @Override
    public void onDestroy() {
        adapter = null;
    }

    @Override
    public IBinder onBind(Intent intent) {
        return adapter.getSyncAdapterBinder();
    }
}
```

Продолжая данное упражнение, создаем метод getFriends на стороне удаленного сервера. Он принимает метку, переданную нам обратно с сервера и сохраненную

в результате успешного входа в систему, — об этом мы говорили выше. Он также передает время, указывающее, когда был сделан последний вызов (если вызов был сделан впервые — передается значение 0). Ответ представляет собой еще одну строку в формате JSON, описывающую друзей (в том числе сообщаящую их ID, имя и номер телефона) и указывающую время, в которое был сделан вызов (или Unix-время на сервере). Здесь также приводится история добавления или удаления друзей в данной учетной записи. В истории в поле `type` может стоять значение 0 или 1. Первое соответствует добавлению друга, второе — удалению. Поле `who` содержит ID друга, а поле `time` сообщает, когда была произведена операция:

uri: `http://<serverBaseUrl>:<port>/getfriends?token=<token>&time=<lasttime>`

response:

```
{
  „time“ : 1295817666232,
  „history“ : [
    {
      „time“ : 1295817655342,
      „type“ : 0,
      „who“ : 1
    }
  ],
  „friend“ : [
    {
      „id“ : 1,
      „name“ : „Mary“,
      „phone“ : „8285552334“
    }
  ]
}
```

Далее следует класс `AbstractThreadedSyncAdapter`, дополняющий `SyncAdapter`:

```
public class SyncAdapter extends AbstractThreadedSyncAdapter {
    private final Context context;

    private static long lastsynctime = 0;

    public SyncAdapter(Context context, boolean autoInitialize) {
        super(context, autoInitialize);
        this.context = context;
    }

    @Override
    public void onPerformSync(Account account, Bundle extras, String
        authority, ContentProviderClient provider, SyncResult
        syncResult) {
        String authToken = null;
        try {
            authToken = AccountManager.get(
```

```

        context).blockingGetAuthToken(account,
        Authenticator.AUTHTOKEN_TYPE, true);

    ListFriends friendsdata =
        ListFriends.fromJSON(
            NetworkUtil.getFriends(context.getString(
                R.string.baseurl), authtoken, lastsynctime, null));

    lastsynctime = friendsdata.time;

    sync(account, friendsdata);
} catch (Exception e) {
    e.printStackTrace();
}
}

private void sync(Account account, ListFriends data) {
    // ТУТ ПРОИСХОДИТ ВОЛШЕБСТВО
}
}

```

Далее приведен весь класс SyncAdapter, содержащий различные действия, которые происходят, когда метод синхронизации получает данные. Здесь же включены различные добавления и удаления контактной информации (операции Contact и ContentProvider были рассмотрены в предыдущих главах и разделах):

```

package com.oreilly.demo.android.pa.clientserver.client.sync.authsync;

import java.util.ArrayList;

import android.accounts.Account;
import android.accounts.AccountManager;
import android.content.AbstractThreadedSyncAdapter;
import android.content.ContentProviderClient;
import android.content.ContentProviderOperation;
import android.content.ContentUris;
import android.content.Context;
import android.content.SyncResult;
import android.database.Cursor;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.provider.ContactsContract.RawContacts;

import com.oreilly.demo.android.pa.clientserver.client.R;
import com.oreilly.demo.android.pa.clientserver.client.sync.NetworkUtil;
import com.oreilly.demo.android.pa.clientserver.client.sync.dataobjects.Change;
import com.oreilly.demo.android.pa.clientserver.client.sync.dataobjects.ListFriends;
import com.oreilly.demo.android.pa.clientserver.client.sync.dataobjects.User;

public class SyncAdapter extends AbstractThreadedSyncAdapter {
    private final Context context;

```

```

private static long lastsynctime = 0;

public SyncAdapter(Context context, boolean autoInitialize) {
    super(context, autoInitialize);
    this.context = context;
}

@Override
public void onPerformSync(Account account, Bundle extras, String
    authority, ContentProviderClient provider,
    SyncResult syncResult) {
    String authtoken = null;
    try {
        // Получаем метку учетной записи. В итоге это приводит
        // к вызову нашего аутентификатора getAuthToken().
        authtoken = AccountManager.get(
            context).blockingGetAuthToken(account,
            Authenticator.AUTHTOKEN_TYPE, true);

        ListFriends friendsdata =
            ListFriends.fromJSON(
                NetworkUtil.getFriends(
                    context.getString(R.string.baseurl),
                    authtoken, lastsynctime, null));

        lastsynctime = friendsdata.time;

        sync(account, friendsdata);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// здесь происходит все волшебство
private void sync(Account account, ListFriends data) {
    User self = new User();
    self.username = account.name;

    ArrayList<ContentProviderOperation> ops =
        new ArrayList<ContentProviderOperation>();

    // цикл, проходящий через историю и находящий операции удаления
    if(data.history != null && !data.history.isEmpty()) {
        for(Change change : data.history) {
            if(change.type == Change.ChangeType.DELETE) {
                ContentProviderOperation op = delete(account,
                    change.who);
                if(op != null) ops.add(op);
            }
        }
    }
}

```

```

// цикл, проходящий по списку друзей, отыскивающий тех,
// кого мы еще не добавили, и добавляющий их
if(data.friends != null && !data.friends.isEmpty()) {
    for(User f : data.friends) {
        ArrayList<ContentProviderOperation> op =
            add(account, f);
        if(op != null) ops.addAll(op);
    }
}

if(!ops.isEmpty()) {
    try {
        context.getContentResolver().applyBatch(
            ContactsContract.AUTHORITY, ops);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

// Добавление контакта. Обратите внимание – мы уже сохранили id, на который
// стоит ссылка в ответе, полученном с сервера, и этот id сохранен
// в поле SYNC1 – так мы сможем найти его по серверному id.
private ArrayList<ContentProviderOperation> add(Account account,
        User f) {
    long rawid = lookupRawContact(f.id);

    if(rawid != 0) return null;
    ArrayList<ContentProviderOperation> ops =
        new ArrayList<ContentProviderOperation>();
    ops.add(ContentProviderOperation.newInsert(
        ContactsContract.RawContacts.CONTENT_URI)
        .withValue(RawContacts.SOURCE_ID, 0)
        .withValue(RawContacts.SYNC1, f.id)
        .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE,
            Authenticator.ACCOUNT_TYPE)
        .withValue(ContactsContract.RawContacts.ACCOUNT_NAME,
            account.name)
        .build());

    if(f.name != null && f.name.trim().length() > 0) {
        ops.add(ContentProviderOperation.newInsert(
            ContactsContract.Data.CONTENT_URI)
            .withValueBackReference(
                ContactsContract.Data.RAW_CONTACT_ID, 0)
            .withValue(ContactsContract.Data.MIMETYPE,
                ContactsContract.CommonDataKinds.
                    StructuredName.CONTENT_ITEM_TYPE)
            .withValue(ContactsContract.CommonDataKinds.
                StructuredName.DISPLAY_NAME, f.name)
            .build());
    }
}

```

```

        if(f.phone != null && f.phone.trim().length() > 0) {
            ops.add(ContentProviderOperation.newInsert(
                ContactsContract.Data.CONTENT_URI)
                .withValueBackReference(
                    ContactsContract.Data.RAW_CONTACT_ID, 0)
                .withValue(ContactsContract.Data.MIMETYPE,
                    ContactsContract.CommonDataKinds.
                        Phone.CONTENT_ITEM_TYPE)
                .withValue(ContactsContract.
                    CommonDataKinds.Phone.NUMBER, f.phone)
                .withValue(ContactsContract.
                    CommonDataKinds.Phone.TYPE,
                    ContactsContract.CommonDataKinds.
                        Phone.TYPE_HOME)
                .build());
        }

        ops.add(ContentProviderOperation.newInsert(
            ContactsContract.Data.CONTENT_URI)
            .withValueBackReference(
                ContactsContract.Data.RAW_CONTACT_ID, 0)
            .withValue(ContactsContract.Data.MIMETYPE,
                "vnd.android.cursor.item/vnd.com.
                    oreilly.demo.android.pa.clientserver.client.sync.profile")
            .withValue(ContactsContract.Data.DATA2, "Ch15 Profile")
            .withValue(ContactsContract.Data.DATA3, "View profile")
            .build()
            );
        return ops;
    }

    // Удаление контакта по id, указанному на сервере
    private ContentProviderOperation delete(Account account, long id) {
        long rawid = lookupRawContact(id);
        if(rawid == 0) return null;
        return ContentProviderOperation.newDelete(
            ContentUris.withAppendedId(
                ContactsContract.RawContacts.CONTENT_URI,
                rawid))
            .build();
    }

    // поиск существующего необработанного id посредством id,
    // сохраненного нами в поле SYNC1
    private long lookupRawContact(long id) {
        long rawid = 0;
        Cursor c = context.getContentResolver().query(
            RawContacts.CONTENT_URI, new String[] {RawContacts._ID},
            RawContacts.ACCOUNT_TYPE + "=" +
                Authenticator.ACCOUNT_TYPE + " AND " +
                RawContacts.SYNC1 + "=?",
            new String[] {String.valueOf(id)},
            null);
    }

```

```

    try {
        if(c.moveToFirst()) {
            rawid = c.getLong(0);
        }
    } finally {
        if (c != null) {
            c.close();
            c = null;
        }
    }
    return rawid;
}
}

```

В предыдущем классе SyncAdapter легко упустить важную деталь. Она заключается в следующем: во время вызова `onPerformSync()` мы пытаемся получить `authToken` от `AccountManager` и пользуемся при этом методом `blockingGetAuthToken()`. В итоге это приводит к вызову `AbstractAccountAuthenticator`, связанного с данной учетной записью. В этом случае происходит вызов класса `Authenticator`, рассмотренного нами в предыдущем разделе. В классе `Authenticator` вызывается метод `getAuthToken()`. Рассмотрим эту ситуацию на примере:

```

@Override
public Bundle getAuthToken(AccountAuthenticatorResponse response, Account
    account, String authTokenType, Bundle loginOptions) {
    // проверяем и убеждаемся, что тип метки – именно такой, как нам требуется
    if (!authTokenType.equals(AUTHTOKEN_TYPE)) {
        final Bundle result = new Bundle();
        result.putString(AccountManager.KEY_ERROR_MESSAGE,
            "invalid authTokenType");
        return result;
    }
    // Если у нас есть пароль, попробуем получить действующую метку
    // аутентификации с сервера.
    String password = AccountManager.get(context).getPassword(account);
    if (password != null) {
        JSONObject json = NetworkUtil.login(
            context.getString(R.string.baseurl),
            account.name, password, true, null);
        if(json != null) {
            Bundle result = new Bundle();
            result.putString(AccountManager.KEY_ACCOUNT_NAME, account.name);
            result.putString(AccountManager.KEY_ACCOUNT_TYPE, ACCOUNT_TYPE);
            result.putString(AccountManager.KEY_AUTHTOKEN,
                json.optString("token"));
            return result;
        }
    }
    // Если все остальное не получается, попробуем показать активность для входа.
    Intent intent = new Intent(context, LoginActivity.class);
    intent.putExtra(LoginActivity.PARAM_USERNAME, account.name);
    intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,

```

```
        response);  
    Bundle bundle = new Bundle();  
    bundle.putParcelable(AccountManager.KEY_INTENT, intent);  
    return bundle;  
}
```

Bluetooth

Название этой технологии в переводе на русский язык означает «Синезубый». Это было прозвище короля Харальда Датского. В статье на сайте разработчиков компании Sun (<http://developers.sun.com/mobility/midp/articles/bluetooth1/>) подробно рассказывается об этой технологии и есть в том числе анекдотическое замечание о том, что на руническом камне, воздвигнутом в честь Харальда, были высечены такие слова:

Харальд крестил данов.

Харальд покорил Данию и Норвегию.

Харальд считает, что коммуникация между мобильными телефонами и ноутбуками должна идти как по маслу.

Чтобы продемонстрировать, как в Android используются классы, обеспечивающие связь по технологии Bluetooth, мы напишем утилиту для установки соединения и передачи данных между устройствами, поддерживающими Bluetooth. Этот код основан на примере BluetoothChat из комплекта Android SDK. Данный код был сделан более универсальным, чтобы охватить более широкий спектр приложений с Bluetooth, а также упрощен, чтобы вам было удобнее адаптировать его для своих целей.

Исследуя различные API Android для работы с Bluetooth, мы рассмотрим, как эти API задействуются в рассматриваемом коде и как их можно использовать для целей, связанных с конкретным приложением, в том числе в диагностическом инструменте, применяемом в ходе разработки для Bluetooth.

Сначала рассмотрим, как работает Bluetooth и как эта технология реализована в Android.

Стек протоколов Bluetooth

Этот подраздел посвящен стандартам и протоколам, из которых состоит стек протоколов Bluetooth (рис. 18.1). Эти протоколы и стандарты исчерпывающе характеризуют Bluetooth: те виды данных, для передачи которых предназначен Bluetooth, количество устройств, между которыми можно одновременно установить связь, время задержки при передаче и т. д.

Bluetooth появился как особый род сетевых взаимодействий, поскольку он представляет собой персональную, или личную, сеть. Такие сети также называются аббревиатурой PAN, или пикосетью. Bluetooth предназначен для соединения до восьми устройств и для передачи данных с максимальной скоростью до 3 Мбит/с. Соединенные устройства должны располагаться близко друг к другу: в пределах 10 м. Bluetooth потребляет очень малое количество энергии, исчисляемое милливаттами. Это означает, что даже самой маленькой батарее надолго хватает для такой

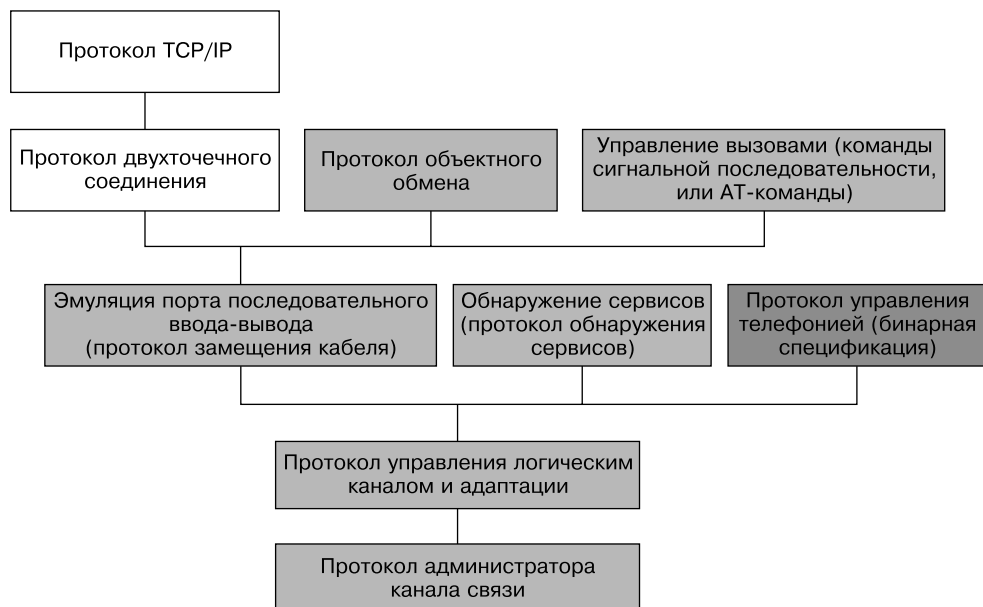


Рис. 18.1. Стек протоколов Bluetooth в Android

связи. Наушники Bluetooth с крошечной, невесомой батареей могут часами обеспечивать голосовую связь — примерно столько же выдерживает гораздо более крупная батарея мобильного телефона, поскольку мобильный радиосигнал с телефона должен достигать значительно более отдаленной антенны.

К приборам, с которыми полезно использовать Bluetooth, относятся устройства с низкой и средней скоростью передачи данных — в частности, клавиатуры, мыши, планшеты, принтеры, микрофоны, наушники (в том числе с микрофонами), а также мобильные и персональные компьютерные устройства, если с их периферийными компонентами может потребоваться связь. Bluetooth также позволяет устанавливать соединения между ПК и мобильными телефонами.

Протоколы, специфичные для Bluetooth, и заимствованные протоколы

Говоря о стеке протоколов Bluetooth, важно разделять эти протоколы на специфичные для Bluetooth и заимствованные, то есть такие, которые работают «поверх» Bluetooth. Вместе все эти протоколы — как относящиеся непосредственно к Bluetooth, так и заимствованные — могут быть очень сложны. Но если на минуту абстрагироваться от того, что над Bluetooth находятся такие сложные протоколы, как OBEX и TCP/IP, то все становится гораздо понятнее. Поэтому мы начнем с более низких уровней Bluetooth и покажем, как эти уровни помогают «оформить» нашу работу с Bluetooth.

Еще одна важная абстракция при работе с Bluetooth сводится к тому, что эта технология заменяет порты последовательного ввода-вывода. Это означает, что нижние уровни Bluetooth эмулируют виртуальный набор последовательных кабелей между соединяемыми устройствами и позволяют вам управлять этими кабе-

лями. Именно с таким протоколом Bluetooth мы и будем работать. Таким образом, мы сможем пользоваться для считывания и записи данных простыми классами ввода-вывода `java.io: InputStream` и `OutputStream`.

Bluez: реализация Bluetooth для Linux

Мобильное устройство может соединяться по Bluetooth с какими угодно другими устройствами — чего не скажешь о периферийных устройствах, которые могут подключаться только к компьютеру или мобильному устройству. Это означает, что на мобильном устройстве нам потребуется практически полная реализация Bluetooth и всех заимствованных протоколов, а также пользовательский интерфейс, обеспечивающий необходимые взаимодействия для установления и использования соединений, а также для работы в приложениях, связывающихся по Bluetooth.

В Android используется стек `Bluez Bluetooth` — наиболее распространенный стек для работы с Bluetooth, применяемый в Linux. Он пришел на смену проекту, называвшемуся `Open BT`. Информация о `Bluez` содержится на сайте проекта `Bluez` — <http://www.bluez.org>.

`Bluez`, разработанный в `Qualcomm`, вошел в состав ядра Linux. Проект начался в 2001 году и до сих пор активен, хорошо поддерживается. Кроме того, `Bluez` — это стабильная реализация, обладающая хорошей совместимостью, — еще одна причина, по которой стоит использовать Linux в операционных системах для мобильных устройств.

Использование Bluetooth в приложениях Android

Под использованием Bluetooth в Android понимается применение классов, специально разработанных для того, чтобы инкапсулировать принцип, по которому Bluetooth работает в операционной системе Android. Стек `Bluez` предоставляет методы для перечисления устройств, слушания соединений и использования соединений. В пакете `java.io` предоставляются классы для считывания и записи данных; а классы `Handler` и `Message` обеспечивают возможности построения связей (мостов) между пользовательским интерфейсом, с одной стороны, и потоками, управляющими вводом и выводом в Bluetooth, — с другой. Рассмотрим код и примеры использования этих классов.

Компиляция и запуск кода помогают понять, что могут делать классы Bluetooth в приложениях Android, предназначенных для установления связи между устройствами, которые расположены вблизи друг от друга.

Начнем испытание программы, использующей Bluetooth, с самого простого: попробуем соединить мобильный телефон с персональным компьютером. Затем нам понадобится программа, которая наблюдала бы за той информацией, которую получает ПК по Bluetooth, и следила, какая именно посланная вами информация дошла до ПК. В данном случае применяется утилита Linux, называемая `hcidump`.

Запустите программу в режиме отладки и установите точки останова в частях приложения, которые занимаются открытием и принятием соединения. Можно создать соединение со своего ПК (в Linux для этого используется апплет `Bluemanager`) или же из приложения. После создания соединения запустите на терминале программу `hcidump` и убедитесь, что информация, которую вы ввели, попала на ПК.

Пользуйтесь перечисленными ниже флагами, чтобы отображать только то содержимое, которое поступает по Bluetooth:

```
sudo hcidump -a -R
```

Теперь информация, которую вы послали с устройства, должна отобразиться в качестве вывода hcidump на вашем ПК.

Bluetooth и связанные с ним классы ввода/вывода

Эта программа при работе опирается на класс `BluetoothAdapter`, обеспечивающий управление адаптером Bluetooth на устройстве, класс `BluetoothDevice`, который представляет состояние подключенного устройства, и класс `BluetoothSocket`, предоставляющий сокет для слушания и установления соединений:

```
package com.finchframework.bluetooth;

import android.os.Handler;
import android.os.Message;

public class BtHelperHandler extends Handler {

    public enum MessageType {
        STATE,
        READ,
        WRITE,
        DEVICE,
        NOTIFY;
    }

    public Message obtainMessage(MessageType message, int count,
                                Object obj) {
        return obtainMessage(message.ordinal(), count, -1, obj);
    }

    public MessageType getMessageType(int ordinal) {
        return MessageType.values()[ordinal];
    }
}
```

В классе `BtHelperHandler` определяется ряд констант, а также предоставляется немного оберточного кода (wrapper code), который позволяет сделать чище методы, связанные с передачей сообщений.

Класс `BtSPPHelper.java` инкапсулирует работу с протоколом Bluetooth SPP (протокол последовательного порта):

```
package com.finchframework.bluetooth;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.UUID;
```

```

import com.finchframework.finch.R;

import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothServerSocket;
import android.bluetooth.BluetoothSocket;
import android.content.Context;
import android.os.Bundle;
import android.os.Message;
import android.util.Log;

/**
 * Вспомогательный класс, запускающий объекты AsyncTask для коммуникации
 * с устройством, поддерживающим Bluetooth.
 * Этот код построен на основе примера с Bluetoothchat, но несколько
 * изменен с целью повысить его модульность и универсальность:
 * Handler – это отдельный класс, облегчающий переходы к другим компонентам.
 *
 * В настоящее время соединение идет только по протоколу Bluetooth SPP.
 * Пример может быть приведен в более общий вид для работы и с другими службами.
 */
public class BtSPPHelper {
    // отладка
    private final String TAG = getClass().getSimpleName();
    private static final boolean D = true;

    public enum State {
        NONE,
        LISTEN,
        CONNECTING,
        CONNECTED;
    }

    // имя записи SDP при создании серверного сокета
    private static final String NAME = "BluetoothTest";

    // универсальный уникальный идентификатор для этого приложения
    private static final UUID SPP_UUID =
        UUID.fromString("00001101-0000-1000-8000-00805F9B34FB");

    // поля компонентов
    private final BluetoothAdapter mAdapter;
    private final BtHelperHandler mHandler;
    private AcceptThread mAcceptThread;
    private ConnectThread mConnectThread;
    private ConnectedThread mConnectedThread;
    private State mState;
    private Context mContext;

    /**
     * Конструктор. Подготавливает новую сессию Bluetooth SPP
     * @param context контекст активности пользовательского интерфейса

```

```

    * @param handler обработчик, отсылающий сообщения обратно
    * к активности пользовательского интерфейса
    */
    public BtSPPHelper(Context context, BtHelperHandler handler) {
        mContext = context;
        mAdapter = BluetoothAdapter.getDefaultAdapter();
        mState = State.NONE;
        mHandler = handler;
    }

    /**
     * Задаёт актуальное состояние чат-соединения.
     * @param state Актуальное состояние соединения
     */
    private synchronized void setState(State state) {
        if (D) Log.d(TAG, "setState() " + mState + " -> " + state);
        mState = state;

        // Задаём новое состояние для Handler, чтобы активность
        // пользовательского интерфейса могла обновиться.
        mHandler.obtainMessage(BtHelperHandler.MessageType.STATE,
                               -1, state).sendToTarget();
    }

    /**
     * Возвращаем актуальное состояние соединения.
     */
    public synchronized State getState() {
        return mState;
    }

    /**
     * Запуск сессии. Запуск AcceptThread для того,
     * чтобы начать соединение в слушающем (серверном) режиме.
     *
     * Как правило, вызывается onResume().
     */
    public synchronized void start() {
        if (D) Log.d(TAG, "start");

        // Отменяем любой поток, который пытается установить соединение.
        if (mConnectThread != null) {mConnectThread.cancel();
                                     mConnectThread = null;}

        // Отменяем любой поток, который в настоящее время поддерживает соединение.
        if (mConnectedThread != null) {
            mConnectedThread.cancel();
            mConnectedThread = null;
        }

        // Запускаем поток для слушания BluetoothServerSocket.
        if (mAcceptThread == null) {

```

```

        mAcceptThread = new AcceptThread();
        mAcceptThread.start();
    }
    setState(State.LISTEN);
}

/**
 * Запускаем поток ConnectThread, чтобы инициировать соединение
 * с удаленным устройством.
 * @param device устройство BluetoothDevice, с которым нужно соединиться
 */
public synchronized void connect(BluetoothDevice device) {
    if (D) Log.d(TAG, "connect to: " + device);

    // Отменяем любой поток, который пытается установить соединение.
    if (mState == State.CONNECTING) {
        if (mConnectThread != null) {
            mConnectThread.cancel();
            mConnectThread = null;
        }
    }

    // Отменяем любой поток, который в настоящее время поддерживает соединение.
    if (mConnectedThread != null) {
        mConnectedThread.cancel();
        mConnectedThread = null;
    }

    // Запускаем поток для соединения с заданным устройством.
    mConnectThread = new ConnectThread(device);
    mConnectThread.start();
    setState(State.CONNECTING);
}

/**
 * Запускаем поток ConnectedThread, чтобы начать управление соединением.
 * Bluetooth
 *
 * @param socket
 *     Сокет BluetoothSocket, с которым было установлено соединение
 * @param device
 *     Устройство BluetoothDevice, которое было подключено
 */
private synchronized void connected(BluetoothSocket socket,
    BluetoothDevice device) {
    if (D)
        Log.d(TAG, "connected");

    // Отменяем поток, завершивший соединение.
    if (mConnectThread != null) {
        mConnectThread.cancel();
        mConnectThread = null;
    }
}

```

```

// Отменяем любой поток, который в настоящее время поддерживает соединение.
if (mConnectedThread != null) {
    mConnectedThread.cancel();
    mConnectedThread = null;
}

// Отменяем принимающий поток, поскольку нам нужно только
// соединиться с одним устройством.
if (mAcceptThread != null) {
    mAcceptThread.cancel();
    mAcceptThread = null;
}

// Запускаем поток для управления соединением и осуществления передач.
mConnectedThread = new ConnectedThread(socket);
mConnectedThread.start();

// Отправляем имя подключенного устройства обратно к активности
// пользовательского интерфейса.
mHandler.obtainMessage(BtHelperHandler.MessageType.DEVICE, -1,
    device.getName()).sendToTarget();
setState(State.CONNECTED);
}

/**
 * Останавливаем все потоки.
 */
public synchronized void stop() {
    if (D) Log.d(TAG, "stop");
    if (mConnectThread != null) {
        mConnectThread.cancel();
        mConnectThread = null;
    }
    if (mConnectedThread != null) {
        mConnectedThread.cancel();
        mConnectedThread = null;
    }
    if (mAcceptThread != null) {
        mAcceptThread.cancel();
        mAcceptThread = null;
    }
    setState(State.NONE);
}

/**
 * Записываем данные в ConnectedThread в асинхронном режиме.
 * @param out байты, которые следует записать
 * @see ConnectedThread#write(byte[])
 */
public void write(byte[] out) {
    ConnectedThread r;

    // Синхронизируем копию ConnectedThread.

```

```

        synchronized (this) {
            if (mState != State.CONNECTED) return;
            r = mConnectedThread;
        }
        // Осуществление несинхронизированной записи.
        r.write(out);
    }

    private void sendErrorMessage(int messageId) {
        setState(State.LISTEN);
        mHandler.obtainMessage(BtHelperHandler.MessageType.NOTIFY, -1,
            mContext.getResources().getString(messageId)).sendToTarget();
    }

    /**
     * Этот поток слушает входящие соединения.
     */
    private class AcceptThread extends Thread {
        // сокет локального сервера
        private final BluetoothServerSocket mmServerSocket;

        public AcceptThread() {
            BluetoothServerSocket tmp = null;

            // Создаем новый сокет для слушания сервера.
            try {
                tmp = mAdapter.listenUsingRfcommWithServiceRecord(NAME,
                                                                    SPP_UUID);
            } catch (IOException e) {
                Log.e(TAG, "listen() failed", e);
            }
            mmServerSocket = tmp;
        }

        public void run() {
            if (D) Log.d(TAG, "BEGIN mAcceptThread" + this);
            setName("AcceptThread");
            BluetoothSocket socket = null;

            // Слушание серверного сокета, когда соединение отсутствует.
            while (mState != BtSPPHelper.State.CONNECTED) {
                try {
                    // Это блокирующий вызов, возвращающий результат только
                    // при успешном соединении либо выдающий исключение.
                    socket = mmServerSocket.accept();
                } catch (IOException e) {
                    Log.e(TAG, "accept() failed", e);
                    break;
                }

                // если соединение было принято
                if (socket != null) {

```

```

synchronized (BtSPPHelper.this) {
    switch (mState) {
        case LISTEN:
        case CONNECTING:
            // Ситуация нормальная. Запускаем соединенный поток.
            connected(socket, socket.getRemoteDevice());
            break;
        case NONE:
        case CONNECTED:
            // Либо система не готова, либо соединение уже
            // установлено. Завершаем работу нового сокета.
            try {
                socket.close();
            } catch (IOException e) {
                Log.e(TAG, "Could not close unwanted
                           socket", e);
            }
            break;
    }
}
}
}
}
if (D) Log.i(TAG, "END mAcceptThread");
}

public void cancel() {
    if (D) Log.d(TAG, "cancel " + this);
    try {
        mmServerSocket.close();
    } catch (IOException e) {
        Log.e(TAG, "close() of server failed", e);
    }
}
}

/**
 * Этот поток работает при попытке установить исходящее соединение
 * с устройством. Он работает прямолинейно: установить соединение
 * либо удастся, либо нет.
 */
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {
        mmDevice = device;
        BluetoothSocket tmp = null;

        // Получение сокета BluetoothSocket для соединения
        // с заданным устройством BluetoothDevice.
        try {
            tmp = device.createRfcommSocketToServiceRecord(SPP_UUID);

```

```

    } catch (IOException e) {
        Log.e(TAG, "create() failed", e);
    }
    mmSocket = tmp;
}

public void run() {
    Log.i(TAG, "BEGIN mConnectThread");
    setName("ConnectThread");

    // Всегда отменяем открытие, так как оно замедляет соединение.
    mAdapter.cancelDiscovery();

    // Устанавливаем соединение с BluetoothSocket.
    try {
        // Это блокирующий вызов, возвращающий результат только
        // при успешном соединении либо выдающий исключение.
        mmSocket.connect();
    } catch (IOException e) {
        sendErrorMessage(R.string.bt_unable);
        // Закрываем сокет.
        try {
            mmSocket.close();
        } catch (IOException e2) {
            Log.e(TAG, "unable to close() socket during connection
                failure", e2);
        }
        // Перезапускаем службу, чтобы заново запустить режим слушания.
        BtSPPHelper.this.start();
        return;
    }

    // Сбрасываем ConnectThread, так как все уже сделано.
    synchronized (BtSPPHelper.this) {
        mConnectThread = null;
    }

    // Запускаем подключенный поток.
    connected(mmSocket, mmDevice);
}

public void cancel() {
    try {
        mmSocket.close();
    } catch (IOException e) {
        Log.e(TAG, "close() of connect socket failed", e);
    }
}

}

/**
 * Этот поток работает во время соединения с удаленным устройством.

```

```

* Он обрабатывает передачи всех входящих и исходящих данных.
*/
private class ConnectedThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final InputStream mmInStream;
    private final OutputStream mmOutStream;

    public ConnectedThread(BluetoothSocket socket) {
        Log.d(TAG, "create ConnectedThread");
        mmSocket = socket;
        InputStream tmpIn = null;
        OutputStream tmpOut = null;

        // Получаем потоки ввода и вывода BluetoothSocket.
        try {
            tmpIn = socket.getInputStream();
            tmpOut = socket.getOutputStream();
        } catch (IOException e) {
            Log.e(TAG, "temp sockets not created", e);
        }

        mmInStream = tmpIn;
        mmOutStream = tmpOut;
    }

    public void run() {
        Log.i(TAG, "BEGIN mConnectedThread");
        byte[] buffer = new byte[1024];
        int bytes;

        // Продолжаем слушать поток InputStream во время соединения.
        while (true) {
            try {
                // Считываем из InputStream.
                bytes = mmInStream.read(buffer);

                // Отправляем полученные байты к активности
                //пользовательского интерфейса.
                mHandler.obtainMessage(BtHelperHandler.MessageType.READ,
                    bytes, buffer).sendToTarget();
            } catch (IOException e) {
                Log.e(TAG, "disconnected", e);
                sendErrorMessage(R.string.bt_connection_lost);
                break;
            }
        }
    }
}

/**
 * Записываем в подключенный OutStream.
 * @param buffer байты, которые необходимо записать
 */

```

```

public void write(byte[] buffer) {
    try {
        mmOutputStream.write(buffer);

        // Вновь предоставляем отправленное сообщение активности
        // пользовательского интерфейса для совместного использования.
        mHandler.obtainMessage(BtHelperHandler.MessageType.WRITE,
                                -1, buffer)
                .sendToTarget();
    } catch (IOException e) {
        Log.e(TAG, "Exception during write", e);
    }
}

public void cancel() {
    try {
        mmSocket.close();
    } catch (IOException e) {
        Log.e(TAG, "close() of connect socket failed", e);
    }
}
}
}

```

Класс `BtSPPHelper` объединяет использование всех этих классов, а также содержит определение вложенных подклассов потока `Thread`, которые занимаются слушанием, установлением соединений и их обслуживанием.

Именно здесь пакет `java.io` стыкуется с `Android Bluetooth`: объекты `BluetoothSocket` содержат методы, возвращающие ссылки на объекты `InputStream` и `OutputStream`, которые, в свою очередь, будут использоваться для считывания и записи данных при установленном сокет-соединении:

```

package com.finchframework.bluetooth;

import java.util.Set;

import com.finchframework.finch.R;

import android.app.Activity;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.Window;
import android.view.View.OnClickListener;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;

```

```

import android.widget.Button;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.AdapterView.OnItemClickListener;

/**
 * Этот пример является производным от Bluetooth Chat. Активность позволяет
 * выбрать сопряженное ранее или обнаруженное устройство с поддержкой Bluetooth.
 */
public class DeviceListActivity extends Activity {
    // отладка
    private static final String TAG = "DeviceListActivity";
    private static final boolean D = true;

    // extra намерения для возврата
    public static String EXTRA_DEVICE_ADDRESS = "device_address";

    // поля компонентов
    private BluetoothAdapter mBtAdapter;
    private ArrayAdapter<String> mPairedDevicesArrayAdapter;
    private ArrayAdapter<String> mNewDevicesArrayAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // задание окна
        setContentView(R.layout.device_list);

        // задание результата CANCELED на случай, если пользователь
        // завершит соединение
        setResult(Activity.RESULT_CANCELED);

        // инициализация кнопки, которая будет выполнять обнаружение устройства
        Button scanButton = (Button) findViewById(R.id.button_scan);
        scanButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                doDiscovery();
                v.setVisibility(View.GONE);
            }
        });

        // Инициализация адаптеров массивов (ArrayAdapter). Один применяется
        // для уже сопряженных устройств, один – для только что
        // обнаруженных устройств.
        mPairedDevicesArrayAdapter = new ArrayAdapter<String>(this,
            R.layout.device_name);
        mNewDevicesArrayAdapter = new ArrayAdapter<String>(this,
            R.layout.device_name);

        // Находим и настраиваем ListView для сопряженных устройств.
        ListView pairedListView = (ListView)

```

```

        findViewById(R.id.paired_devices);
        pairedListView.setAdapter(mPairedDevicesArrayAdapter);
        pairedListView.setOnItemClickListener(mDeviceClickListener);

        // Находим и настраиваем ListView для только что обнаруженных устройств.
        ListView newDevicesListView = (ListView)
            findViewById(R.id.new_devices);
        newDevicesListView.setAdapter(mNewDevicesArrayAdapter);
        newDevicesListView.setOnItemClickListener(mDeviceClickListener);

        // Регистрируемся на широковещательные сообщения,
        // когда обнаруживаем устройство.
        IntentFilter filter = new
            IntentFilter(BluetoothDevice.ACTION_FOUND);
        this.registerReceiver(mReceiver, filter);

        // Регистрируемся на широковещательные сообщения, когда обнаружение
        // завершено.
        filter = new IntentFilter(
            BluetoothAdapter.ACTION_DISCOVERY_FINISHED);
        this.registerReceiver(mReceiver, filter);

        // Получаем локальный адаптер Bluetooth.
        mBtAdapter = BluetoothAdapter.getDefaultAdapter();

        // Получаем набор устройств, сопряженных в настоящий момент.
        Set<BluetoothDevice> pairedDevices = mBtAdapter.getBondedDevices();

        // Если имеются сопряженные устройства, добавляем каждое из них
        // к ArrayAdapter.
        if (pairedDevices.size() > 0) {
            findViewById(R.id.title_paired_devices).setVisibility(
                View.VISIBLE);
            for (BluetoothDevice device : pairedDevices) {
                mPairedDevicesArrayAdapter.add(device.getName() +
                    "\n" + device.getAddress());
            }
        } else {
            String noDevices =
                getResources().getText(R.string.none_paired).toString();
            mPairedDevicesArrayAdapter.add(noDevices);
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();

        // Убеждаемся, что мы больше не занимаемся обнаружением.
        if (mBtAdapter != null) {
            mBtAdapter.cancelDiscovery();
        }
    }

```

```

        // Разрегистраруем слушатели широковещательных сообщений.
        this.unregisterReceiver(mReceiver);
    }

    /**
     * Запускаем обнаружение устройства с применением BluetoothAdapter.
     */
    private void doDiscovery() {
        if (D) Log.d(TAG, "doDiscovery()");

        // Обозначаем процесс сканирования в заголовке.
        setProgressBarIndeterminateVisibility(true);
        setTitle(R.string.scanning);

        // Активируем подзаголовок для новых устройств.
        findViewById(R.id.title_new_devices).setVisibility(View.VISIBLE);

        // Если мы уже занимаемся обнаружением – останавливаем этот процесс.
        if (mBtAdapter.isDiscovering()) {
            mBtAdapter.cancelDiscovery();
        }

        // запрос обнаружения от BluetoothAdapter
        mBtAdapter.startDiscovery();
    }

    // Слушатель нажатий, принимающий события от всех устройств,
    // перечисленных в ListViews.
    private OnItemClickListener mDeviceClickListener = new
        OnItemClickListener() {
        public void onItemClick(AdapterView<?> av, View v, int arg2, long
            arg3) {
            // Отмена процедуры обнаружения, поскольку она расходует
            // ресурсы, а мы уже готовимся подключиться.
            mBtAdapter.cancelDiscovery();

            // Получаем MAC-адрес устройства, это последние 17 символов в виде.
            String info = ((TextView) v).getText().toString();
            String address = info.substring(info.length() - 17);

            // Создаем результирующее намерение и включаем MAC-адрес.
            Intent intent = new Intent();
            intent.putExtra(EXTRA_DEVICE_ADDRESS, address);

            // Устанавливаем результат и завершаем эту активность.
            setResult(Activity.RESULT_OK, intent);
            finish();
        }
    };

    // Широковещательный приемник BroadcastReceiver, слушающий информацию от
    // обнаруженных устройств и изменяющий заголовок, когда обнаружение завершено.
    private final BroadcastReceiver mReceiver = new BroadcastReceiver() {

```

```

@Override
public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();

    // когда при обнаружении удастся найти устройство
    if (BluetoothDevice.ACTION_FOUND.equals(action)) {
        // Получаем от намерения объект BluetoothDevice.
        BluetoothDevice device =
            intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
        // Если оно уже сопряжено — пропускаем, так как мы уже слушаем его.
        if (device.getBondState() != BluetoothDevice.BOND_BONDED) {
            mNewDevicesArrayAdapter.add(
                device.getName() + "\n" + device.getAddress());
        }
    }
    // По завершении этапа обнаружения заменяем заголовок этой активности.
} else if (BluetoothAdapter.ACTION_DISCOVERY_FINISHED.
    equals(action)) {
    setProgressBarIndeterminateVisibility(false);
    setTitle(R.string.select_device);
    if (mNewDevicesArrayAdapter.getCount() == 0) {
        String noDevices =
            getResources().getText(
                R.string.none_found).toString();
        mNewDevicesArrayAdapter.add(noDevices);
    }
}
}
};
}

```

Класс DeviceListActivity

Эта активность отображает диалоговое окно, в котором перечисляются известные устройства и пользователь может активировать поиск (сканирование) устройств. В отличие от частей приложения, в которых подклассы Thread применяются для реализации асинхронного ввода/вывода, а подклассы Handler — для передачи результатов потоку пользовательского интерфейса, метод startDiscovery класса BluetoothAdapter запускает отдельный поток и сообщает результаты при помощи широковещательных намерений. Здесь для обработки результатов используется приемник широковещательных намерений BroadcastReceiver.

Класс BtConsoleActivity

Класс BtConsoleActivity создает активность для обмена мгновенными сообщениями (вроде чата) для взаимодействия с устройством Bluetooth. Меню этой активности позволяют подключаться к устройству, а основной вид в составе данной активности — это прокручиваемый список тех данных, которые отправляются и принимаются. В нижней части экрана расположено текстовое поле EditText. Сюда вводится текст, который потом отправляется на другой конец SPP-соединения.

Классы обработчиков используются для «склеивания» однопоточного пользовательского интерфейса с потоками, занимающимися слушанием, соединением и осуществлением операций ввода/вывода по сокетным соединениям.

19 **Комплект для нативной разработки в Android (NDK)**

Интерфейс для нативной разработки на языке Java (JNI) — это часть стандарта Java, позволяющая разработчикам писать его нативные методы на других языках, например на C или C++, и вызывать эти методы из кода Java. Кроме того, именно JNI соединяет среду времени исполнения Java с базовой операционной системой. Подробнее об этом — в документе Java Native Interface Specification (Спецификация нативного интерфейса Java) (<http://docs.oracle.com/javase/1.5.0/docs/guide/jni/>).

JNI оказывается особенно кстати, если вы собираетесь применять платформо-специфичные функции или воспользоваться преимуществами, предлагаемыми аппаратной частью платформы, которых не позволяют достичь API Android. Например, может быть необходимо повысить скорость вычислений, пользуясь командами FPU. Еще один аспект, с которым удобно применять JNI, — код для интенсивной обработки графики, активно использующий OpenGL API.

В этой главе изложены основы работы с JNI для программистов, работающих с комплектом для нативной разработки в Android (Android NDK). NDK упрощает компиляцию нативного кода, который может задействоваться программами Android.

Принимая решение о том, стоит ли начинать разработку в нативном коде, внимательно изучите стоящие перед вами требования и подумайте, нет ли уже в Android SDK необходимой вам функциональности. Если в программе используется JNI, то усложняется ее отладка. Кроме того, программа может работать только на процессорах, поддерживающих NDK. В настоящее время это всего два варианта: ARM и x86.

Нативные методы и вызовы нативного интерфейса Java (JNI)

JNI требует следовать определенным соглашениям, чтобы можно было делать вызовы методов из других языков. Нативные методы (в сущности, относящиеся к библиотекам C или C++) претерпевают при этом более серьезные изменения, чем код Java.

Соглашения на стороне нативных методов

Когда VM (виртуальная машина, в случае с Android это Dalvik) активирует функцию, реализованную на языке C или C++, машина передает функции два специальных параметра:

- указатель JNIEnv, своего рода описатель того потока, в котором виртуальная машина вызывает нативный метод;
- указатель jobject, представляющий собой ссылку на вызывающий класс.

Эти параметры явно передаются коду Java. Это означает, что они не упоминаются в сигнатуре метода, объявляемого в вызывающем коде Java. Вызов Java просто явно передает все остальные параметры, которые нужны для работы вызванной функции.

Функция JNI может иметь следующий вид:

```
/* образец метода, в котором вызов Java не передает никаких параметров */
void Java_ClassName_MethodName (JNIEnv *env, jobject obj) {
    /* выполняем какую-нибудь операцию */
}

/* Еще один метод, которому переданы два параметра, возвращающие значение double.
*/
jdouble Java_ClassName_MethodName ( JNIEnv* env, jobject obj,
                                     jdouble x, jdouble y) {
    return x + y;
}
```

В этих примерах показаны два параметра, которые автоматически передаются каждому нативному методу, и два параметра с типами, отображаемыми на типы Java.

При вызове нативного метода данный метод выполняет тот же процесс и тот же поток, что и код Java, вызывающий его. Как будет показано далее в этой главе, он может выделять память из кучи Java, чтобы пользоваться преимуществами, связанными со сборкой мусора, а также брать память вне кучи Java, чтобы обходить систему управления памятью, действующую в Java. Стековые переменные из кода на C или C++ имеют ту же семантику, что и в нативных исполняемых модулях на этих языках. Они размещаются в стеке того процесса, в котором работают.

В JNI предоставляются типы, соответствующие типам Java (табл. 19.1).

Таблица 19.1. Отображение данных

Нативный тип	Тип Java	Описание
boolean	jboolean	8 бит без знака
byte	jbyte	8 бит со знаком
char	jchar	16 бит без знака
short	jshort	16 бит со знаком
int	jint	32 бита со знаком
long	jlong	64 бита со знаком
float	jfloat	32 бита
double	jdouble	64 бита
void	void	Недоступно

В составных типах — таких как объекты, массивы и строки — нативный код должен явно преобразовывать данные, вызывая методы преобразования, доступные через указатель JNIEnv.

Соглашения на стороне Java

Перед тем как нативные методы можно будет использовать в классе Java, нужно загрузить библиотеку с нативными методами. Для этого требуется вызвать `System.loadLibrary`. Как правило, тот класс, которому требуется нативный метод, статически ее загружает. Нативные методы, к которым обращается класс, объявляются в классе при помощи ключевого слова `native`:

```
public class ClassWithNativeMethod {

    public native double nativeMethod();    // нативный метод

    static {
        System.loadLibrary("sample");    // загружаем библиотеку 'sample'
    }

    public static void main(String[] args) {
        ClassWithNativeMethod cwnm = new ClassWithNativeMethod();

        double answer = cwnm.nativeMethod(); // вызов нативного метода

        System.out.println("Answer is : "+answer);
    }
}
```

Комплект для нативной разработки в Android (Android NDK)

Комплект для нативной разработки в Android (NDK) — это парный инструмент для Android SDK. Если вы используете NDK для написания нативного кода, то ваши приложения, как и в других случаях, будут находиться в архивах APK и работать на устройстве внутри виртуальной машины. Фундаментальная модель приложения Android не изменяется.

Настройка среды для нативной разработки в Android

Чтобы приступить к работе с NDK, сначала нужно установить и настроить SDK. Системные требования для установки NDK и работы с ним таковы:

- Windows XP (32 бита) или Vista (32 или 64 бита) со средой Cygwin версии 1.7 и выше; либо Mac OS X 10.4.8 или выше; либо Linux (32 или 64 бита);

- GNU Make 3.81 или выше;
- новая версия Awk (GNU Awk или Nawk).

Сначала нужно скачать и установить NDK (<http://developer.android.com/sdk/ndk/index.html>). Установка проста: распакуйте архив. Название каталога верхнего уровня включает номер версии NDK. Мы назовем этот каталог `ndk`. Если версия, с которой вы будете работать, окажется более новой, чем использованная нами при написании книги, то, возможно, она будет поддерживать больше нативных API.

После того как NDK будет скачан и установлен, вы обнаружите в подкаталоге `ndk/docs` довольно много документации. Настоятельно рекомендуем прочитать ее, начиная с файла `OVERVIEW.html`. Кроме того, в NDK включены примеры (расположенные в подкаталоге `ndk/samples`). На примерах объясняется более обширный материал, чем мы затрагиваем в этой главе, поэтому, поработав некоторое время с NDK, вы изучите и все эти примеры.

Редактирование кода C/C++ в Eclipse

Чтобы максимально эффективно редактировать код языка C в Eclipse, нужно установить инструментарий разработки Eclipse C/C++, также называемый Eclipse CDT. Он добавляет в среду Eclipse редактор, способный редактировать код C с подсветкой синтаксиса, обеспечивает форматирование и другие возможности, сравнимые с функционалом Eclipse для редактирования Java.

Название репозитория для нужной версии CDT зависит от того, с какой версией Eclipse вы работаете. В случае с Eclipse Indigo это репозиторий, расположенный по адресу <http://download.eclipse.org/tools/cdt/releases/indigo>. Eclipse Indigo вы найдете в диалоговом окне **Install New Software** (Установка новых программ). О работе с Eclipse, добавлении пакетов в среду Eclipse и использовании статического анализа подробнее рассказано в главе 5.

Компиляция с NDK

Чтобы разработать нативный код с применением NDK, нужно сделать следующее.

1. Создать в своем проекте каталог `jni`.
2. Разместить в каталоге `jni` нативный исходный код.
3. Создать файл `Android.mk` (возможно, потребуется еще и `Application.mk`) в каталоге `jni`.
4. Запустить команду `ndk/ndk-build` в каталоге `jni`.

Необязательный файл `Application.mk` описывает, какие нативные модули потребуются для вашего приложения, а также специфические типы ABI (двоичных интерфейсов приложений), в соответствии с которыми будет происходить компиляция. В данном случае мы задаем `all`, указывая, что нас интересуют все доступные типы API. На момент написания книги к ним относятся: ARM5, именуемый в NDK `armeabi`; ARM7, именуемый `armeabi-v7a`; а также архитектура Intel x86, именуемая `x86`. Более подробно этот вопрос описывается в файле `APPLICATION-MK.html` в документации. Вот пример файла `Application.mk`:

```
# Сборка всех доступных ABI
APP_ABI := all
# Для какой платформы (уровень API) производится сборка
APP_PLATFORM := android-14
```

Файл `Android.mk` содержит описание вашего исходного кода для системы сборки. Это маленький фрагмент `GNU Makefile`, который проходит специальный синтаксический разбор системой сборки во время компиляции вашего приложения. Подробнее этот этап описан в файле документации `ANDROID-MK.html`. Вот пример `Android.mk`:

```
# необходимо определить LOCAL_PATH и вернуть актуальный каталог
LOCAL_PATH := $(call my-dir)

# очистка различных переменных... создание чистой сборки
include $(CLEAR_VARS)

# идентификация имени модуля/библиотеки
LOCAL_MODULE := sample
# указание исходных файлов
LOCAL_SRC_FILES := sample.c
# загрузка локальных библиотек (здесь загружается библиотека журналов)
LOCAL_LDLIBS := -llog

# построение разделяемой библиотеки, определенной выше
include $(BUILD_SHARED_LIBRARY)
```

Когда будут написаны файл `Android.mk`, а также файл `Application.mk` и сами нативные файлы с исходным кодом, запустите `ndk/ndk-build` в каталоге проекта, чтобы скомпилировать ваши библиотеки. Если сборка пройдет успешно, то разделяемые библиотеки будут скопированы в корневой каталог проекта с вашим приложением и добавлены к его сборке.

Если скачать исходный код `Android` и хорошенько в нем разобраться, то станет понятно, что файлы сборки очень напоминают подобный файл, используемый во всей системе сборки `Android`. Написать простое приложение, использующее `Android SDK`, — хороший способ познакомиться с применением интерфейса `JNI`, нативного кода, а также понять, как нативный код интегрируется в систему `Android`.

JNI, NDK и SDK: образец приложения

Чтобы проиллюстрировать, как могут совместно работать `SDK` и нативный код, мы сделали следующее приложение-образец. Оно описывает активность `SampleActivityWithNativeMethods`. Вот фрагмент из файла описания `Android`:

```
<activity android:name=".SampleActivityWithNativeMethods"
    android:label="Sample Activity With Native Methods"
    android:debuggable="true" />
```

Активность `SampleActivityWithNativeMethods` использует следующий макет:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
```

```

        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
    >
<Button
    android:id="@+id/whatami"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:paddingTop="5dp"
    android:paddingBottom="5dp"
    android:text="What CPU am I?"
/>
</LinearLayout>

```

В том библиотечном коде на языке C, который мы взяли в качестве примера, есть метод `whatAmI`, который наша активность на Java будет привязывать к кнопке, имеющей ID `whatami`. Кроме того, мы определим функцию под названием `LOGINFO`, преобразуемую в вызов `__android_log_print`. Вот как делается запись в журнале Android:

```

// НЕОБХОДИМО подключить здесь библиотеку jni
#include <jni.h>
// здесь подключается библиотека журналов
#include <android/log.h>

// работа с журналом
#define LOGINFO(x...) __android_log_print(ANDROID_LOG_INFO, "SampleJNI", x)

jstring Java_com_oreilly_demo_pa_ch18_SampleActivityWithNativeMethods_whatAmI(
    JNIEnv* env, jobject thisobject) {
    LOGINFO("SampleJNI", "Sample Info Log Output");

    return (*env)->NewStringUTF(env, "Unknown");
}

```

Далее идет наш файл `Android.mk`. Обратите внимание: именно он обеспечивает загрузку библиотеки журналов.

```

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := sample
LOCAL_SRC_FILES := sample.c
LOCAL_LDLIBS := -llog

include $(BUILD_SHARED_LIBRARY)

```

Наконец, вот исходный код активности `SampleActivityWithNativeMethods` на языке Java. Класс загружает библиотеку и объявляет нативный метод `whatAmI()`. При нажатии кнопки вызывается метод `whatAmI()`, возвращающий "Unknown". В результате этого отображается Toast (всплывающее сообщение) со строкой CPU: Unknown (процессор: неизвестен). Если вывод вам кажется информативным, не беспокойтесь: информацию о процессоре мы тоже добавим, но в следующем разделе:

```

package com.oreilly.demo.android.pa.ndkdemo;

import com.oreilly.demo.android.pa.ndkdemo.R;
import android.widget.Toast;

public class SampleActivityWithNativeMethods extends Activity {

    static {
        System.loadLibrary("sample"); // загружаем нашу библиотеку-образец
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.sample);

        setupview();
    }

    public native String whatAmI(); // пример нативного метода из библиотеки

    private void setupview() {
        findViewById(R.id.whatami).setOnClickListener(
            new View.OnClickListener() {

                public void onClick(View v) {
                    String whatami = whatAmI();
                    Toast.makeText(getBaseContext(), "CPU: "+whatami,
                                Toast.LENGTH_SHORT).show();
                }
            }
        );
    }
}

```

Нативные библиотеки и заголовки, предоставляемые в NDK

В состав NDK входят следующие заголовки для стабильных нативных API:

- заголовки `libc` (библиотека языка C);
- заголовки `libm` (математическая библиотека);
- заголовки интерфейса JNI;
- заголовки `libz` (архивация ZLib);
- заголовок `liblog` (журналирование в Android);
- заголовки OpenGL ES 1.1 и OpenGL ES 2.0 (это библиотеки трехмерной графики);
- заголовок `libjnigraphics` (для доступа к буферу пикселей в Android 2.2 и выше);
- минимальный набор заголовков для поддержки C++;

- нативные аудиобиблиотеки для OpenSL ES;
- нативные интерфейсы программирования приложений Android.



Другие применяемые в Android нативные системные библиотеки, кроме перечисленных выше, не являются стабильными и в более новых версиях платформы могут измениться. В своем приложении вам следует использовать только стабильные нативные системные библиотеки, входящие в состав NDK.

На некоторые библиотеки, в частности `libc` и `libm`, в процессе сборки автоматически ставятся ссылки, поэтому на них нужно ссылаться только в исходном коде как на `#include`. Вместе с тем на некоторые библиотеки ссылки не ставятся автоматически, и в файле сборки `Android.mk` для ссылки на них требуются специальные инструкции.

Вот пример файла `Android.mk`, в котором импортируется модуль `cpufeatures`. Он сообщит нам ту информацию, которой не хватало в нашем предыдущем примере с `whatAmI`:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := sample
LOCAL_SRC_FILES := sample.c
LOCAL_LDLIBS := -llog
# Здесь делается ссылка на модуль cpufeatures
LOCAL_STATIC_LIBRARIES := cpufeatures

include $(BUILD_SHARED_LIBRARY)

# Здесь импортируются модули cpufeatures
$(call import-module,cpufeatures)
```

В следующем исходном коде (дополняющем функцию `whatAmI`, которую мы показали в предыдущем разделе) используется модуль `cpufeatures`, который мы включили:

```
// подключение модуля cpu-features
#include <cpu-features.h>
#include <jni.h>
#include <android/log.h>

#define LOGINFO(x...) __android_log_print(ANDROID_LOG_INFO,"SampleJNI",x)

jstring
Java_com_oreilly_demo_android_pa_ndkdemo_SampleActivityWithNativeMethods_whatAmI(
    JNIEnv* env, jobject thisobject) {
    LOGINFO("SampleJNI","Sample Info Log Output");

    // -- здесь используется cpufeatures -- //
    uint64_t cpu_features;
```

```

if (android_getCpuFamily() != ANDROID_CPU_FAMILY_ARM) {
    return (*env)->NewStringUTF(env, "Not ARM");
}

cpu_features = android_getCpuFeatures();

if ((cpu_features & ANDROID_CPU_ARM_FEATURE_ARMv7) != 0) {
    return (*env)->NewStringUTF(env, "ARMv7");
} else if ((cpu_features & ANDROID_CPU_ARM_FEATURE_VFPv3) != 0) {
    return (*env)->NewStringUTF(env, "ARM w VFPv3 support");
} else if ((cpu_features & ANDROID_CPU_ARM_FEATURE_NEON) != 0) {
    return (*env)->NewStringUTF(env, "ARM w NEON support");
}

// -- окончание использования cpufeatures -- //

return (*env)->NewStringUTF(env, "Unknown");
}

```

Создание собственных пользовательских библиотечных модулей

В этом разделе мы объединим несколько техник, рассмотренных в данной главе, и применим их для создания и использования простого модуля на языке С, применяющего математическую библиотеку для расчета мощности. Начнем с файла `Android.mk`. Обратите внимание: нам нужно собрать библиотеку (`sample_lib`) и экспортировать включения. Затем на эту библиотеку в примере ставится ссылка:

```

LOCAL_PATH := $(call my-dir)

# это наша библиотека-пример
include $(CLEAR_VARS)

LOCAL_MODULE := sample_lib
LOCAL_SRC_FILES := samplelib/sample_lib.c
# необходимо убедиться, что всем элементам "известно",
# где находятся другие нужные им элементы
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/samplelib

include $(BUILD_STATIC_LIBRARY)

# пример использует созданную нами библиотеку-пример
include $(CLEAR_VARS)

LOCAL_MODULE := sample
LOCAL_SRC_FILES := sample.c
LOCAL_LDLIBS := -llog
# загрузка нашей библиотеки-примера
LOCAL_STATIC_LIBRARIES := sample_lib

include $(BUILD_SHARED_LIBRARY)

```

У нас есть короткий файл заголовка, `sample_lib.h`:

```
#ifndef SAMPLE_LIB_H
#define SAMPLE_LIB_H

extern double calculatePower(double x, double y);

#endif
```

Вот исходный код для нашей функции `sample_lib.c`:

```
#include "sample_lib.h"
// подключаем математическую библиотеку
#include "math.h"

// используем математическую библиотеку
double calculatePower(double x, double y) {
    return pow(x, y);
}
```

Далее рассмотрим файл `sample.c`, склеивающий нашу библиотеку `sample_lib` с кодом Java:

```
// подключаем библиотеку-образец
#include "sample_lib.h"
#include <jni.h>
#include <android/log.h>

#define LOGINFO(x...) __android_log_print(ANDROID_LOG_INFO, "SampleJNI", x)

jdouble
Java_com_oreilly_demo_android_pa_ndkdemo_SampleActivityWithNativeMethods_
    calculatePower(JNIEnv* env, jobject thisobject, jdouble x,
        jdouble y) {

    LOGINFO("Sample Info Log Output");

    // вызываем метод расчетов из библиотеки-примера
    return calculatePower(x, y);
}
```

Активность будет использовать следующий макет:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <EditText
        android:id="@+id/x"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:paddingTop="5dp"
        android:paddingBottom="5dp"
```

```

        android:textColor="#000"
        android:hint="X Value"
    />
<EditText
    android:id="@+id/y"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:paddingTop="5dp"
    android:paddingBottom="5dp"
    android:textColor="#000"
    android:hint="Y Value"
/>
<Button
    android:id="@+id/calculate"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:paddingTop="5dp"
    android:paddingBottom="5dp"
    android:text="Calculate X^Y"
/>
</LinearLayout>

```

Далее приведена активность `SampleActivityWithNativeMethods`, которую мы изменили так, чтобы она могла работать с этой новой библиотекой. Загружается библиотека-пример и объявляется метод `calculatePower()`. При нажатии кнопки **Calculate** (Рассчитать) мы берем числа, записанные в двух текстовых полях (по умолчанию используется значение 2, если текст отсутствует или не является числом), и передаем их методу `calculatePower()`. Затем возвращенное число с плавающей запятой (`double`) отображается во всплывающем сообщении (`Toast`):

```

package com.oreilly.demo.android.pa.ndkdemo;

import com.oreilly.demo.android.pa.ndkdemo.R;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class SampleActivityWithNativeMethods extends Activity {

    static {
        System.loadLibrary("sample"); // загружаем нашу библиотеку-пример
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.sample);

        setupview();
    }
}

```

```

// нативный метод из библиотеки-примера
public native double calculatePower(double x, double y);

private void setupview() {

    findViewById(R.id.calculate).setOnClickListener(
        new View.OnClickListener() {

            public void onClick(View v) {
                String answer = "";
                double x = 2;
                double y = 2;

                String sx = ((EditText)
                    findViewById(R.id.x)).getText().toString();
                String sy = ((EditText)
                    findViewById(R.id.y)).getText().toString();

                if(sx == null) {
                    answer = "X defaults to 2\n";
                } else {
                    try {
                        x = Double.parseDouble(sx);
                    } catch (Exception e) {
                        answer = "X is not a number, defaulting to 2\n";
                        x = 2;
                    }
                }

                if(sy == null) {
                    answer += "Y defaults to 2\n";
                } else {
                    try {
                        y = Double.parseDouble(sy);
                    } catch (Exception e) {
                        answer = "Y is not a number, defaulting to 2\n";
                        y = 2;
                    }
                }

                double z = calculatePower(x, y);

                answer += x+"^"+y+" = "+z;

                Toast.makeText(SampleActivityWithNativeMethods.this,
                    answer, Toast.LENGTH_SHORT).show();
            }
        });
}
}

```

Нативные активности

В Android 2.3 (API уровня 9) и в версии 5 Android NDK разработчик может писать целые активности и приложения как нативный исходный код, используя класс `NativeActivity` для доступа к жизненному циклу приложения Android.

Для использования этого метода на `android.app.NativeActivity` нужно поставить ссылку в файле описания Android. Обратите внимание на атрибут `hasCode`, имеющийся у ссылки на приложение (application reference). Если в приложении нет кода на языке Java (только `NativeActivity`), то этот атрибут должен иметь значение `false`. Но в нашем случае код на языке Java есть, поэтому данный атрибут будет иметь значение `true`:

```
<!-- В данном архиве APK есть код на Java, поэтому атрибут hasCode
      получает значение true, которое действует по умолчанию. -->
<!-- Если бы мы работали только с нативным приложением (только с активностью
      'android.app.NativeActivity') -->
<!-- то атрибут имел бы значение false. -->

<application android:icon="@drawable/icon" android:label="@string/app_name"
      android:hasCode="true" >

    <activity android:name=".NDKApp" android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <activity android:name="android.app.NativeActivity"
        android:label="SampleNativeActivity"
        android:debuggable="true" >

        <!-- здесь мы объявляем, на какую библиотеку сослаться -->
        <meta-data android:name="android.app.lib_name"
            android:value="sample_native_activity" />
    </activity>

</application>
```

В данном примере мы использовали файл заголовка `android_native_app_glue.h`, а не `native_activity.h`. Интерфейс `native_activity.h` работает на основе обратных вызовов, предоставляемых приложением, которые будут выполняться в основном потоке активности при возникновении определенного события. Это означает, что вызовы не должны блокироваться — и это ограничивающий фактор. Файл `android_native_app_glue.h` предоставляет вспомогательную библиотеку, в которой реализуется иная модель управления, позволяющая приложению реализовывать собственную основную функцию в другом главном потоке. Функция должна иметь имя `android_main()` и вызываться, когда приложение создается и ей передается объект `android_app`. Таким образом, у нас появляется возможность ставить ссылку на приложение или активность и слушать различные события жизненного цикла.

В следующем простом примере `nativeactivity` создается активность и происходит слушание событий `Motion`. Экранные координаты x и y , в которых происходят события `Motion`, отправляются в `LogCat`:

```
#include <jni.h>
#include <android/log.h>
#include <android_native_app_glue.h>

// работа с журналом
#define LOGINFO(x...) __android_log_print(ANDROID_LOG_INFO, "SampleNativeActivity", x)
// обработка команд
static void custom_handle_cmd(struct android_app* app, int32_t cmd) {
    switch(cmd) {
        case APP_CMD_INIT_WINDOW:
            LOGINFO("App Init Window");
            break;
    }
}

// обработка ввода
static int32_t custom_handle_input(struct android_app* app, AInputEvent*
                                event) {
    // видим событие движения и регистрируем его
    if (AInputEvent_getType(event) == AINPUT_EVENT_TYPE_MOTION) {
        LOGINFO("Motion Event: x %f / y %f", AMotionEvent_getX(event, 0),
            AMotionEvent_getY(event, 0));
        return 1;
    }
    return 0;
}

// Это функция, которую должен реализовывать код приложения.
// представляющая входную точку в приложение.
void android_main(struct android_app* state) {
    // Удостоверяемся, что склеивающий код не разорван.
    app_dummy();

    int events;
    // Делаем так, чтобы при поступлении команд мы вызывали
    // наш пользовательский обработчик.
    state->onAppCmd = custom_handle_cmd;
    // Делаем так, чтобы при поступлении ввода мы вызывали
    // наш пользовательский обработчик.
    state->onInputEvent = custom_handle_input;

    while (1) {
        struct android_poll_source* source;

        // блокировка событий
        while (ALooper_pollAll(-1, NULL, &events, (void**)&source) >= 0) {
```

```

        // обработка этого события
        if (source != NULL) {
            source->process(state, source);
        }

        // проверка того, следует ли сделать здесь выход
        if (state->destroyRequested != 0) {
            LOGINFO("We are exiting");
            return;
        }
    }
}
}
}

```

Вот файл `Android.mk` нашей активности-примера, `nativeactivity`. Обратите внимание на то, что он загружает модуль `android_native_app_glue` и ссылается на него:

```

LOCAL_PATH := $(call my-dir)

# Это наш пример нативной активности.
include $(CLEAR_VARS)

LOCAL_MODULE := sample_native_activity
LOCAL_SRC_FILES := sample_nativeactivity.c
LOCAL_LDLIBS := -llog -landroid
LOCAL_STATIC_LIBRARIES := android_native_app_glue

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)

```

Далее идет основная активность `Android` на языке `Java`, вызываемая, когда пользователь запускает приложение. При нажатии кнопки запускается созданная нами активность `NativeActivity`:

```

package com.oreilly.demo.android.pa.ndkdemo;

import com.oreilly.demo.android.pa.ndkdemo.R;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class NDKApp extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        findViewById(R.id.nativeactivity).setOnClickListener(
            new View.OnClickListener() {

```

```

        public void onClick(View v) {
            startActivity(new Intent(getBaseContext(),
                android.app.NativeActivity.class)); // вызов
                                                    // nativeactivity
        }
    });
}
}

```

Если скомпилировать и запустить этот пример, то мы увидим, что при запуске нативной активности экран пуст. Если при этом просмотреть LogCat, в этом окне будут появляться различные записи журнала (особенно если провести пальцем по экрану). Но все это, конечно, не слишком интересно. Чтобы экран выглядел красивее, нам нужно воспользоваться OpenGL ES. В следующем примере мы изменяем цвет экрана.

Вот нативный исходный код с дополнительным материалом из OpenGL ES. При отображении активности экран просто становится ярко-красным:

```

#include <jni.h>
#include <android/log.h>
#include <android_native_app_glue.h>

#include <EGL/egl.h>
#include <GLES/gl.h>

// работа с журналом
#define LOGINFO(x...)
__android_log_print(ANDROID_LOG_INFO, "NativeWOpenGL", x)

struct eglengine {
    EGLDisplay display;
    EGLSurface surface;
    EGLContext context;
};

// инициализация движка egl
static int engine_init_display(struct android_app* app, struct eglengine*
    engine) {
    const EGLint attribs[] = {
        EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
        EGL_BLUE_SIZE, 8,
        EGL_GREEN_SIZE, 8,
        EGL_RED_SIZE, 8,
        EGL_NONE
    };
    EGLint w, h, dummy, format;
    EGLint numConfigs;
    EGLConfig config;
    EGLSurface surface;
    EGLContext context;

```

```

EGLDisplay display = eglGetDisplay(EGL_DEFAULT_DISPLAY);
eglInitialize(display, 0, 0);
eglChooseConfig(display, attribs, &config, 1, &numConfigs);
eglGetConfigAttrib(display, config, EGL_NATIVE_VISUAL_ID, &format);

ANativeWindow_setBuffersGeometry(app->window, 0, 0, format);

surface = eglCreateWindowSurface(display, config, app->window, NULL);
context = eglCreateContext(display, config, NULL, NULL);

if (eglMakeCurrent(display, surface, surface, context) == EGL_FALSE) {
    LOGINFO(„eglMakeCurrent FAIL");
    return -1;
}

eglQuerySurface(display, surface, EGL_WIDTH, &w);
eglQuerySurface(display, surface, EGL_HEIGHT, &h);

engine->display = display;
engine->context = context;
engine->surface = surface;

glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST);
glEnable(GL_CULL_FACE);
glShadeModel(GL_SMOOTH);
glDisable(GL_DEPTH_TEST);

return 0;
}

// рисуем на экране
static void engine_color_screen(struct eglengine* engine) {
    if (engine->display == NULL) {
        return;
    }

    glClearColor(255, 0, 0, 1); // весь экран станет красным
    glClear(GL_COLOR_BUFFER_BIT);

    eglSwapBuffers(engine->display, engine->surface);
}

// когда все нужно завершить
static void engine_terminate(struct eglengine* engine) {
    if (engine->display != EGL_NO_DISPLAY) {
        eglMakeCurrent(engine->display, EGL_NO_SURFACE, EGL_NO_SURFACE,
            EGL_NO_CONTEXT);
        if (engine->context != EGL_NO_CONTEXT) {
            eglDestroyContext(engine->display, engine->context);
        }
        if (engine->surface != EGL_NO_SURFACE) {

```

```

        eglDestroySurface(engine->display, engine->surface);
    }
    eglTerminate(engine->display);
}
engine->display = EGL_NO_DISPLAY;
engine->context = EGL_NO_CONTEXT;
engine->surface = EGL_NO_SURFACE;
}

// обработка команд
static void custom_handle_cmd(struct android_app* app, int32_t cmd) {
    struct eglengine* engine = (struct eglengine*)app->userData;
    switch(cmd) {
        // Все начинается... Инициализируем движок и окрасим экран.
        case APP_CMD_INIT_WINDOW:
            if (app->window != NULL) {
                engine_init_display(app, engine);
                engine_color_screen(engine);
            }
            break;
        case APP_CMD_TERM_WINDOW: // Тут все заканчивается... очищаем движок.
            engine_terminate(engine);
            break;
    }
}

// обработка ввода
static int32_t custom_handle_input(struct android_app* app, AInputEvent*
                                event) {
    // замечаем событие движения и регистрируем его
    if (AInputEvent_getType(event) == AINPUT_EVENT_TYPE_MOTION) {
        LOGINFO("Motion Event: x %f / y %f", AMotionEvent_getX(event, 0),
            AMotionEvent_getY(event, 0));
    }
    return 1;
}

// Это функция, которую должен реализовывать код приложения,
// представляющая входную точку в приложение.
void android_main(struct android_app* state) {
    // Удостоверяемся, что склеивающий код не разорван.
    app_dummy();

    // Здесь в приложение добавляется движок eglengine.
    struct eglengine engine;
    memset(&engine, 0, sizeof(engine));
    // Движок задаем как пользовательские данные, чтобы на него было
    // можно ссылаться.
    state->userData = &engine;
}

```

```

int events;
// Делаем так, чтобы при поступлении команд мы вызывали
// наш пользовательский обработчик.
state->onAppCmd = custom_handle_cmd;
// Делаем так, чтобы при поступлении ввода мы вызывали
// наш пользовательский обработчик.
state->onInputEvent = custom_handle_input;

while (1) {
    struct android_poll_source* source;

    // блокировка событий
    while (ALooper_pollAll(-1, NULL, &events, (void**)&source) >= 0) {

        // Обработка этого события
        if (source != NULL) {
            source->process(state, source);
        }

        // Проверка того, следует ли сделать здесь выход.
        if (state->destroyRequested != 0) {
            LOGINFO("We are exiting");
            return;
        }
    }
}
}
}

```

Файл `Android.mk` для активности `sample_native_activity_opengl` загружает библиотеки `EGL` и `GLSv1_CM`:

```

LOCAL_PATH := $(call my-dir)

# Это наша нативная активность-образец с opengl.
include $(CLEAR_VARS)

LOCAL_MODULE := sample_native_activity_opengl
LOCAL_SRC_FILES := sample_nativeactivity_opengl.c
# загрузка библиотек для журнала, android, egl, gles
LOCAL_LDLIBS := -llog -landroid -lEGL -lGLSv1_CM
LOCAL_STATIC_LIBRARIES := android_native_app_glue

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)

```