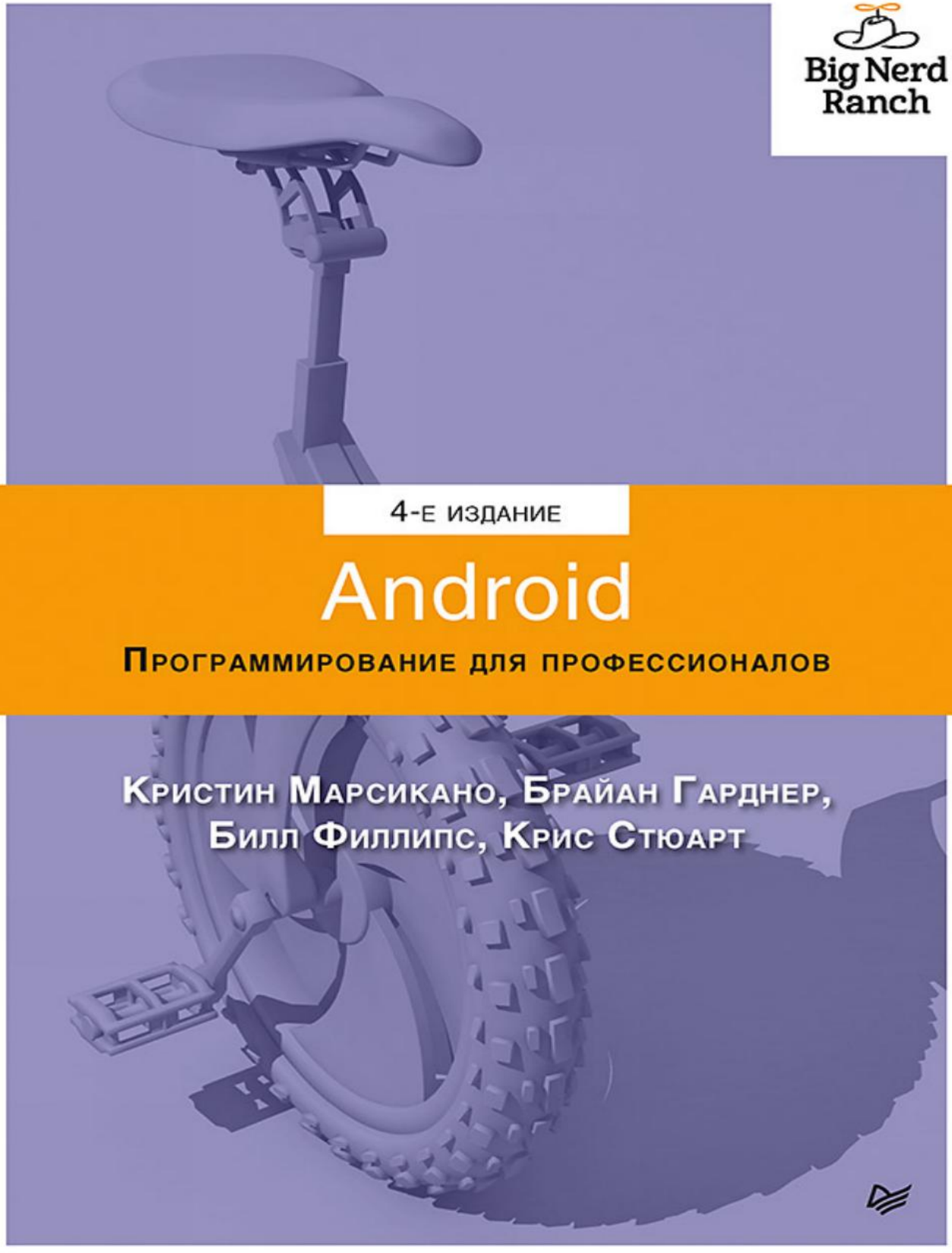


Big Nerd Ranch

4-е издание

Android

Программирование для профессионалов

Кристин Марсикано, Брайан Гарднер,
Билл Филлипс, Крис Стюарт

Билл Филлипс, Крис Стюарт, Кристин Марсикано, Брайан Гарднер

Android. Программирование для профессионалов. 4-е издание

2020

Переводчик *С. Черников*

Художник *В. Мостипан*

Корректоры *М. Молчанова (Котова), Г. Шкатова*

Билл Филлипс, Крис Стюарт, Кристин Марсикано, Брайан Гарднер

Android. Программирование для профессионалов. 4-е издание. — СПб.: Питер, 2020.

ISBN 978-5-4461-1657-7

Посвящается Филу, Ною и Сэму за их любовь и поддержку на протяжении работы над всеми изданиями этой книги.

К. М.

Посвящается моей жене Карли за поддержку во всех начинаниях и за напоминание о том, что действительно важно.

Б. Г.

Посвящается старому проигрывателю на моем рабочем столе. Благодарю тебя за то, что ты был со мной все это время. Обещаю скоро купить тебе новую иглу.

Б. Ф.

Посвящается моему отцу Дэвиду, научившему меня тому, как важно упорно работать, и моей матери Лизе, которая требовала, чтобы я всегда поступал правильно.

К. С.

Благодарности

В четвертом издании книги мы скажем то же, что и всегда: книги создаются не только авторами. Своим существованием они обязаны многочисленным помощникам и коллегам, без которых мы не смогли бы представить и написать весь этот материал.

- Спасибо Брайану Харди (Brian Hardy), которому вместе с Биллом Филлипсом (Bill Phillips) хватило смелости воплотить в реальность самое первое издание этой книги. Начав с нуля, Брайан и Билл сделали замечательный проект.

- Спасибо Эрику Максвеллу (Eric Maxwell), который в одиночку написал раздел «Для любознательных» о внедрении зависимостей, за улучшение информирования по нашим каналам, а также за исправление многочисленных замечаний редакторов на некоторых страницах этой книги.

- Спасибо Дэвиду Гринхалгу (David Greenhalgh) и Джошу Скину (Josh Skeen) за их бесценный опыт, пригодившийся нам, когда мы «котлинизировали» эту книгу и узнавали, как стать разработчиками Android в мире Kotlin.

- Спасибо Джереми Шерману (Jeremy Sherman) за его неожиданные познания в Android, пригодившиеся как раз тогда, когда это было нужно больше всего. Спасибо, Джереми, за ваши подробные и вдумчивые обзоры, а также за то, что позволили нам процитировать вас в одном из разделов.

- Спасибо Брайану Линдси (Bryan Lindsey), нашему эксперту по `LiveData` (и многим другим вопросам, связанным с

Android). Спасибо за особое внимание, которое вы уделили приложениям BeatBox и PhotoGallery.

- Спасибо Эндрю Бейли (Andrew Bailey), лучшему воплощению интеллигенции на свете, за то, что много раз выслушивал и помогал нам принимать непростые концептуальные решения. Также спасибо за помощь в добавлении обновлений Oreo в нашу книгу.

- Спасибо Джейми Ли (Jamie Lee), нашему экстраординарному стажеру-разработчику-редактору. Спасибо за редактирование иллюстраций, обзор решений и внесение замечаний. Ваше внимание к деталям бесценно и уникально.

- Спасибо Эндрю Маршаллу (Andrew Marshall) за активное содействие в улучшении книги, за быстрое создание одного из наших классов, а также за быстрое включение в процесс и помощь в редактировании иллюстраций.

- Спасибо Заку Саймону (Zack Simon), нашему фантастически талантливому дизайнеру с вкрадчивым голосом, который украсил изящную шпаргалку, приложенную к этой книге. Если вам она понравится, вы должны найти Зака и сообщить ему об этом. А мы от себя поблагодарим Зака прямо здесь: спасибо, Зак!

- Спасибо нашему редактору Элизабет Холадей (Elizabeth Holaday). Легендарный битник Уильям Берроуз порой писал так: нарезал рукопись на фрагменты, подбрасывал в воздух и складывал их так, как они упали. Если бы не Элизабет, мы бы тоже скатились до таких методов, отчасти от путаницы, отчасти от простодушного воодушевления. Благодарим за

то, что внесла в нашу работу точность и ясность и помогла сфокусироваться.

- Спасибо Элли Фолькхаузен (Ellie Volckhausen) за дизайн обложки этой книги.

- Анна Бентли (Anna Bentley) — наш замечательный редактор и корректор. Спасибо за то, что отшлифовали текст этой книги.

- Крис Лопер (Chris Loper) из **IntelligentEnglish.com** сверстал и подготовил печатную и электронную версии книги. Его инструментарий DocBook также существенно упростил нашу жизнь.

- Спасибо Аарону Хиллегасу (Aaron Hillegass), Стейси Генри (Stacy Henry) и Эмили Герман (Emily Herman). С практической точки зрения было бы невозможно проделать всю эту работу без Big Nerd Ranch, компании Аарона под бесстрашным руководством Стейси и Эмили в роли исполнительного и операционного директоров. Благодарим вас!

Остается лишь поблагодарить наших студентов. Между нами существует обратная связь: мы преподаем материал, они высказывают свое мнение. Без этой обратной связи книга не появилась бы на свет и не была бы переиздана. Если книги Big Nerd Ranch действительно отличаются от других (как мы надеемся), то именно благодаря этой обратной связи. Спасибо вам!

Изучение Android

Начинающему программисту Android предстоит основательно потрудиться. Изучать Android — все равно что жить в другой стране: даже если вы говорите на местном языке, на первых порах вы все равно не чувствуете себя как дома. Такое впечатление, что все окружающие понимают что-то такое, чего вы еще не усвоили. И даже то, что уже известно, в новом контексте оказывается попросту неправильным.

У Android существует определенная культура. Носители этой культуры общаются на Kotlin или Java (или сразу на обоих), но знать Kotlin или Java недостаточно. Чтобы понять Android, необходимо изучить много новых идей и приемов. Когда оказываешься в незнакомой местности, полезно иметь под рукой путеводитель.

Здесь на помощь приходим мы.

Мы, сотрудники Big Nerd Ranch, считаем, что каждый программист Android должен:

- *писать* приложения для Android;
- *понимать*, что он пишет.

Эта книга поможет вам в достижении обеих целей. Мы обучали тысячи профессиональных программистов Android. Мы проведем вас по пути разработки нескольких Android-приложений, описывая новые концепции и приемы по мере надобности. Если на пути нам встретятся какие-то трудности, если что-то покажется слишком сложным или нелогичным, мы постараемся объяснить, почему все именно так и не иначе.

Такой подход позволит вам с ходу применить полученные сведения, вместо того чтобы, накопив массу теоретических знаний, разбираться, как же их использовать на практике. Перевернув последнюю страницу, вы будете обладать опытом, необходимым для дальнейшей работы в качестве Android-разработчика.

Подготовка

Для работы с этой книгой читатель должен быть знаком с языком Kotlin, включая такие концепции, как классы и объекты, интерфейсы, слушатели, пакеты, внутренние классы, анонимные внутренние классы и обобщенные классы.

Без знания этих концепций вы почувствуете себя в джунглях начиная со второй страницы. Лучше начните с вводного учебника по Kotlin и вернитесь к этой книге после его прочтения. Сегодня существует много превосходных книг для начинающих; подберите нужный вариант в зависимости от своего опыта программирования и стиля обучения. Мы рекомендуем книгу *Kotlin Programming: The Big Nerd Ranch Guide*[1].

Если вы хорошо разбираетесь в концепциях объектно-ориентированного программирования, но успели малость подзабыть Kotlin, скорее всего, все будет нормально. Мы приводим краткие напоминания о некоторых специфических возможностях Kotlin (таких как интерфейсы и анонимные внутренние классы). Держите учебник по Kotlin наготове на случай, если вам понадобится дополнительная информация во время чтения.

Что нового в четвертом издании?

В этом издании мы провели капитальный ремонт и изменили буквально каждую главу. Самое большое изменение заключается в том, что программы теперь написаны на Kotlin, а не на Java. Поэтому неофициальным рабочим названием этого издания было «Android 4K».

Еще одно радикальное изменение — включение библиотек компонентов Android Jetpack. Теперь мы используем Jetpack-библиотеки (их еще называют AndroidX) вместо Support Library. Кроме того, мы включили новые API Jetpack, где это было уместно. Например, мы используем `ViewModel` для сохранения состояния пользовательского интерфейса при вращении. Мы используем Room и `LiveData` для реализации базы данных и запросов данных из нее. А для планирования фоновой работы мы используем WorkManager. И это лишь часть нововведений. В этой книге компоненты Jetpack в той или иной мере вплетены во все проекты.

Чтобы сфокусироваться на том, как разрабатываются современные приложения для Android, в этой книге используются библиотеки сторонних разработчиков, а не только API в пределах данного фреймворка или Jetpack. Один из примеров — отказ от `HttpURLConnection` и других сетевых API нижнего уровня в пользу использования Retrofit и его зависимых библиотек. Мы тем самым сильно отходим от наших предыдущих книг, но считаем, что такой подход подготовит вас к погружению в профессиональную разработку приложений после прочтения нашей книги. Выбранные библиотеки мы используем в повседневной жизни, разрабатывая приложения на Android для наших клиентов.

Kotlin vs Java

Официальная поддержка языка Kotlin для разработки под Android была объявлена на Google I/O в 2017 году. До этого существовало небольшое движение разработчиков Android, использующих Kotlin, несмотря на отсутствие официальной поддержки. С 2017 года Kotlin получил широкое распространение, и сегодня это наиболее предпочтительный язык для разработчиков на Android. Мы в Big Nerd Ranch используем Kotlin во всех наших проектах по разработке приложений, даже для старых проектов, которые в основном написаны на Java.

Взлеты и падения Kotlin весьма заметны. Команда разработчиков фреймворка Android начала добавлять в код старой платформы аннотации `@nullable`. Они также выпускают все больше и больше расширений Kotlin для Android. На момент написания этого абзаца Google уже добавляет примеры Kotlin в официальную документацию Android.

Фреймворк Android был первоначально написан на Java. Это означает, что большинство классов Android, с которыми вы взаимодействуете, — это классы Java. К счастью, Kotlin совместим с Java, поэтому проблем с этим не возникнет.

Листинги API мы тоже приводим на Kotlin, даже если внутри они реализованы на Java. Вы можете посмотреть листинги Java API, найдя интересующий вас класс на странице **developer.android.com/reference**.

Будь вы фанат Kotlin или Java, эта книга научит вас, как писать приложения для Android. Знания и опыт, полученные при разработке приложений для платформы Android, отлично транслируются на любой язык.

Как работать с книгой

Эта книга не справочник. Мы старались помочь в преодолении начального барьера, чтобы вы могли извлечь максимум пользы из существующих справочников и пособий. Книга основана на материалах пятидневного учебного курса в Big Nerd Ranch. Соответственно предполагается, что вы будете читать ее с самого начала. Каждая глава базируется на предшествующем материале, и пропускать главы не рекомендуется.

На наших занятиях студенты прорабатывают эти материалы, но в обучении также важно и другое — ваш настрой и обстановка.

Желательно, чтобы ваша учебная среда была похожа на нашу. В частности, стоит хорошенько высыпаться и найти спокойное место для работы. Следующие факторы тоже сыграют положительную роль:

- Создайте учебную группу с друзьями или коллегами.
- Выделяйте время, когда вы будете заниматься исключительно чтением книги.
- Примите участие в работе форума книги на сайте **forums.bignerdranch.com**.
- Найдите специалиста по Android, который поможет вам в трудный момент.

Структура книги

В этой книге мы напишем семь приложений для Android. Два приложения очень просты, и на их создание уходит всего одна глава. Другие приложения часто оказываются более сложными, а самое длинное приложение занимает 11 глав. Все приложения спроектированы так, чтобы продемонстрировать

важные концепции и приемы и дать опыт их практического применения.

GeoQuiz

В первом приложении мы исследуем основные принципы создания проектов Android, activity, макеты и явные интенты. Вы также научитесь без проблем работать с конфигурацией.

CriminalIntent

Самое большое приложение в книге предназначено для хранения информации о проступках ваших коллег по офису. Вы научитесь использовать фрагменты, интерфейсы «главное-детализированное представление», списковые интерфейсы, меню, камеру, неявные интенты и многое другое.

BeatBox

Наведите ужас на своих врагов и узнайте больше о фрагментах, воспроизведении мультимедийного контента, архитектуре MVVM, связывании данных, тестировании, темах и графических объектах.

NerdLauncher

Нестандартный лаунчер раскроет тонкости работы системы интентов, процессов и задач.

PhotoGallery

Клиент Flickr для загрузки и отображения фотографий из общедоступной базы Flickr. Приложение демонстрирует работу со службами, многопоточное программирование, обращения к веб-службам и т.д.

DragAndDraw

В этом простом графическом приложении рассматривается обработка событий касания и создание нестандартных представлений.

Sunset

В этом крохотном приложении вы создадите красивое представление заката над водой, а заодно освоите тонкости анимации.

Упражнения

Многие главы завершаются разделом с упражнениями. Это ваша возможность применить полученные знания, покопаться в документации и отработать навыки самостоятельного решения задач.

Мы настоятельно рекомендуем выполнять упражнения. Сойдя с проторенного пути и найдя собственный способ, вы закрепите учебный материал и приобретете уверенность в работе над собственными проектами.

Если же окажетесь в тупике, вы всегда сможете обратиться за помощью на форум **forums.bignerdranch.com**.

А вы любознательны?

В конце многих глав также имеется раздел «Для любознательных». В нем приводятся углубленные объяснения или дополнительная информация по темам, представленным в главах. Содержимое этих разделов читать необязательно, но мы надеемся, что оно покажется вам интересным и полезным.

Типографские соглашения

Все листинги с кодом и разметкой XML напечатаны моноширинным шрифтом. Код или разметка XML, которые вы должны ввести, выделяются жирным шрифтом. Код или разметка XML, которые нужно удалить, зачеркиваются.

Например, в следующей реализации функции вызов `Toast.makeText(...).show()` удаляется, а вместо него добавляется вызов `checkAnswer(true)`.

```
trueButton.setOnClickListener { view: View ->
        Toast.makeText(
            this,
            R.string.correct_toast,
            Toast.LENGTH_SHORT
        )
            .show()
    checkAnswer(true)
    }
```

Версии Android

Мы будем говорить о версиях Android, широко используемых на момент написания книги. Для данного издания это версии Android 5.0 (Lollipop, API уровня 21) — Android 9.0 (Pie, API уровня 28). Несмотря на то что более старые версии все еще используются, нам кажется, что усилия, требуемые для поддержки этих версий, того не стоят.

Если вы хотите получить информацию о поддержке версий Android ранее 5.0, вы можете почитать предыдущие издания этой книги. Третье издание было нацелено на Android 4.4 и выше, второе — на Android 4.1 и выше, а первое — на Android 2.3 и выше.

Даже после выхода новых версий Android приемы, изложенные в книге, будут работать благодаря политике обратной совместимости Android (подробности см. в главе 7). На сайте **forums.bignerdranch.com** будет публиковаться

информация об изменениях, а также комментарии по поводу использования материала книги с последними версиями.

От издательства

Если вам понравилась книга, ознакомьтесь с другими книгами издательства «Питер» по адресу **www.piter.com**. На веб-сайте издательства вы найдете подробную информацию о наших книгах.

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

Мы будем рады узнать ваше мнение!

1 Скин Джош, Гринхол Дэвид. Kotlin. Программирование для профессионалов. — СПб.: Питер, 2020. — 464 с.: ил. — *Примеч. ред.*

Необходимые инструменты

Прежде всего вам понадобится Android Studio — интегрированная среда разработки для Android-программирования, созданная на базе популярной среды IntelliJ IDEA.

Установка Android Studio включает в себя:

Android SDK

последнюю версию Android SDK.

Инструменты и платформенные средства Android SDK

средства отладки и тестирования приложений.

Образ системы для эмулятора Android,

который позволяет создавать и тестировать приложения на различных виртуальных устройствах.

На момент написания книги среда Android Studio находилась в активной разработке и часто обновлялась. Учтите, что ваша версия Android Studio может отличаться от описанной в книге. За информацией об отличиях обращайтесь на сайт **forums.bignerdranch.com**.

Загрузка и установка Android Studio

Пакет Android Studio доступен на сайте разработчиков Android по адресу **developer.android.com/studio**.

Возможно, вам также придется установить пакет Java Development Kit (JDK8), если он еще не установлен в вашей системе; его можно загрузить на сайте **www.oracle.com**.

Если у вас все равно остаются проблемы, обратитесь по адресу **developer.android.com/studio** за дополнительной информацией.

Загрузка старых версий SDK

Android Studio предоставляет SDK и образ эмулируемой системы для последней платформы. Однако не исключено, что вы захотите протестировать свои приложения в более ранних версиях Android.

Компоненты любой платформы можно получить при помощи инструмента Android SDK Manager. В Android Studio выполните команду **Tools⇒SDKManager**. (Меню **Tools** отображается только при наличии открытого проекта. Если вы еще не создали проект, SDK Manager также можно вызвать из окна **AndroidWelcome**; в разделе **QuickStart** выберите вариант **Configure⇒SDKManager**, как показано на рис. 1.)

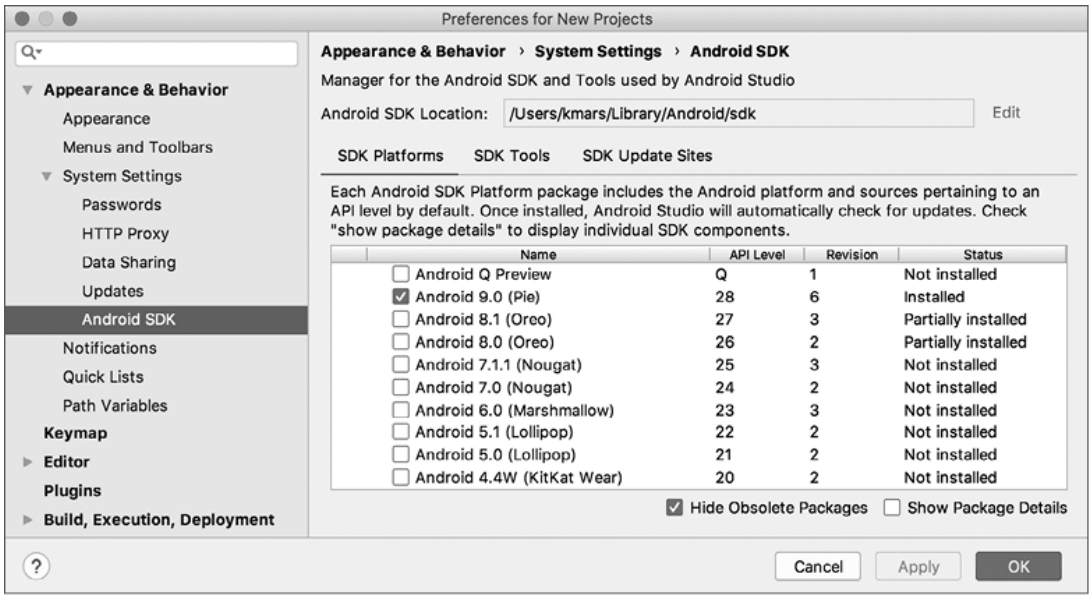


Рис. 1. Android SDK Manager

Выберите и установите каждую версию Android, которая будет использоваться при тестировании. Учтите, что загрузка этих компонентов может потребовать времени.

Android SDK Manager также используется для загрузки новейших выпусков Android — например, новой платформы или обновленных версий инструментов.

Аппаратное обеспечение

Эмулятор удобен для тестирования приложений. Тем не менее он не заменит реальное устройство на базе Android для оценки быстродействия. Если у вас имеется физическое устройство, мы рекомендуем время от времени использовать его в ходе работы с книгой.

1. Первое Android-приложение

В первой главе даются новые концепции и составляющие, необходимые для разработки Android-приложений. Не беспокойтесь, если к концу главы что-то останется непонятным, — это нормально. Мы еще вернемся к этим концепциям в последующих главах и рассмотрим их более подробно.

Приложение, которое мы создадим, называется GeoQuiz. Оно проверяет, насколько хорошо пользователь знает географию. Пользователь отвечает на вопрос, нажимая кнопку **True** или **False**, а GeoQuiz мгновенно сообщает ему результат.

На рис. 1.1 показан результат нажатия кнопки **True**.

Основы разработки приложения

Приложение GeoQuiz состоит из activity и макета (layout):

- Activity представлена экземпляром `Activity` — классом из Android SDK. Она отвечает за взаимодействие пользователя с информацией на экране.

- Чтобы реализовать функциональность, необходимую приложению, разработчик пишет подклассы `Activity`. В простом приложении бывает достаточно одного подкласса; в сложном приложении их может потребоваться несколько.

- GeoQuiz — простое приложение, поэтому в нем используется всего один подкласс `Activity` — `MainActivity`. Класс `MainActivity` управляет пользовательским интерфейсом, изображенным на рис. 1.1.

- Макет определяет набор объектов пользовательского интерфейса и их расположение на экране. Макет формируется из определений, написанных на языке XML. Каждое определение используется для создания объекта, выводимого на экране (например, кнопки или текста).

- Приложение GeoQuiz включает файл макета `activity_main.xml`. Разметка XML в этом файле определяет пользовательский интерфейс, изображенный на рис. 1.1.

Рис. 1.1. А вы знаете географию Зеленого континента?

Отношения между `MainActivity` и `activity_main.xml` изображены на рис. 1.2.

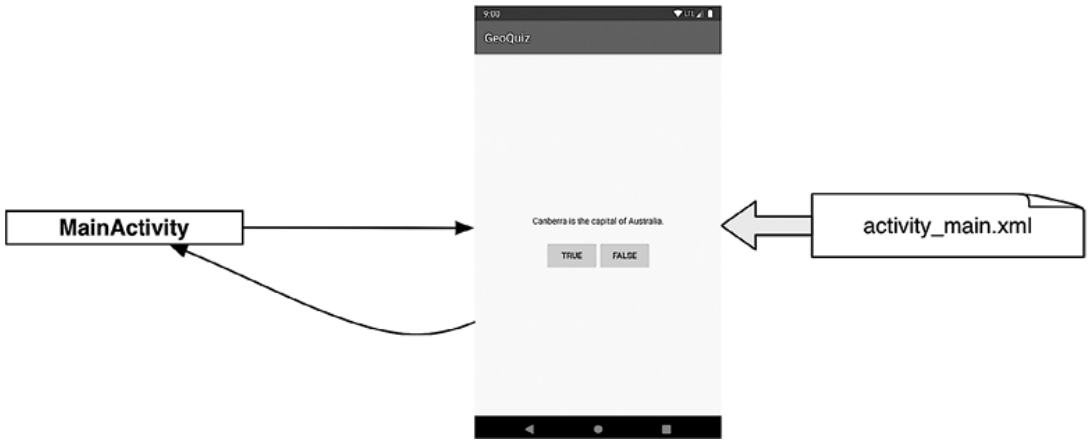


Рис. 1.2. MainActivity управляет интерфейсом, определяемым в файле activity_main.xml

Учитывая все сказанное, давайте соберем приложение.

Создание проекта Android

Работа начинается с создания *проекта* Android. Проект Android содержит файлы, из которых состоит приложение. Чтобы создать новый проект, откройте Android Studio.

Если Android Studio запускается на вашем компьютере впервые, на экране появляется диалоговое окно с приветствием (рис. 1.3).

Рис. 1.3. Добро пожаловать в Android Studio

Перед тем как начать работу над новым проектом, стоит отключить все функции, которые могут помешать вашей работе. Функция **InstantRun** предназначена для упрощения разработки, позволяя сразу внедрять изменения кода без создания нового файла APK. К сожалению, она не всегда работает, как задумано, поэтому мы рекомендуем отключить ее на время работы над этой книгой.

В нижней части окна приветствия нажмите кнопку **Configure**, затем выберите пункт **Settings**. В левой части окна **PreferencesforNewProjects** (рис. 1.4) разверните список **Build,Execution,Deployment** и выберите вкладку **InstantRun**. Сбросьте флажок **EnableInstantRuntohotswapcode/resourcechangesondeploy** (по умолчанию эта функция включена) и нажмите кнопку **OK**.

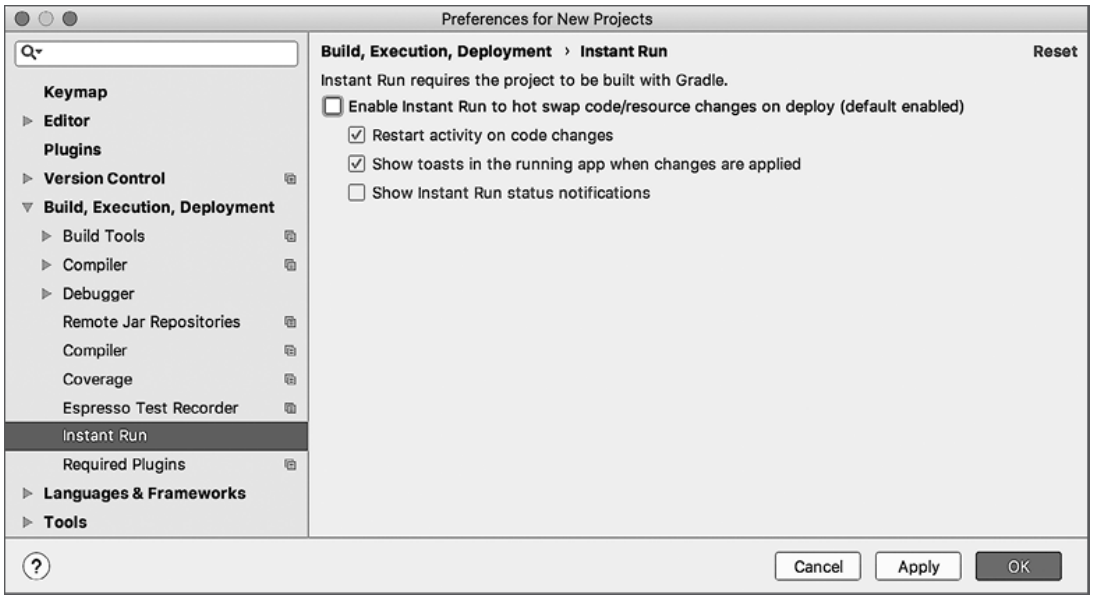


Рис. 1.4. Настройки нового проекта

(Если вы ранее использовали Android Studio и не видели приветственного окна при запуске, перейдите в раздел **AndroidStudio⇒Preferences**, затем откройте список **Build,Execution,Deployment** и дальше по инструкции выше.)

В окне приветствия нажмите кнопку **StartanewAndroidStudioproject**. Если окно приветствия не появилось, выберите команду меню **File⇒New⇒NewProject**.

Перед вами откроется мастер создания нового проекта (рис. 1.5). Убедитесь, что выбрана вкладка **PhoneandTablet**, выберите шаблон **EmptyActivity** и нажмите кнопку **Next**.

Теперь перед вами появится экран **Configureyourproject** (рис. 1.6). Задайте для нового приложения имя **GeoQuiz**. Задайте для пакета имя **com.bignerdranch.android.geomain**. В качестве расположения проекта используйте любой каталог в вашей файловой системе.

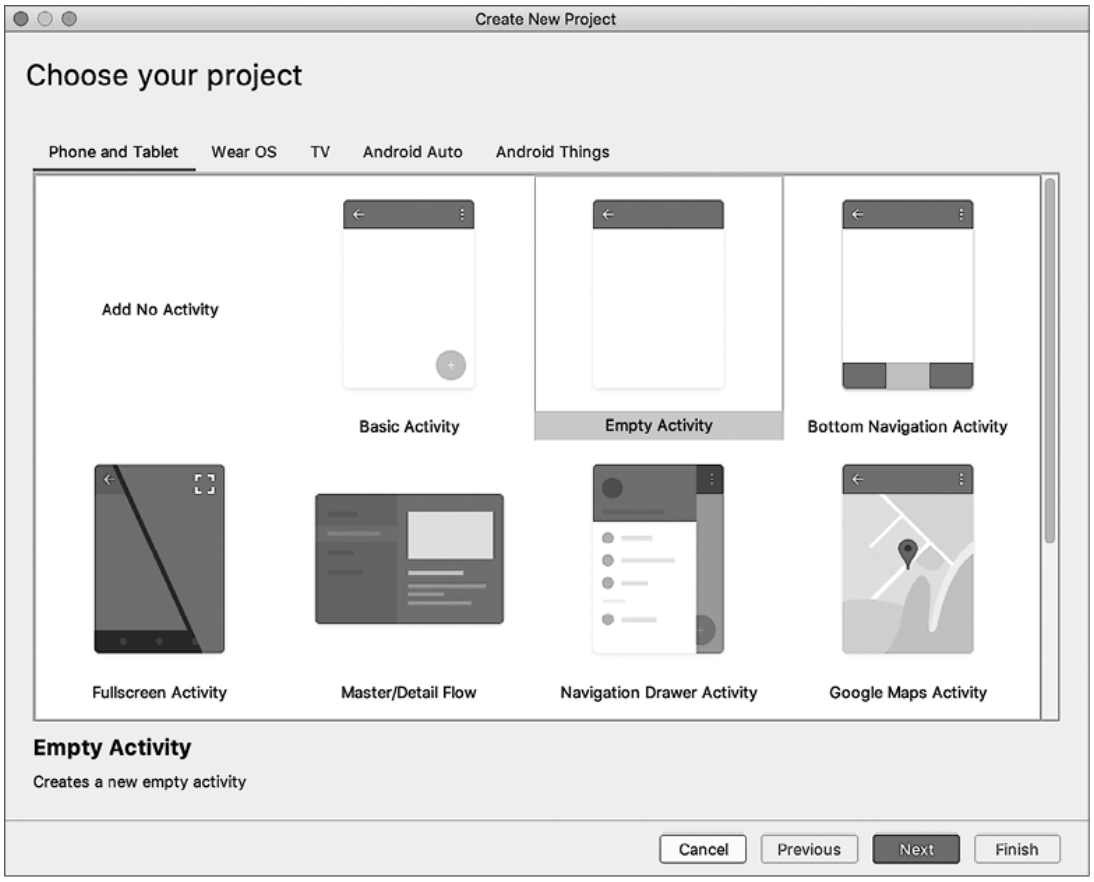


Рис. 1.5. Выбор шаблона проекта

Выберите язык **Kotlin** в раскрывающемся списке **Language**. В раскрывающемся списке **MinimumAPIlevel** выберите пункт **API 21:Android5.0(Lollipop)**. О различных версиях Android мы поговорим в главе 7. Наконец, убедитесь, что установлен флажок **UseAndroidXartifacts**. Ваш экран должен выглядеть так, как показано на рис. 1.6.

Обратите внимание, что в имени пакета используется соглашение об именовании «обратный DNS»: доменное имя вашей организации переворачивается и добавляется слева, вместе с дополнительными идентификаторами. Это соглашение позволяет делать имена пакетов уникальными и отличать приложения на устройстве и в магазине Google Play Store.

На момент написания книги новые проекты, сгенерированные Android Studio, по умолчанию создаются на Java. Если вы выберете язык Kotlin, то Android Studio подключит к проекту соответствующие зависимости, такие как инструменты сборки Kotlin, необходимые для написания и сборки кода Kotlin.

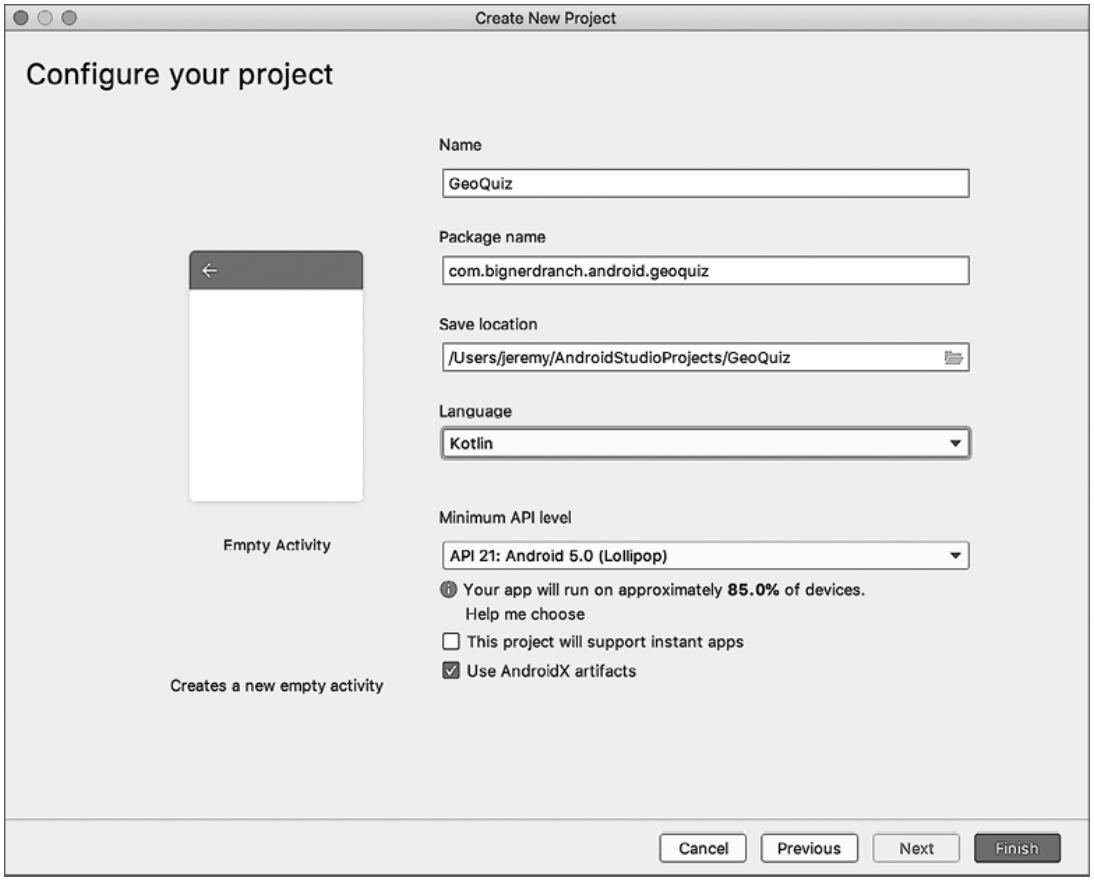


Рис. 1.6. Настройка нового проекта

Java был единственным официально поддерживаемым языком разработки до мая 2017 года, когда команда разработчиков Android анонсировала официальную поддержку Kotlin для разработки под Android на Google I/O. На сегодняшний день Kotlin — предпочтительный язык для большинства разработчиков, и мы не исключение, поэтому в этой книге используется Kotlin. Но если вы решите

использовать Java, общие понятия и концепции работы с Android все равно останутся полезны и применимы.

Раньше Google предоставлял монолитную «поддерживающую» библиотеку справок и надстроек для совместимости. В AndroidX эта библиотека была поделена на множество отдельно разрабатываемых и разделенных на версии библиотек, совместно называемых Jetpack. Выбор опции **UseAndroidXartifacts** позволяет подготовить ваш проект к использованию новых независимых инструментов. Вы узнаете больше о Jetpack и AndroidX в главе 4, а в этой книге мы будем использовать различные Jetpack-библиотеки.

(Android Studio регулярно обновляется, поэтому внешний вид мастера настройки может отличаться от того, что показано в этой книге. Обычно это не проблема, и все настройки следует выбирать точно так же. Если ваш мастер выглядит совсем по-другому, это значит, что и инструментарий в нем сильно поменялся. Не паникуйте. Зайдите на форум этой книги по ссылке **forum.bignerdranch.com**, и мы поможем вам сориентироваться в последней версии.)

Нажмите кнопку **Finish**, после чего Android Studio создаст и откроет ваш новый проект.

Навигация в Android Studio

Android Studio открывает новый проект в окне, показанном на рис. 1.7. Если вы уже запускали Android Studio ранее, то конфигурация вашего окна может выглядеть несколько иначе.

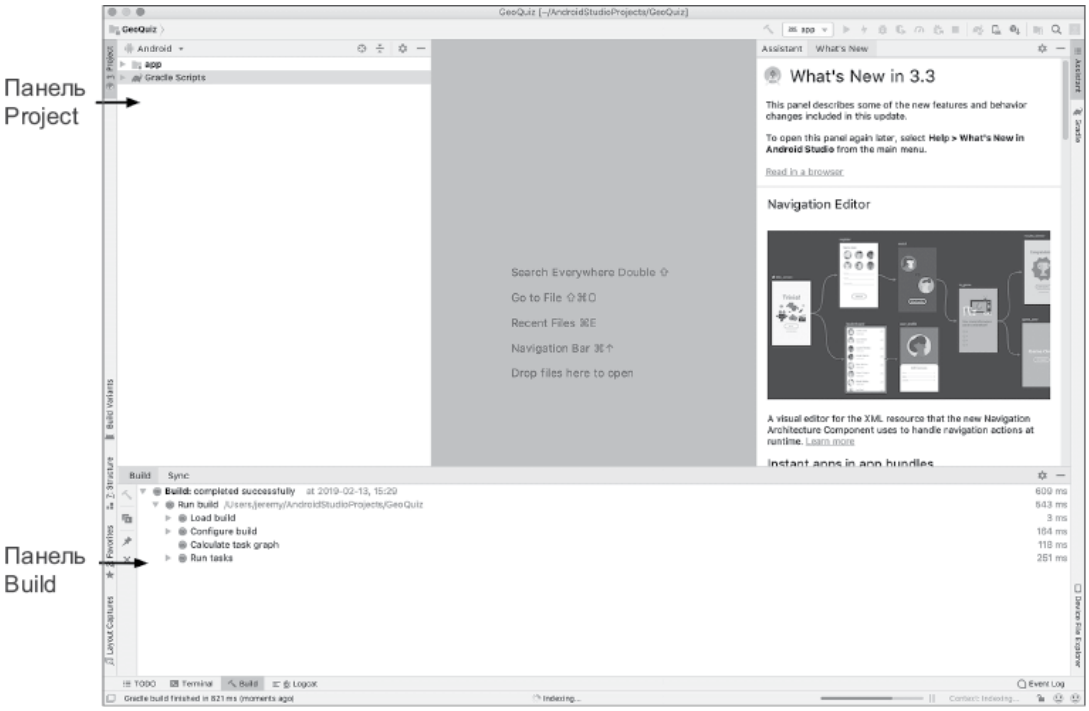


Рис. 1.7. Окно нового проекта

Различные части окна проекта называются *панелями*.

Слева располагается *панель* **Project**. На ней выполняются операции с файлами, относящимися к вашему проекту.

Снизу находится *панель* **Build**. Здесь вы можете просмотреть подробную информацию о процессе компиляции и состоянии сборки. Создавая новый проект, Android Studio автоматически собирает его. На панели **Build** отображается информация о том, что процесс сборки завершился успешно.

На панели **Project** раскройте пункт **app**. Android Studio автоматически откроет файлы `active_main.xml` и `MainActivity.kt` на основной панели, называемой *панелью редактора* или просто *редактором* (рис. 1.8). Если вы не в первый раз открываете Android Studio, то панель редактора могла открыться автоматически при создании проекта.

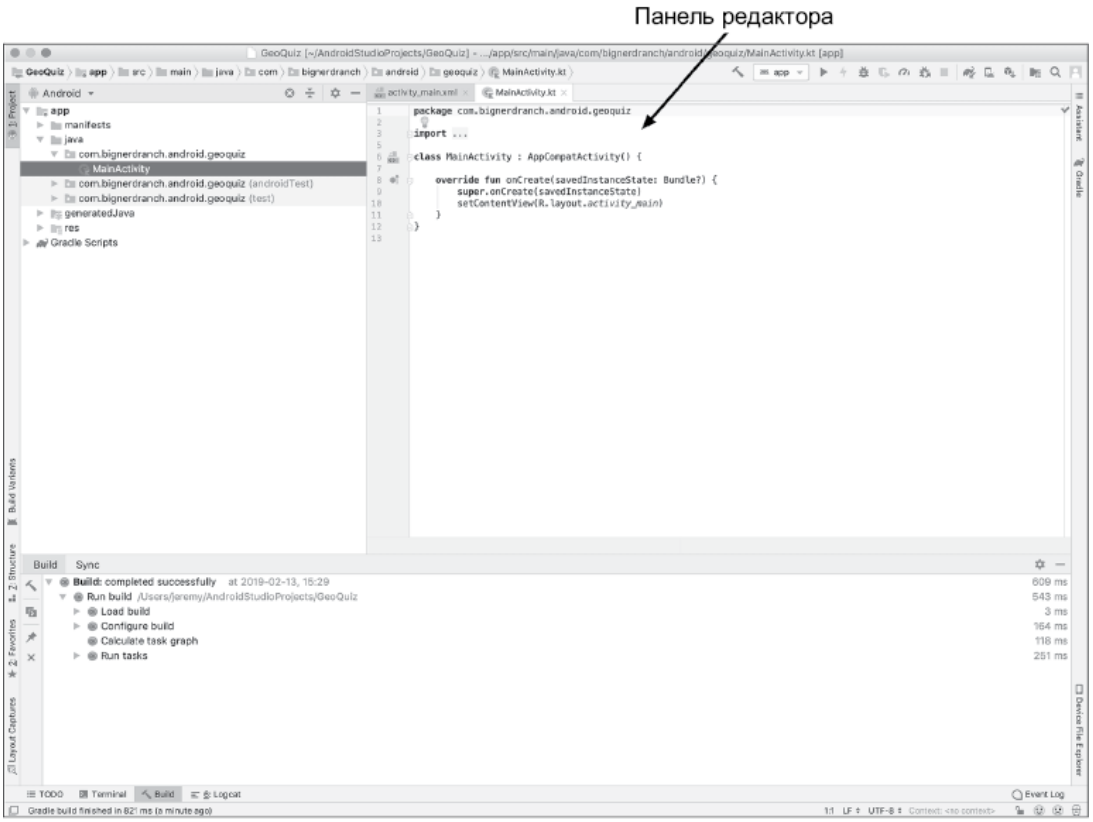


Рис. 1.8. Интерфейс с панелью редактора

Обратите внимание на суффикс `Activity` возле имени класса. Он не обязательный, но использовать его крайне рекомендуется.

Отображением различных панелей можно управлять, щелкая по вкладкам с соответствующими названиями у левого, правого или нижнего края экрана. Также для многих панелей определены специальные комбинации клавиш. Если полосы с кнопками не отображаются, щелкните мышью по серой квадратной кнопке в левом нижнем углу главного окна или выполните команду **View⇒ToolButtons**.

Создание макета пользовательского интерфейса

Перейдите на вкладку файла макета `activity_main.xml`. На панели редактора откроется редактор макетов (рис. 1.9). Если вы не видите вкладку `activity_main.xml`, не волнуйтесь. Откройте папку `app/res/layout/` на панели **Project**. Дважды щелкните мышью по файлу `activity_main.xml`, чтобы открыть его. Если вместо редактора макетов откроется вкладка `active_main.xml` с XML-файлом вместо макета, щелкните по вкладке **Design** в нижней части панели редактора.

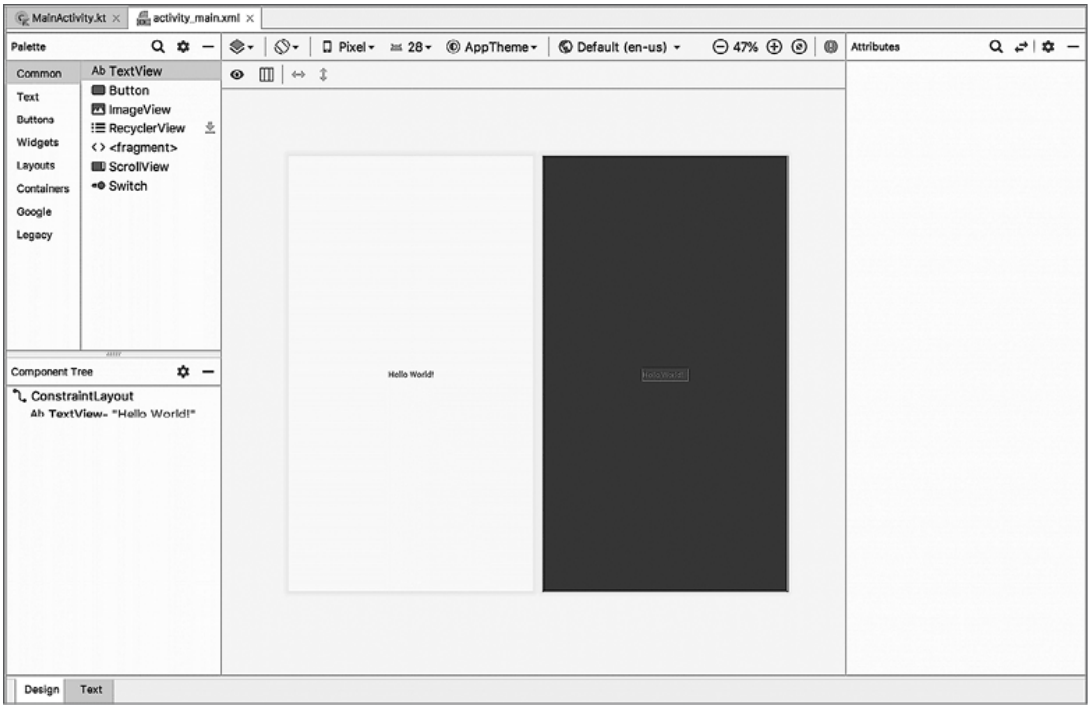


Рис. 1.9. Редактор макета

По соглашению имя файла макета выбирается в зависимости от activity, с которой он связан: его имя начинается со слова `activity_`, а остаток имени activity пишется строчными буквами с использованием нижнего подчеркивания для разделения слов (стиль «змеиный регистр»). Так, например, ваш файл макета называется `activity_main.xml`, файл макета для activity под названием `SplashScreenActivity` будет называться

activity_splash_screen. Змеиный регистр рекомендуется использовать и для макетов, и для других ресурсов, о которых вы узнаете позже.

В редакторе макетов отображается предварительный вид макета. Выберите вкладку **Text** внизу, чтобы увидеть его XML-вид.

В настоящее время файл activity_main.xml определяет разметку для activity по умолчанию. Такая разметка часто изменяется, но XML-код будет выглядеть примерно так, как показано в листинге 1.1.

Листинг 1.1. Разметка activity по умолчанию (res/layout/activity_main.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLay
out
        xmlns:android="http://schemas.android.c
om/apk/res/android"
        xmlns:tools="http://schemas.android.com
/tools"
        xmlns:app="http://schemas.android.com/a
pk/res-auto"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
```

```
        app:layout_constraintBottom_toBottomOf=
"parent"
        app:layout_constraintLeft_toLeftOf="par
ent"
        app:layout_constraintRight_toRightOf="p
arent"
        app:layout_constraintTop_toTopOf="paren
t"/>

</androidx.constraintlayout.widget.ConstraintLa
yout>
```

У макета activity по умолчанию есть два *представления* (*view*): `ConstraintLayout` и `TextView`.

Представления — это структурные элементы, из которых составляется пользовательский интерфейс. Все, что вы видите на экране, — это представление. Представления, которые видит пользователь или с которыми взаимодействует, называются *виджетами*.

Одни виджеты могут выводить текст, другие — графику. Третьи, например кнопки, выполняют действия при касании.

В Android SDK предусмотрено множество виджетов, которые можно настраивать для получения нужного оформления и поведения. Каждый виджет является экземпляром класса `View` или одного из его подклассов (например, `TextView` или `Button`).

Где-то должна содержаться информация о том, где место каждого виджета на экране. Объект `ViewGroup` — это такой подвид `View`, который содержит и упорядочивает другие виджеты. При этом сам `ViewGroup` контент не отображает. Он распоряжается там, где отображается содержимое других

представлений. Объекты ViewGroup еще часто называют макетами.

В макете activity по умолчанию ConstraintLayout — это ViewGroup, отвечающий за расположение своего единственного дочернего элемента, виджета TextView. Вы узнаете больше о макетах и виджетах, а также об использовании ConstraintLayout в главе 10.

На рис. 1.10 показано, как выглядят на экране виджеты ConstraintLayout и TextView, определенные в листинге 1.1.

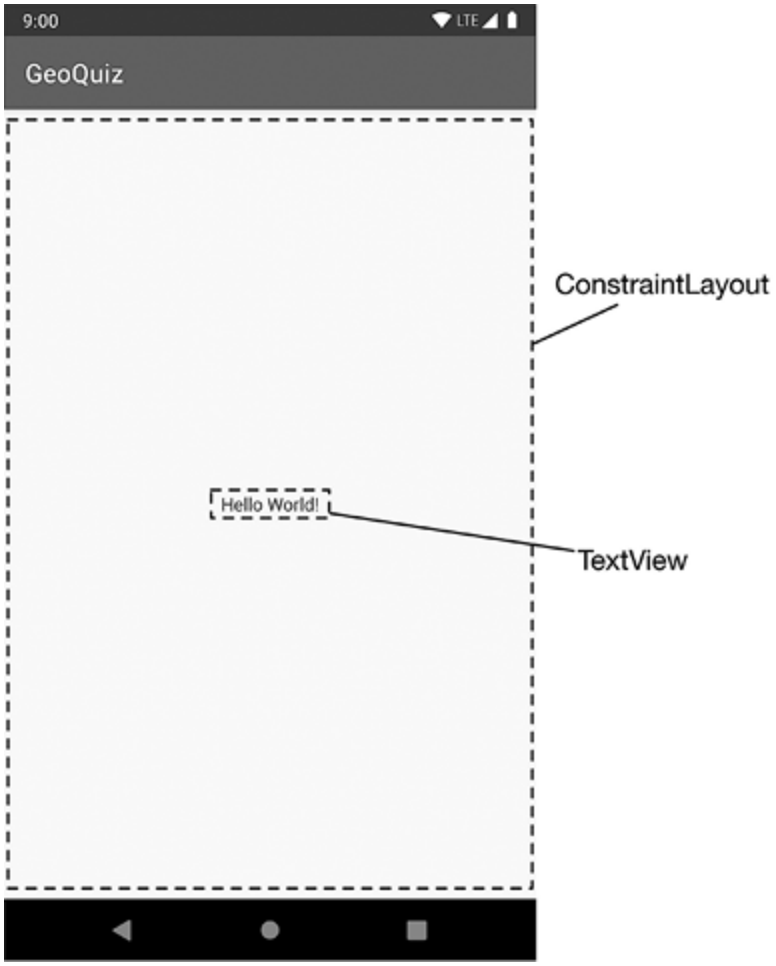


Рис. 1.10. Представления по умолчанию на экране

Впрочем, это не все виджеты. В интерфейсе MainActivity задействованы пять виджетов:

• вертикальный виджет LinearLayout;

• TextView;

• горизонтальный виджет LinearLayout;

• две кнопки Button.

На рис. 1.11 показано, как из этих виджетов образуется интерфейс MainActivity.

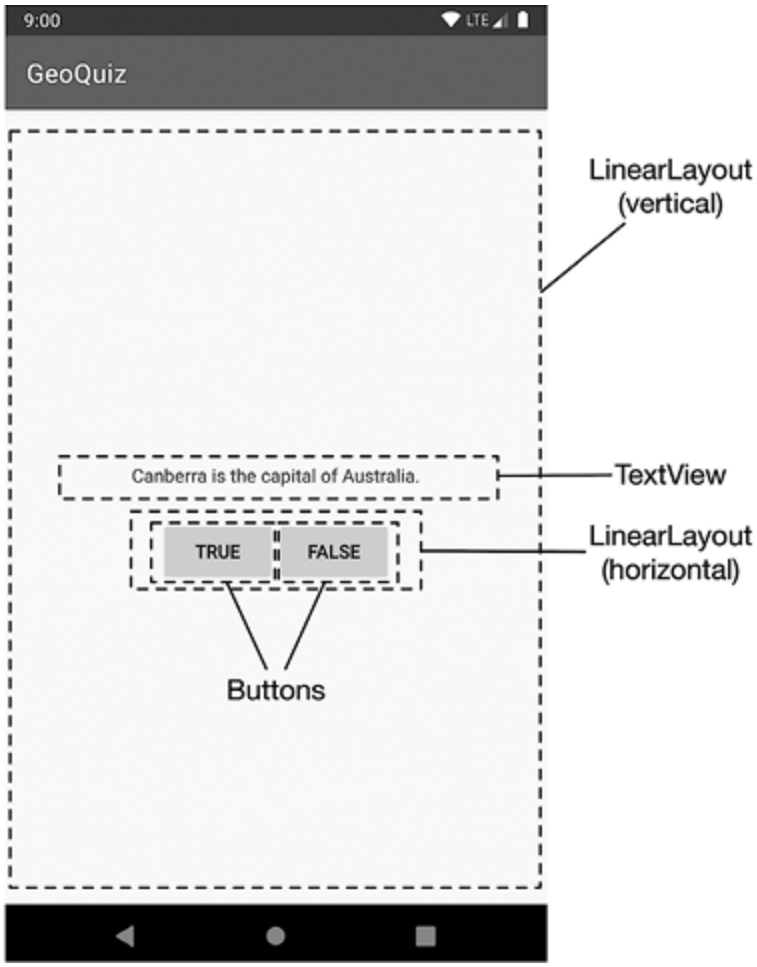


Рис. 1.11. Запланированное расположение виджетов на экране

Теперь нужно определить эти виджеты в файле `activity_main.xml`.

Внесите изменения, представленные в листинге 1.2. Разметка XML, которую нужно удалить, выделена перечеркиванием, а добавляемая разметка XML — жирным шрифтом. Эти обозначения будут использоваться во всей книге.

Не беспокойтесь, если смысл вводимой разметки остается непонятным; скоро вы узнаете, как она работает. Но будьте внимательны: разметка макета не проверяется, и опечатки рано или поздно вызовут проблемы.

В трех строках, начинающихся с `android:text`, будут обнаружены ошибки. Пока не обращайте внимания, мы их скоро исправим.

Листинг 1.2. Определение виджетов в XML (res/layout/activity_main.xml)

```
~~<androidx.constraintlayout.widget.ConstraintLayout~~
~~xmlns:android="http://schemas.android.com/apk/res/android"~~
~~xmlns:tools="http://schemas.android.com/tools"~~
~~xmlns:app="http://schemas.android.com/apk/res-auto"~~
~~android:layout_width="match_parent"~~
~~android:layout_height="match_parent"~~
~~tools:context=".MainActivity">~~

    ~~<TextView~~
```

~~android:layout_width="wrap_content"~~
~~android:layout_height="wrap_content"~~
~~android:text="Hello World!"~~
~~app:layout_constraintBottom_toBottomOf="parent"~~
~~app:layout_constraintLeft_toLeftOf="parent"~~
~~app:layout_constraintRight_toRightOf="parent"~~
~~app:layout_constraintTop_toTopOf="parent"/>~~

~~</androidx.constraintlayout.widget.ConstraintLayout>~~

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        android:text="@string/question_text" />

    <LinearLayout
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content
"
             android:text="@string/true_button"
/>

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content
"
            android:text="@string/false_button"
/>

    </LinearLayout>

</LinearLayout>
```

Сравните XML с пользовательским интерфейсом, изображенным на рис. 1.11. Каждому виджету в разметке соответствует элемент XML. Имя элемента определяет тип виджета.

Каждый элемент обладает набором *атрибутов* XML. Атрибуты можно рассматривать как инструкции по настройке виджетов.

Чтобы лучше понять, как работают элементы и атрибуты, полезно взглянуть на разметку с точки зрения иерархии.

Иерархия представления

Виджеты входят в иерархию объектов `View`, называемую *иерархией представления*. На рис. 1.12 изображена иерархия виджетов для разметки XML из листинга 1.2.

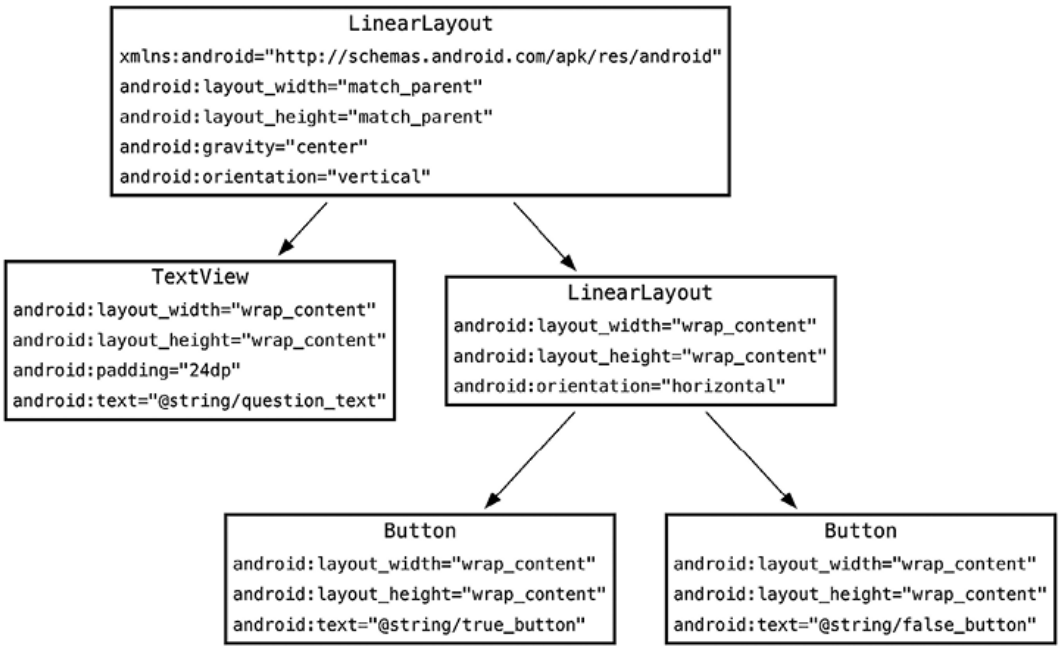


Рис. 1.12. Иерархия виджетов и атрибутов

Корневым элементом иерархии представления является элемент `LinearLayout`. В нем должно быть указано пространство имен XML-ресурсов Android `http://schemas.android.com/apk/res/android`.

`LinearLayout` наследует от подкласса `View` виджет `ViewGroup`, который предназначен для хранения и размещения других виджетов. `LinearLayout` используется в тех случаях, когда вы хотите выстроить виджеты в один столбец или строку. Другие подклассы `ViewGroup` — `ConstraintLayout` и `FrameLayout`.

Если виджет содержится в `ViewGroup`, он называется *потомком*`ViewGroup`. Корневой элемент `LinearLayout` имеет

двух потомков: TextView и другой элемент LinearLayout. У LinearLayout имеются два собственных потомка Button.

Атрибуты виджетов

Рассмотрим некоторые атрибуты, используемые для настройки виджетов.

android:layout_width и android:layout_height

Атрибуты android:layout_width и android:layout_height, определяющие ширину и высоту, необходимы практически для всех разновидностей виджетов. Как правило, им задаются значения match_parent или wrap_content:

match_parent — размеры представления определяются размерами родителя;

wrap_content — размеры представления определяются размерами содержимого.

В корневом элементе LinearLayout атрибуты ширины и высоты равны match_parent. Элемент LinearLayout — корневой, но у него все равно есть родитель — представление Android для размещения иерархии представлений вашего приложения.

У других виджетов макета ширине и высоте задается значение wrap_content. На рис. 1.11 показано, как в этом случае определяются их размеры.

Виджет TextView чуть больше содержащегося в нем текста из-за атрибута android:padding="24dp". Этот атрибут приказывает виджету добавить заданный отступ вокруг содержимого при определении размера, чтобы текст вопроса не соприкасался с кнопкой. (Интересуетесь, что это за единицы —

dp? Это пикселы, не зависящие от плотности (density-independent pixels), о которых будет рассказано в главе 10.)

android:orientation

Атрибут `android:orientation` двух виджетов `LinearLayout` определяет, как будут выстраиваться потомки — по вертикали или горизонтали. Корневой элемент `LinearLayout` имеет вертикальную ориентацию; у его потомка `LinearLayout` горизонтальная ориентация.

Порядок определения потомков устанавливает порядок их отображения на экране. В вертикальном элементе `LinearLayout` потомок, определенный первым, располагается выше остальных. В горизонтальном элементе `LinearLayout` первый потомок является крайним левым. (Если только на устройстве не используется язык с письменностью справа налево, например арабский или иврит; в таком случае первый потомок будет находиться в крайней правой позиции.)

android:text

Виджеты `TextView` и `Button` содержат атрибуты `android:text`. Этот атрибут сообщает виджету, какой текст должен в нем отображаться.

Обратите внимание: значения атрибутов представляют собой не строковые литералы, а ссылки на *строковые ресурсы* с синтаксисом `@string/`.

Строковый ресурс — строка, находящаяся в отдельном файле XML, который называется строковым файлом. Виджету можно назначить фиксированную строку (например, `android:text="True"`), но так делать не стоит. Лучше размещать строки в отдельном файле, а затем ссылаться на них,

так как использование строковых ресурсов упрощает локализацию (см. главу 17).

Строковые ресурсы, на которые мы ссылаемся в `activity_main.xml`, еще не существуют. Давайте исправим этот недостаток.

Создание строковых ресурсов

Каждый проект по умолчанию включает строковый файл `res/values/strings.xml`.

Откройте файл `res/values/strings.xml`. В шаблон уже включен один строковый ресурс. Добавьте три новые строки для вашего макета.

Листинг 1.3. Добавление строковых ресурсов (res/values/strings.xml)

```
<resources>
    <string name="app_name">GeoQuiz</string>
      <string name="question_text">Canberra is
the capital of Australia.</string>
    <string name="true_button">True</string>
    <string name="false_button">False</string>
</resources>
```

(В зависимости от версии Android Studio файл может содержать другие строки. Не удаляйте их — это может породить каскадные ошибки в других файлах.)

Теперь по ссылке `@string/false_button` в любом файле XML проекта GeoQuiz вы будете получать строковый литерал False на стадии выполнения.

Если в файле `activity_main.xml` оставались ошибки, связанные с отсутствием строковых ресурсов, они должны исчезнуть. (Если ошибки остались, проверьте оба файла — возможно, где-то допущена опечатка.)

Строковый файл по умолчанию называется `strings.xml`, но ему можно присвоить любое имя на ваш выбор. Проект может содержать несколько строковых файлов. Если файл находится в каталоге `res/values/`, содержит корневой элемент `resources` и дочерние элементы `string`, ваши строки будут найдены и правильно использованы приложением.

Предварительный просмотр макета

Макет готов (рис. 1.13). Вернитесь к вкладке `active_main.xml` и просмотрите макет на панели **Design**, щелкнув по вкладке в нижней части панели редактора (рис. 1.13).

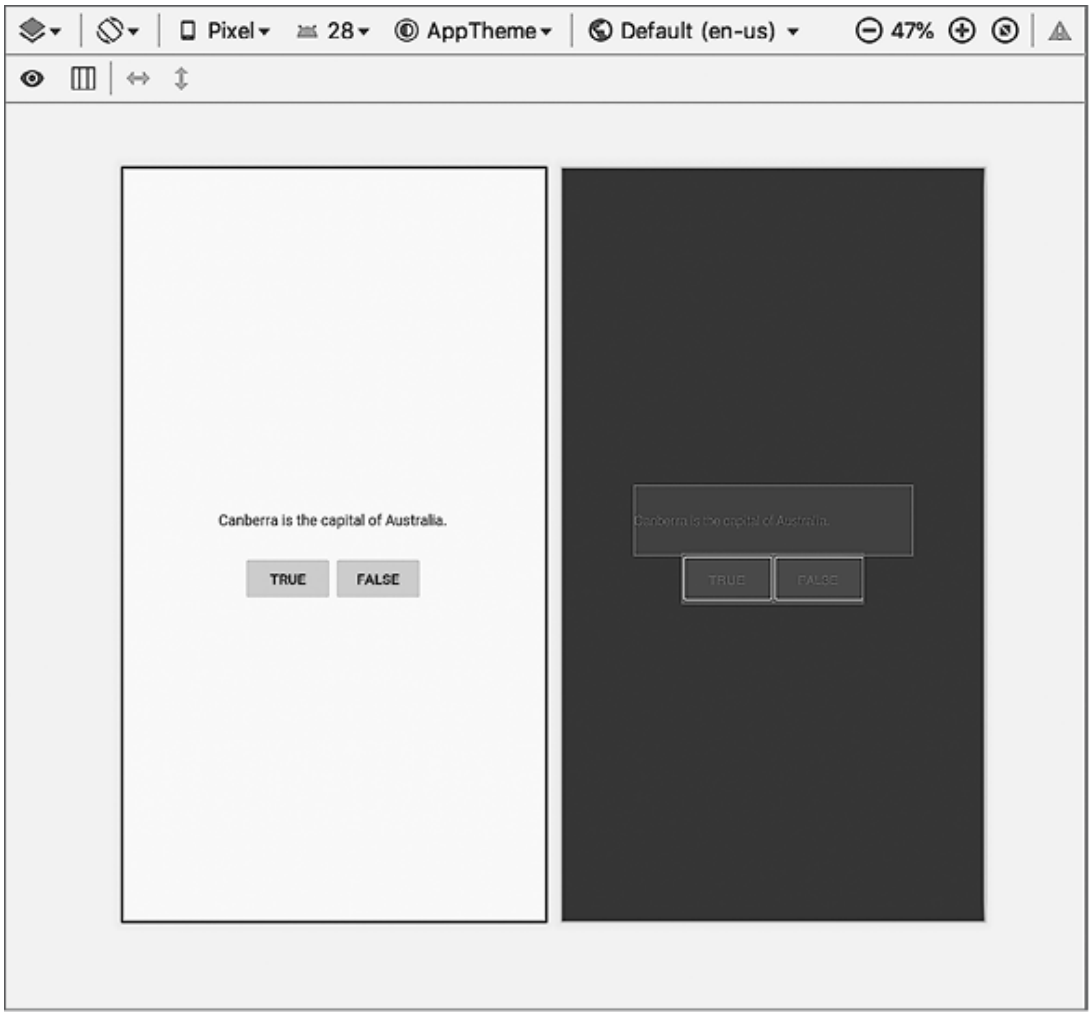


Рис. 1.13. Предварительный просмотр в графическом конструкторе макетов (activity_main.xml)

На рис. 1.13 показаны два доступных вида предварительного просмотра. Вы можете выбрать нужный вам тип предварительного просмотра с помощью меню, которое выпадает из кнопки в виде синего алмаза, расположенной на верхней панели слева. Вы можете вывести на экран любой вид предварительного просмотра по отдельности или оба сразу, как показано на рисунке.

Вид слева — это предварительный просмотр *Design*. Он показывает, как будет выглядеть макет на устройстве с применением графической темы.

Вид справа — это область раскладки (*blueprint*). В этом предварительном просмотре основное внимание уделяется размеру виджетов и взаимосвязям между ними.

Панель **Design** также позволяет увидеть, как будет выглядеть макет на устройствах с различными конфигурациями. В верхней части панели можно указать тип устройства, версию Android, тему устройства и язык, используемый при рендеринге макета. Вы даже можете сделать так, чтобы текст выводился справа налево.

Помимо предварительного просмотра вы также можете создавать свои макеты с помощью редактора макетов. Слева находится панель, на которой есть все встроенные виджеты (рис. 1.14). Вы можете перетаскивать эти виджеты из панели на макет. Вы также можете перетащить их в дерево компонентов в нижнем левом углу, чтобы более точно задать расположение виджета.

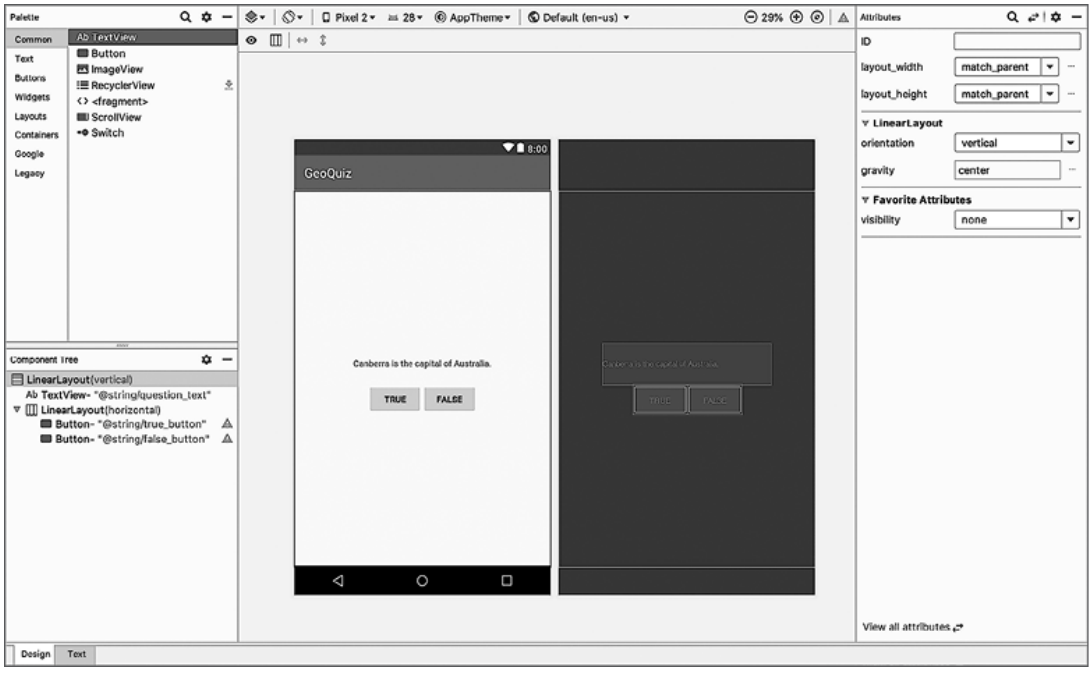


Рис. 1.14. Графический редактор макетов

На рис. 1.14 показан вид с *декорациями макета* — панель состояния устройства, панель приложений с меткой GeoQuiz и панель кнопок виртуального устройства. Чтобы увидеть эти декорации, нажмите кнопку в виде глаза на панели непосредственно над окном предварительного просмотра и выберите опцию **ShowLayoutDecorations**.

Вы найдете этот графический редактор особенно ценным при работе с `ConstraintLayout`, как сами увидите в главе 10.

От разметки XML к объектам View

Как элементы XML в файле `activity_main.xml` превращаются в объекты `View`? Ответ на этот вопрос начинается с класса `MainActivity`.

При создании проекта GeoQuiz был автоматически создан подкласс `Activity` с именем `MainActivity`. Файл класса `MainActivity` находится в каталоге `app/java` (в котором хранится Java-код вашего проекта).

Немного отвлечемся и обсудим имена каталогов, прежде чем мы перейдем к тому, как макеты превращаются в представления. Этот каталог называется `java`, потому что Android изначально поддерживал только код на Java. В вашем проекте, настроенном на работу с Kotlin (а Kotlin полностью совместим с Java), в папке `java` будет лежать код Kotlin. Вы можете создать каталог `kotlin` и разместить там свои файлы Kotlin, но тогда придется сообщить Android, что код лежит уже в другой папке. В большинстве случаев разделение файлов кода по языку не приносит никакой пользы, поэтому в большинстве проектов файлы Kotlin лежат в каталоге `java`.

Файл `MainActivity.kt` уже можно открыть в окне редактора. Если это не так, найдите каталог `app/java` на

панели **Project** и щелкните по нему, чтобы открыть его содержимое, затем щелкните, чтобы открыть содержимое пакета `com.bignerdranch.android.geoquiz` (не тот, который подкрашен зеленым, — это тестовые пакеты. Продакт-пакет не закрашен). Откройте файл `MainActivity.kt` и посмотрите его содержимое.

Листинг 1.4. Файл класса MainActivity по умолчанию (MainActivity.kt)

```
package com.bignerdranch.android.geoquiz

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

(Интересуетесь, что такое `AppCompatActivity`? Это подкласс, наследующий от класса Android `Activity` и обеспечивающий поддержку старых версий Android. Более подробная информация об `AppCompatActivity` приведена в главе 14.)

Если вы не видите все директивы `import`, щелкните по значку + слева от первой директивы `import`, чтобы раскрыть список.

Файл содержит одну функцию Activity: onCreate(Bundle?).

Функция onCreate(Bundle?) вызывается при создании экземпляра подкласса activity. Такому классу нужен пользовательский интерфейс, которым он будет управлять. Чтобы предоставить activity ее пользовательский интерфейс, следует вызвать функцию Activity.setContentView(layoutResID:Int).

Эта функция *заполняет* макет и выводит его на экран. При заполнении макета создаются экземпляры всех виджетов в файле макета с параметрами, определяемыми его атрибутами. Чтобы указать, какой именно макет следует заполнить, вы передаете *идентификатор ресурса* макета.

Ресурсы и идентификаторы ресурсов

Макет представляет собой *ресурс*. Ресурсом называется часть приложения, которая не является кодом, — графические файлы, аудиофайлы, файлы XML и т.д.

Ресурсы проекта находятся в подкаталоге каталога app/res. На панели **Project** видно, что файл activity_main.xml находится в каталоге res/layout/. Строковый файл, содержащий строковые ресурсы, находится в res/values/.

Для обращения к ресурсу в коде используется его идентификатор ресурса. Нашему макету назначен идентификатор ресурса R.layout.activity_main.

Чтобы посмотреть текущие идентификаторы ресурсов GeoQuiz, вам придется набраться храбрости и окунуться в мир автогенерируемого кода — кода, который пишется сборщиком Android для вас. Сначала запустите инструмент сборки, нажав на зеленый молоточек на панели инструментов в верхней части окна Android Studio.

По умолчанию Android Studio отображает проект в режиме **Android** на панели **Project**. В этом случае скрывается истинная структура каталогов вашего Android-проекта, и вы можете сосредоточиться на файлах и папках, которые вам нужны чаще всего. Чтобы увидеть файлы и папки в вашем проекте такими, какие они есть на самом деле, найдите раскрывающийся список в верхней части панели и выберите в нем пункт **Project** вместо **Android** (рис. 1.15).

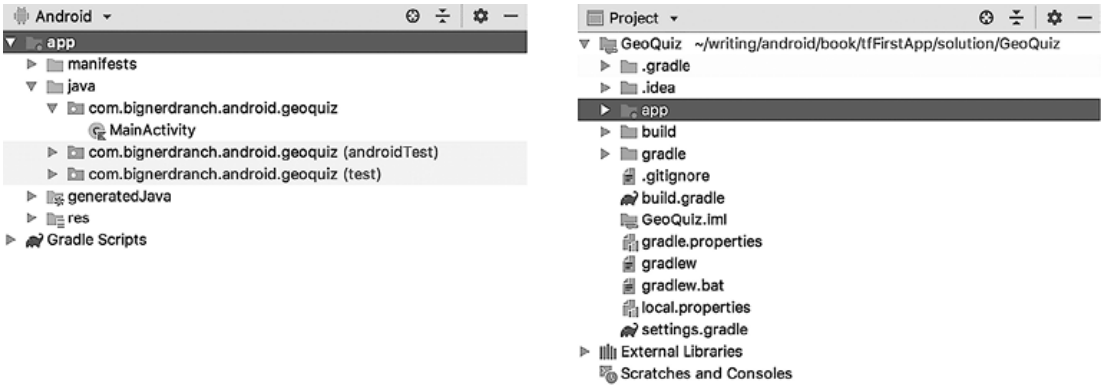


Рис. 1.15. Режимы Project и Android на панели Project

На панели **Project** раскройте каталог `GeoQuiz` и содержимое каталога `GeoQuiz/app/build/generated/not_namespaced_r_class_sources/debug/processDebugResources/r/`. В этом каталоге найдите имя пакета своего проекта и откройте файл `R.java` из этого пакета (рис. 1.16).

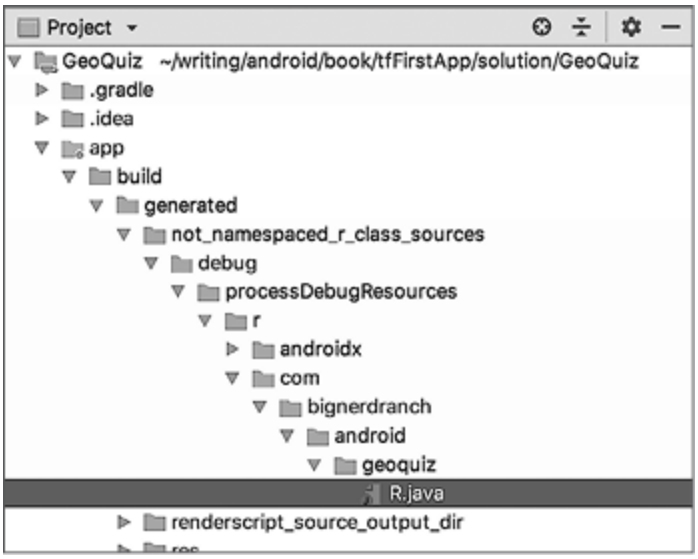


Рис. 1.16. Файл R.java

Дважды щелкните по файлу, чтобы открыть его. Поскольку файл `R.java` генерируется в процессе сборки, вы не должны изменять его, о чем написано в верхней части файла (листинг 1.5).

Листинг 1.5. Текущие идентификаторы ресурсов GeoQuiz (R.java)

```
/* AUTO-GENERATED FILE. DO NOT MODIFY.
*
* This class was automatically generated by the
* aapt tool from the resource data it found. It
* should not be modified by hand.
*/

package com.bignerdranch.android.geoquiz;

public final class R {
    public static final class anim {
```

```
        ...
    }
    ...
    public static final class id {
        ...
    }
    public static final class layout {
        ...
                    public static final Int
activity_main=0x7f030017;
    }
    public static final class mipmap {
                    public static final Int
ic_launcher=0x7f030000;
    }
    public static final class string {
        ...
                    public static final Int
app_name=0x7f0a0010;
                    public static final Int
false_button=0x7f0a0012;
                    public static final Int
question_text=0x7f0a0014;
                    public static final Int
true_button=0x7f0a0015;
    }
}
```

Кстати, вы можете и не увидеть этот файл сразу после внесения изменений в ваши ресурсы. Android Studio ведет скрытый файл `R.java`, на основе которого строится ваш код. `R.java`-файл в листинге 1.5 — это файл, который генерируется

для вашего приложения непосредственно перед его установкой на устройство или эмулятор. Вы увидите обновление этого файла при запуске приложения.

Файл `R.java` может быть довольно большим; в листинге 1.5 значительная часть его содержимого не показана.

Теперь понятно, откуда взялось имя `R.layout.activity_main` — это целочисленная константа `activity_main` из внутреннего класса `layout` класса `R`.

Строкам также назначаются идентификаторы ресурсов. Мы еще не ссылались на строки в коде, но эти ссылки обычно выглядят так:

```
setTitle(R.string.app_name);
```

Android генерирует идентификатор ресурса для всего макета и для каждой строки, но не для отдельных виджетов из файла `activity_main.xml`. Не каждому виджету нужен идентификатор ресурса. В этой главе мы будем взаимодействовать лишь с двумя кнопками, поэтому идентификаторы ресурсов нужны только им.

Чтобы сгенерировать идентификатор ресурса для виджета, включите в определение виджета атрибут `android:id`. В файле `activity_main.xml` добавьте атрибут `android:id` для каждой кнопки.

Листинг 1.6. Добавление идентификаторов к объектам Button (res/layout/activity_main.xml)

```
<LinearLayout ... >

    <TextView
        android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:padding="24dp"
            android:text="@string/question_text" />

        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:orientation="horizontal">

            <Button
                android:id="@+id/true_button"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content
    "
                 android:text="@string/true_button"
    />

            <Button
                android:id="@+id/false_button"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content
    "
                android:text="@string/false_button"
    />

        </LinearLayout>

    </LinearLayout>
```

Обратите внимание: знак + присутствует в значениях `android:id`, но не в значениях `android:text`. Это связано с

тем, что мы создаем идентификаторы, а на строки только *ссылаемся*.

Перед тем как двигаться дальше, переключите панель **Project** с режима **Project** на **Android**. На протяжении всей книги будет использоваться режим **Android** — но вы можете использовать **Project**, если хотите. Также закройте файл `R.java`, щелкнув по крестику в соответствующей вкладке.

Разработка виджетов

Вы готовы приступить к созданию собственных виджетов. Этот процесс разделен на два этапа:

- получение ссылок на заполненные объекты `View`;
- назначение для этих объектов слушателей, реагирующих на действия пользователя.

Установка ссылок на виджеты

Теперь, когда кнопкам назначены идентификаторы ресурсов, к ним можно обращаться в `MainActivity`. Введите следующий код в `MainActivity.kt` (листинг 1.7). (Не используйте автозавершение — введите его самостоятельно.) После сохранения файла выводятся два сообщения об ошибках, скоро вы с ними справитесь.

Листинг 1.7. Добавление полей (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    private lateinit var trueButton: Button
    private lateinit var falseButton: Button
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

                                    trueButton    =
findViewById(R.id.true_button)
                                   falseButton    =
findViewById(R.id.false_button)
}
}
```

В activity можно получить ссылку на виджет, вызвав функцию `Activity.findViewById(Int)`. Эта функция возвращает соответствующую панель. Вместо того чтобы возвращать ее как `View`, она приводится к ожидаемому подтипу `View`. Здесь этот тип — `Button`.

В приведенном выше коде вы используете идентификаторы ресурсов ваших кнопок, чтобы получить объекты и присвоить их свойствам представлений. Так как объекты не появляются в памяти до тех пор, пока в `onCreate(...)` не будет вызвана функция `setContentView(...)`, вы используете `lateinit` в объявлениях свойств, чтобы указать компилятору, что вы введете ненулевое значение `View` перед попыткой использовать содержимое свойства. Затем в функции `onCreate(...)` вы ищете и назначаете объектам представления соответствующие свойства. Подробнее о функции `onCreate(...)` и жизненном цикле activity вы узнаете в главе 3.

Сейчас мы исправим ошибки. Наведите указатель мыши на красные индикаторы ошибок. Ошибка следующая:

`Unresolvedreference:Button`.

Чтобы избавиться от ошибок, следует импортировать класс `android.widget.Button` в `MainActivity.kt`. Введите следующую директиву импортирования в начале файла:

```
import android.widget.Button;
```

А можно пойти по простому пути и поручить эту работу Android Studio. Нажмите сочетание клавиш □+□ (или **Alt+Enter**), и к вам на помощь придет магия IntelliJ. Новая директива `import` сама появится под другими директивами в начале файла. Этот прием часто бывает полезным, если ваш код работает не так, как положено. Экспериментируйте с ним почаще!

Ошибки должны исчезнуть. (Если они остались, поищите опечатки в коде и XML.) Когда с ошибками покончено, настало время сделать приложение интерактивным.

Назначение слушателей

Android-приложения обычно *событийно-управляемые* (*event-driven*). В отличие от консольных программ или сценариев, такие приложения запускаются и ожидают наступления некоторого события, например нажатия кнопки пользователем. (События также могут инициироваться ОС или другим приложением, но события, инициируемые пользователем, наиболее очевидны.)

Когда ваше приложение ожидает наступления конкретного события, мы говорим, что оно «прослушивает» данное событие. Объект, создаваемый для ответа на событие, называется *слушателем* (*listener*). Такой объект реализует *интерфейс слушателя* данного события.

Android SDK поставляется с интерфейсами слушателей для разных событий, поэтому вам не придется писать собственные реализации. В нашем случае прослушиваемым событием является «щелчок» на кнопке, поэтому слушатель должен реализовать интерфейс `View.OnClickListener`.

Начнем с кнопки **TRUE**. В файле `MainActivity.kt` включите следующий фрагмент кода в функцию `onCreate(Bundle?)` непосредственно после присваивания.

Листинг 1.8. Назначение слушателя для кнопки TRUE (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    trueButton = findViewById(R.id.true_button)
                            falseButton =
findViewById(R.id.false_button)

trueButton.setOnClickListener { view: View ->
// Что-то выполнить после нажатия
}
}
```

(Если вы получили ошибку `Unresolvedreference:View`, воспользуйтесь комбинацией □+□ (**Alt+Enter**) для импортирования класса `View`.)

В листинге 1.8 вы устанавливаете слушателя, который будет сообщать вам о нажатии кнопки `trueButton`. Фреймворк Android определяет `View.OnClickListener` как Java-

интерфейс с одним методом `onClick(View)`. Интерфейсы с *одним абстрактным методом* достаточно распространены в Java, и у такого шаблона есть имя SAM.

В Kotlin предусмотрена специальная поддержка этого шаблона внутри уровня Java. Это позволяет писать функции буквально, и происходит превращение в объект, реализующий интерфейс. Этот процесс называется *SAM-преобразованием*.

Ваш слушатель, ожидающий нажатия, реализуется с помощью лямбда-выражения. Назначьте аналогичного слушателя для кнопки **FALSE**.

Листинг 1.9. Назначение слушателя для кнопки FALSE (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    ...
     trueButton.setOnClickListener { view: View
->
        // Что-то выполнить после нажатия
    }
falseButton.setOnClickListener { view: View ->
// Что-то выполнить после нажатия
}
}
```

Уведомления

Пора заставить кнопки делать что-то полезное. В нашем приложении каждая кнопка будет выводить на экран *временное уведомление* (*toast*) — короткое сообщение, которое содержит какую-либо информацию для пользователя, но не требует ни

ввода, ни действий. Наши уведомления будут сообщать пользователю, правильно ли он ответил на вопрос (рис. 1.17).

Для начала вернитесь к файлу `strings.xml` и добавьте строковые ресурсы, которые будут отображаться в уведомлении.

Листинг 1.10. Добавление строк уведомлений (res/values/strings.xml)

```
<resources>
    <string name="app_name">GeoQuiz</string>
      <string name="question_text">Canberra is
the capital of Australia.</string>
    <string name="true_button">True</string>
    <string name="false_button">False</string>
        <string name="correct_toast">Correct!
</string>
      <string name="incorrect_toast">Incorrect!
</string>
</resources>
```

Рис. 1.17. Уведомление с информацией для пользователя

Теперь нужно обновить слушателей нажатий, чтобы они создавали и выводили уведомление. Используйте автозаполнение кода, чтобы быстрее написать код слушателя. Автозаполнение кода может сэкономить вам много времени, так что хорошо познакомиться с ним заблаговременно.

Начните вводить код, показанный в листинге 1.11, в `MainActivity.kt`. Возле класса `Toast` появится всплывающее окно со списком предлагаемых функций и констант из класса `Toast`.

Для выбора одного из предложений используйте клавиши со стрелками вверх и вниз. (Если вы не хотите использовать автозаполнение, можно просто продолжить набор текста. Само

по себе ничего не произойдет, если не нажать клавишу **Tab**, ⏎ (**Enter**) или не щелкнуть по всплывающему окну.)

В списке предложений выберите `makeText(context:Context,resId:Int,duration:Int)`. Автозаполнение кода добавит вам полный вызов функции.

Заполните параметры функции `makeText(...)`, получив код, приведенный в листинге 1.11.

Листинг 1.11. Создание уведомлений (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    ...
     trueButton.setOnClickListener { view: View
->
            // Что-то выполнить после нажатия
        Toast.makeText(
                this,
                R.string.correct_toast,
                Toast.LENGTH_SHORT)
                .show()
    }

    false.setOnClickListener { view: View ->
            // Что-то выполнить после нажатия
        Toast.makeText(
                this,
                R.string.incorrect_toast,
                Toast.LENGTH_SHORT)
                .show()
    }
```

```
    }
```

Для создания уведомления вызывается статическая функция `Toast.makeText(Context!,Int,Int)`. Эта функция создает и настраивает объект `Toast`. Параметр `Context` обычно является экземпляром `Activity` (а `Activity` — подклассом `Context`). Здесь в качестве аргумента `Context` передается экземпляр `MainActivity`.

Второй параметр — идентификатор ресурса строки, которую должно отобразить уведомление. Аргумент `Context` необходим классу `Toast`, чтобы он мог найти и использовать идентификатор ресурса строки. Третий параметр — одна из двух констант `Toast`, которые указывают, как долго уведомление должно отображаться.

После того как создали уведомление, вы вызываете на нем функцию `Toast.show()`, чтобы вывести его на экран.

Благодаря использованию автозаполнения вам не придется ничего специально делать для импортирования класса `Toast`. Когда вы соглашаетесь на рекомендацию автозаполнения, необходимые классы импортируются автоматически.

Посмотрим, как работает новое приложение.

Выполнение в эмуляторе

Для запуска Android-приложений необходимо устройство — физическое или виртуальное. Виртуальные устройства работают под управлением эмулятора Android, включенного в поставку средств разработчика.

Чтобы создать виртуальное устройство Android (AVD, Android Virtual Device), выполните команду **Tools⇒AVDManager**. Когда на экране появится окно **AVDManager**, щелкните мышью по кнопке **+CreateVirtualDevice...** в левой части этого окна.

Открывается диалоговое окно с многочисленными параметрами настройки виртуального устройства. Для первого AVD выберите эмуляцию устройства Pixel 2, как показано на рис. 1.18. Нажмите кнопку **Next**.

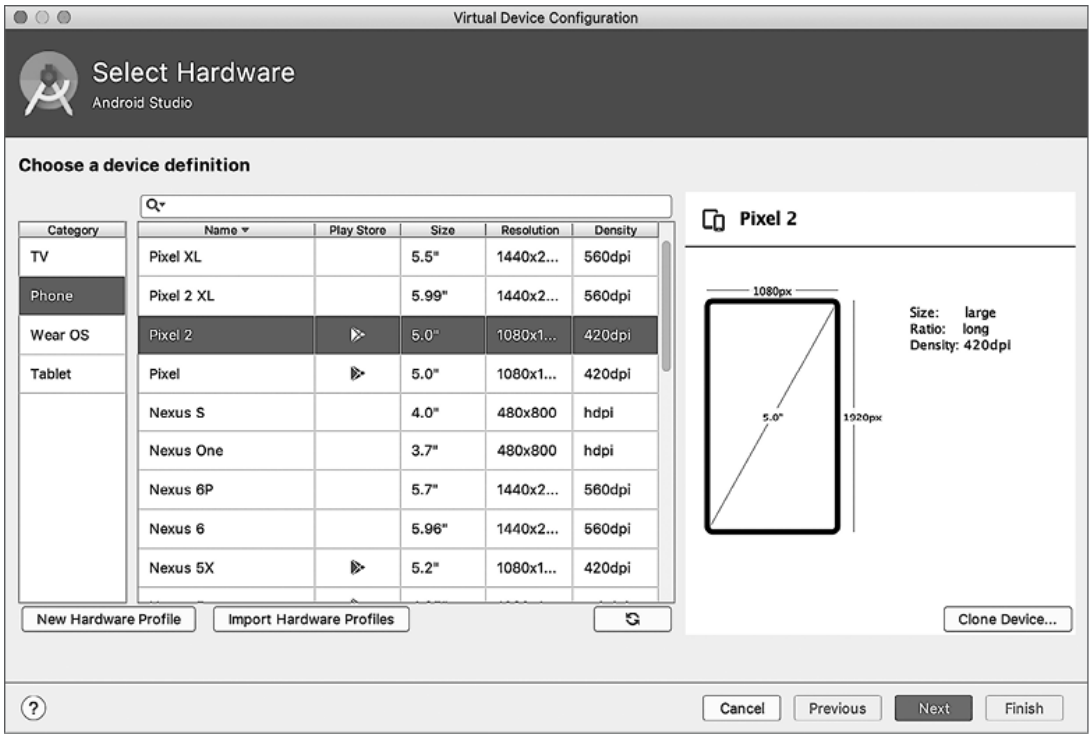


Рис. 1.18. Выбор виртуального устройства

На следующем экране выберите образ системы, на основе которого будет работать эмулятор. Выберите эмулятор x86 Pie и нажмите кнопку **Next** (рис. 1.19). (Возможно, перед этим вам придется выполнить необходимые действия для загрузки компонентов эмулятора.)

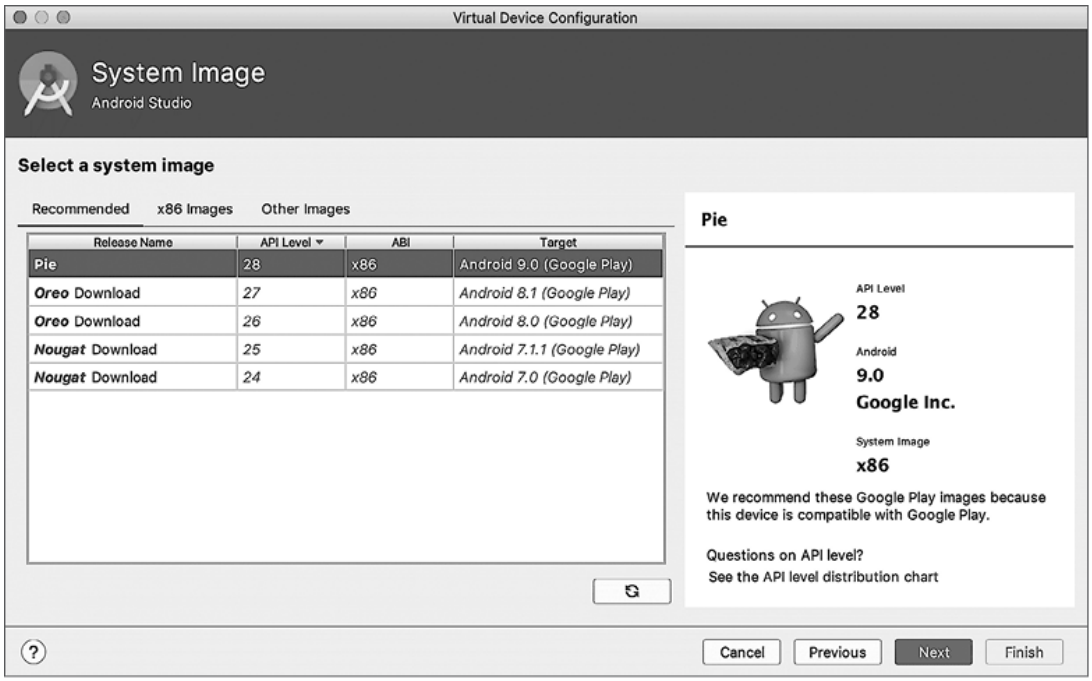


Рис. 1.19. Выбор образа системы

Наконец, просмотрите и при необходимости измените свойства эмулятора (впрочем, свойства существующего эмулятора также можно изменить позднее). Пока присвойте своему эмулятору имя, по которому вы сможете узнать его в будущем, и нажмите кнопку **Finish** (рис. 1.20).

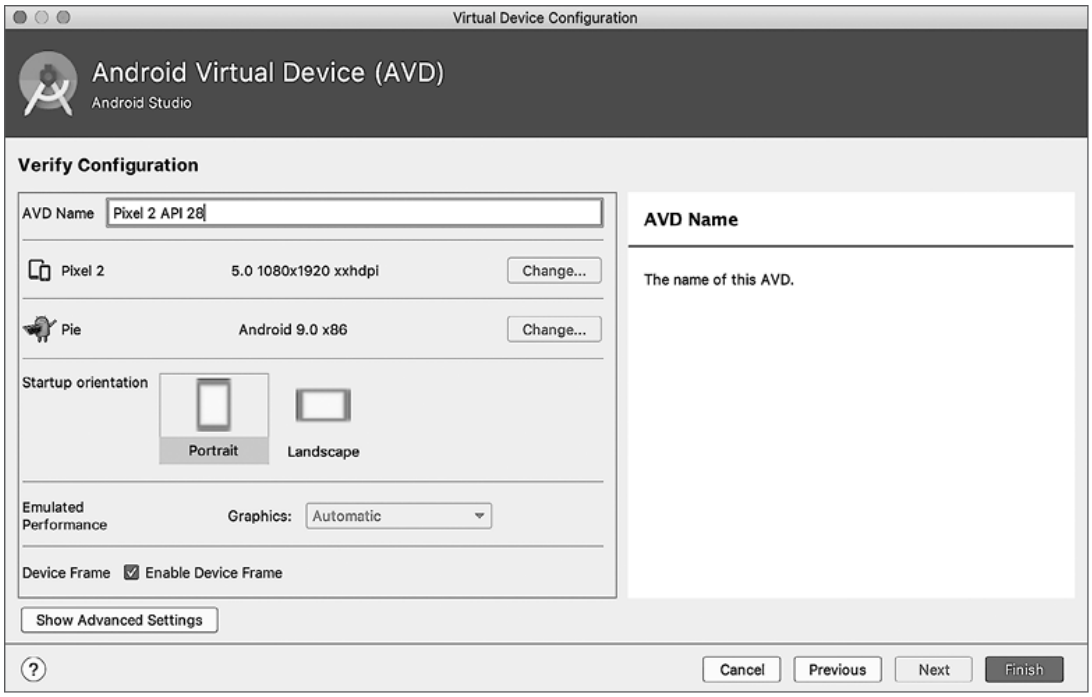


Рис. 1.20. Настройка свойств эмулятора

Когда виртуальное устройство будет создано, в нем можно запустить приложение GeoQuiz. На панели инструментов Android Studio нажмите кнопку **Run** (зеленый символ «воспроизведение») или нажмите сочетание клавиш **Ctrl+R**. Android Studio находит созданное виртуальное устройство, запускает его, устанавливает на нем пакет приложения и запускает приложение.

Возможно, на запуск эмулятора потребуется некоторое время, но вскоре приложение GeoQuiz запустится на созданном вами виртуальном устройстве. Понажимайте кнопки и оцените уведомления.

Если при запуске GeoQuiz или нажатии кнопки происходит сбой, в нижней части панели **LogCat** окна Android Studio выводится полезная информация. (Если панель **LogCat** не открылась автоматически при запуске GeoQuiz, приложение можно открыть при помощи кнопки Android Monitor в нижней части окна Android Studio.) Введите `MainActivity` в поле

поиска в верхней части панели **Logcat**, чтобы отфильтровать сообщения журнала. Ищите исключения в журнале; они выделены красным цветом (рис. 1.21).

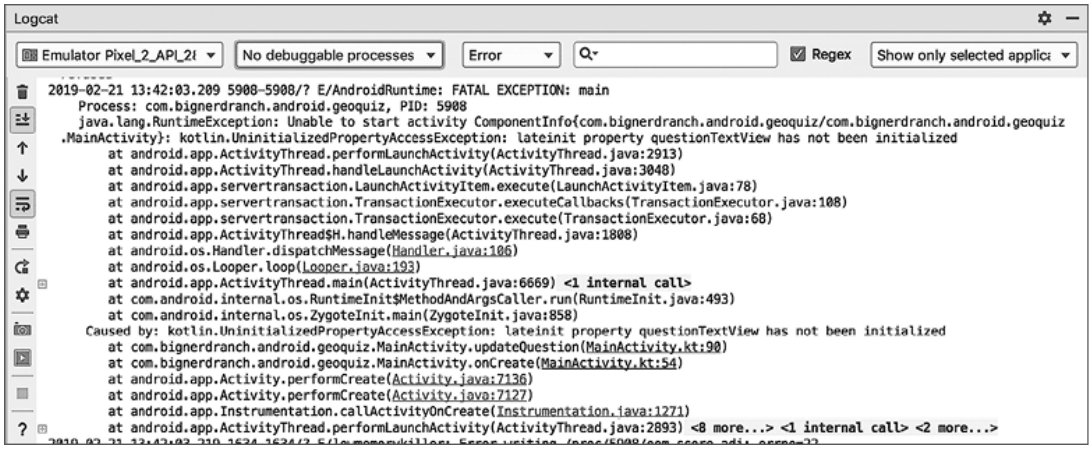


Рис. 1.21. Исключение NullPointerException

Сравните свой код с кодом в книге и попытайтесь найти источник проблемы. Затем снова запустите приложение. (**LogCat** и процесс отладки более подробно рассматриваются в следующих двух главах.)

Не закрывайте эмулятор; не стоит ждать, пока он загружается при каждом запуске.

Вы можете остановить приложение кнопкой «Назад» (треугольник, обращенный влево; в старых версиях Android использовалось изображение U-образной стрелки), а затем снова запустить приложение из Android Studio, чтобы протестировать изменения.

Эмулятор полезен, но тестирование на реальном устройстве дает более точные результаты. В главе 2 мы запустим приложение GeoQuiz на физическом устройстве, а также расширим набор вопросов по географии.

Для любознательных: процесс сборки Android-приложений

Вероятно, у вас уже накопились вопросы относительно того, как организован процесс сборки Android-приложений. Вы уже видели, что Android Studio строит проект автоматически в процессе его изменения, а не по команде. Во время сборки инструментарий Android берет ваши ресурсы, код и файл `AndroidManifest.xml` (содержащий метаданные приложения) и преобразует их в файл `.apk`. Полученный файл подписывается отладочным ключом, что позволяет запускать его в эмуляторе. (Чтобы распространять файл `.apk` среди пользователей, необходимо подписать его ключом публикации. Дополнительную информацию об этом процессе можно найти в документации разработчика Android по адресу **developer.android.com/tools/publishing/preparing.html**.)

Как содержимое `activity_main.xml` преобразуется в объекты `View` в приложении? В процессе сборки утилита `aapt2` (Android Asset Packaging Tool) компилирует ресурсы файлов макетов в более компактный формат. Откомпилированные ресурсы упаковываются в файл `.apk`. Затем, когда функция `setContentView(...)` вызывается в функции `onCreate(Bundle?)` класса `MainActivity`, `MainActivity` использует класс `LayoutInflater` для создания экземпляров всех объектов `View`, определенных в файле макета (рис. 1.22).

(Также классы представлений можно создать на программном уровне в activity вместо определения их в XML. Однако отделение презентационной логики от логики приложения имеет свои преимущества, главное из которых — использование изменений конфигурации, встроенных в SDK; мы подробнее поговорим об этом в главе 3.)

За дополнительной информацией о том, как работают различные атрибуты XML и как происходит отображение представлений на экране, обращайтесь к главе 10.

Инструменты сборки

Все программы, которые мы запускали до настоящего времени, исполнялись из среды Android Studio. Этот процесс интегрирован в среду разработки — он вызывает стандартные средства сборки Androi-приложений (такие как `aapt2`), но сам процесс сборки проходит под управлением Android Studio.

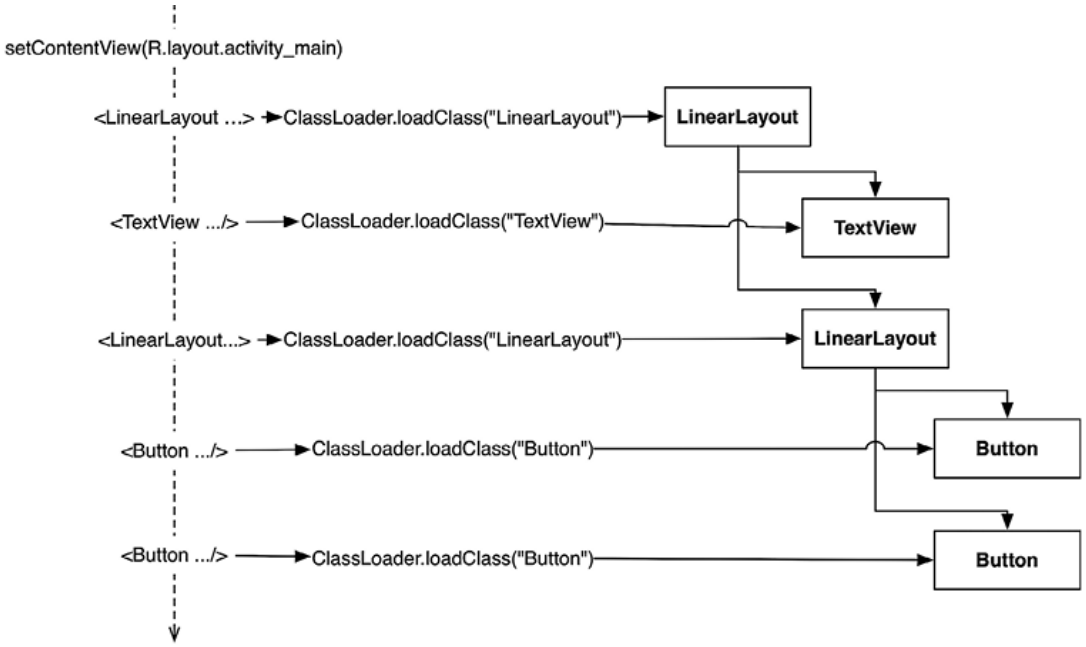


Рис. 1.22. Заполнение activity_main.xml

Может оказаться, что вам по каким-то своим причинам потребуется собрать приложение за пределами среды Android Studio. Для этого проще всего воспользоваться консольной программой сборки. В настоящее время для сборки Android-приложений используется программа Gradle.

(Вы и сами поймете, представляет ли эта информация для вас интерес. Если нет, просто бегло просмотрите ее, не беспокоясь о том, зачем это нужно и что делать, если что-то не работает. Рассмотрение всех тонкостей использования командной строки выходит за рамки книги.)

Чтобы запустить программу Gradle из командной строки, перейдите в каталог своего проекта и выполните следующую команду:

```
$ ./gradlew tasks
```

В системе Windows команда выглядит немного иначе:

```
> gradlew.bat tasks
```

Команда выводит список задач, которые можно выполнить. Та задача, которая вам нужна, называется `installDebug`. Запустите ее следующей командой:

```
$ ./gradlew installDebug
```

Или в системе Windows:

```
> gradlew.bat installDebug
```

Команда устанавливает приложение на подключенное устройство, но не запускает его — вы должны сделать это вручную.

Упражнения

Упражнения в конце главы предназначены для самостоятельной работы. Некоторые из них просты и рассчитаны на повторение материала главы. Другие, более сложные, упражнения потребуют логического мышления и навыков решения задач.

Трудно переоценить важность упражнений. Они закрепляют усвоенный материал, повышают уверенность в обретенных навыках и сокращают разрыв между изучением

программирования Android и умением самостоятельно писать программы для Android.

Если у вас возникнут затруднения с каким-либо упражнением, сделайте перерыв, потом вернитесь и попробуйте еще раз. Если и эта попытка окажется безуспешной, обратитесь на форум книги **forums.bignerdranch.com**. Здесь вы сможете просмотреть вопросы и решения, полученные от других читателей, а также опубликовать свои вопросы и решения.

Чтобы избежать случайного повреждения текущего проекта, мы рекомендуем создать копию и практиковаться на ней.

В файловом менеджере компьютера перейдите в корневой каталог своего проекта. Скопируйте папку `GeoQuiz` и вставьте новую копию рядом с оригиналом (в macOS воспользуйтесь функцией дублирования). Переименуйте новую папку и присвойте ей имя `GeoQuizChallenge`. В Android Studio выполните команду **File⇒ImportProject...**. В окне импорта перейдите к папке `GeoQuizChallenge` и нажмите кнопку **OK**. Скопированный проект появится в новом окне готовым к работе.

Упражнение. Настройка уведомления

Измените уведомление так, чтобы оно отображалось в верхней, а не в нижней части экрана. Для изменения способа отображения уведомления воспользуйтесь функцией `setGravity` класса `Toast`. Выберите режим Gravity.TOP. За дополнительной информацией обращайтесь к документации разработчика по адресу **developer.android.com/reference/kotlin/android/widget/Toast#setgravity**.

2. Android и модель MVC

В этой главе мы обновим приложение GeoQuiz и включим в него дополнительные вопросы (рис. 2.1).

Рис. 2.1. Больше вопросов!

Для этого в проект GeoQuiz будет добавлен класс `Question`. Экземпляр этого класса инкапсулирует один вопрос с ответом «да/нет».

Затем мы создадим группу объектов `Question`, с которым будет работать класс `MainActivity`.

Создание нового класса

На панели **Project** щелкните правой кнопкой мыши по пакету `com.bignerdranch.android.geoquiz` и выберите команду **New⇒KotlinFile/Class** в контекстном меню. Введите имя класса `Question`, выберите пункт **Class** в раскрывающемся списке **Kind** и нажмите кнопку **OK** (рис. 2.2).

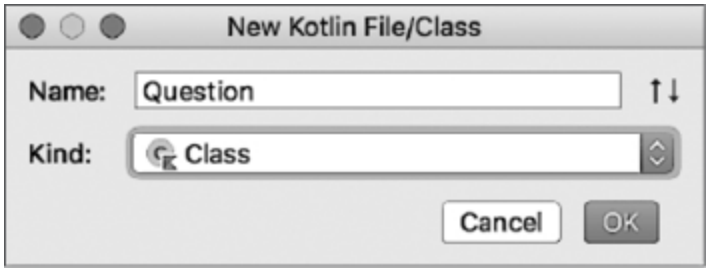


Рис. 2.2. Создание класса Question

Android Studio создаст файл `Question.kt`. Добавьте в этот файл две переменные и конструктор.

Листинг 2.1. Добавление класса Question (Question.kt)

```
class Question {
}
data class Question(@StringRes val textResId:
Int, val answer: Boolean)
```

Класс `Question` содержит два вида данных: текст вопроса и правильный ответ (да/нет).

Аннотация `@StringRes` не обязательна, но мы рекомендуем включить ее по двум причинам. Во-первых, аннотация помогает встроенному в Android Studio инспектору кода (под названием Lint) проверить во время компиляции, что в конструкторе используется правильный строковый идентификатор ресурса. Это предотвращает сбои во время

выполнения, когда конструктор используется с недействительным идентификатором ресурса (например, если идентификатор указывает не на строку). Во-вторых, аннотация делает ваш код более читаемым для других разработчиков.

Почему `textResId` — это `Int`, а не `String`? Потому что переменная `textResId` содержит идентификатор (всегда `Int`) строкового ресурса.

Мы используем ключевое слово `data` для всех классов моделей в этой книге. Тем самым мы намекаем на то, что класс предназначен для хранения данных. Кроме того, с такими классами компилятор работает дополнительно, например, автоматически определяет такие полезные функции, как `equals()`, `hashCode()`, и вводит красивое форматирование через `toString()`.

Класс `Question` готов. Через некоторое время мы изменим `MainActivity` для работы с функцией `Question`. Но сначала давайте посмотрим, как будут вместе работать кусочки GeoQuiz.

Класс `MainActivity` должен создать список объектов `Question`. В процессе работы он взаимодействует с `TextView` и тремя виджетами `Button` для вывода вопросов и предоставления обратной связи на ответы пользователя. Отношения между этими объектами изображены на рис. 2.3.

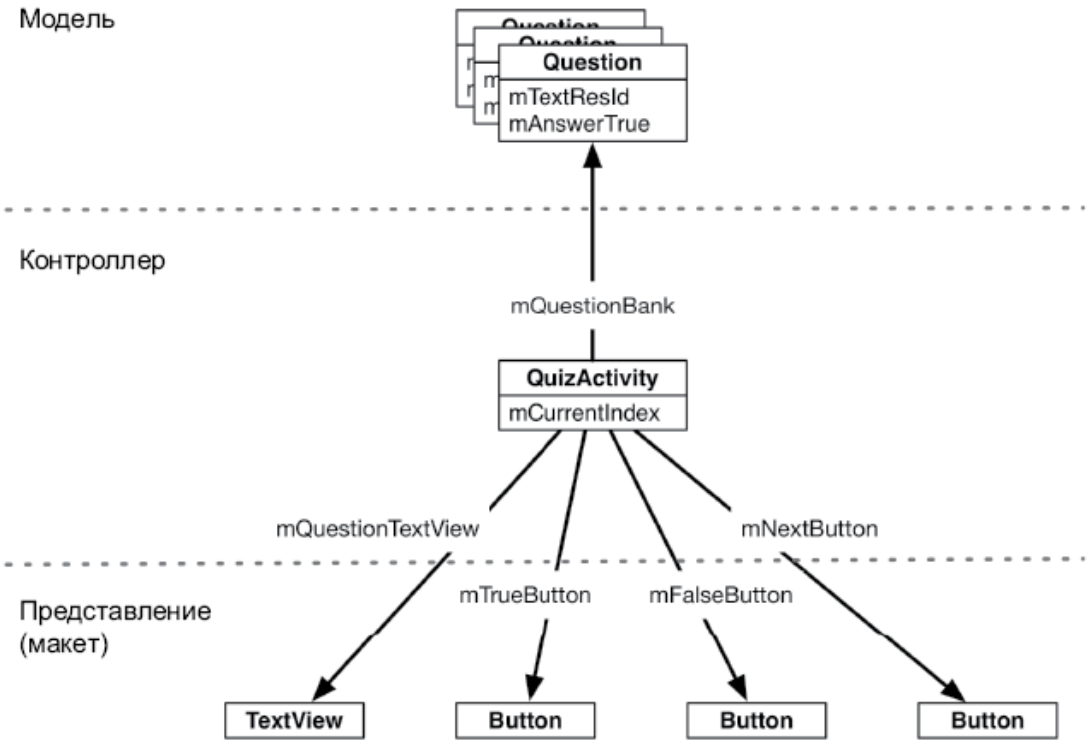


Рис. 2.3. Диаграмма объектов GeoQuiz

Архитектура «Модель — Представление — Контроллер» и Android

Вероятно, вы заметили, что объекты на рис. 2.3 разделены на три области: «Модель», «Контроллер» и «Представление». Android-приложения строятся на базе архитектуры, называемой «*Модель — Представление — Контроллер*», или сокращенно *MVC* (*Model-View-Controller*). Согласно канонам MVC, каждый объект приложения должен быть *объектом модели*, *объектом представления* или *объектом контроллера*.

- Объект модели содержит данные приложения и «бизнес-логику». Классы модели обычно проектируются для *моделирования* сущностей, с которыми имеет дело приложение: пользователь, продукт в магазине, фотография на сервере, вопрос «да/нет» и т.д. Объекты модели ничего не

знают о пользовательском интерфейсе; их единственная цель — хранение и управление данными.

В приложениях Android классы моделей обычно создаются разработчиком для конкретной задачи. Все объекты модели в вашем приложении составляют его *уровень модели*.

Уровень модели GeoQuiz состоит из класса `Question`.

- Объекты представлений умеют отображать себя на экране и реагировать на ввод пользователя, например касания. Простейшее правило: если вы видите что-то на экране, значит, это представление.

  Android дает широкий набор настраиваемых классов представлений. Разработчик также может создавать собственные классы представлений. Объекты представления в приложении образуют *уровень представления*.

  В GeoQuiz уровень представления состоит из виджетов, заполненных по содержимому файла `res/layout/activity_main.xml`.

- Объекты контроллеров связывают объекты представления и модели; они содержат «логику приложения». Контроллеры реагируют на различные события, инициируемые объектами представлений, и управляют потоками данных между объектами модели и уровнем представления.

  В Android контроллер обычно представляется подклассом `Activity` или `Fragment`. (Фрагменты рассматриваются в главе 8.)

Уровень контроллера GeoQuiz в настоящее время состоит только из класса `MainActivity`.

На рис. 2.4 показана логика передачи управления между объектами в ответ на пользовательское событие, такое как нажатие кнопки. Обратите внимание: объекты модели и представления не взаимодействуют друг с другом напрямую; в любом взаимодействии участвуют «посредники» — контроллеры, получающие сообщения от одних объектов и передающие инструкции другим.

Преимущества MVC

Приложение может обрастать функциональностью до тех пор, пока не станет слишком сложным для понимания. Разделение кода на классы упрощает проектирование и понимание приложения в целом; разработчик мыслит в контексте классов, а не отдельных переменных и функций.

Рис. 2.4. Взаимодействия MVC при получении ввода от пользователя

Аналогичным образом разделение классов на уровни модели, представления и контроллера упрощает

проектирование и понимание приложения; вы можете мыслить в контексте уровней, а не отдельных классов.

Приложение GeoQuiz нельзя назвать сложным, но преимущества разделения уровней видны и здесь. Вскоре мы обновим уровень представления GeoQuiz и добавим в него кнопку **NEXT**. При этом нам совершенно не нужно помнить о только что созданном классе `Question`.

MVC также упрощает повторное использование классов. Класс с ограниченными обязанностями лучше подходит для повторного использования, чем класс, который пытается заниматься всем сразу.

Например, класс модели `Question` ничего не знает о виджетах, используемых для вывода вопроса «да/нет». Это упрощает использование `Question` в приложении для разных целей. Например, если потребуется вывести полный список всех вопросов, вы можете использовать тот же объект, который используется для вывода всего одного вопроса.

Принцип MVC отлично подойдет для маленьких и простых приложений вроде GeoQuiz. Но в больших и сложных приложениях уровень контроллера может при таком подходе получиться очень большим и сложным. Нам же, напротив, хочется, чтобы в activity и в контроллере было поменьше кода. В таких «тонких» activity должно быть как можно меньше настроек и действий. Как только MVC перестает подходить для тонких контроллеров в вашем приложении, стоит подумать об альтернативных вариантах, таких как модель «Модель — Представление — Представление» (о которой вы узнаете в главе 19).

Обновление уровня представления

После теоретического знакомства с MVC пора переходить к практике: обновим уровень представления GeoQuiz и включим в него кнопку **NEXT**.

В Android объекты уровня представления обычно заполняются на основе разметки XML в файле макета. Весь макет GeoQuiz определяется в файле `activity_main.xml`. В него следует внести изменения, представленные на рис. 2.5. (Для экономии места на рисунке не показаны атрибуты виджетов, оставшихся без изменений.)

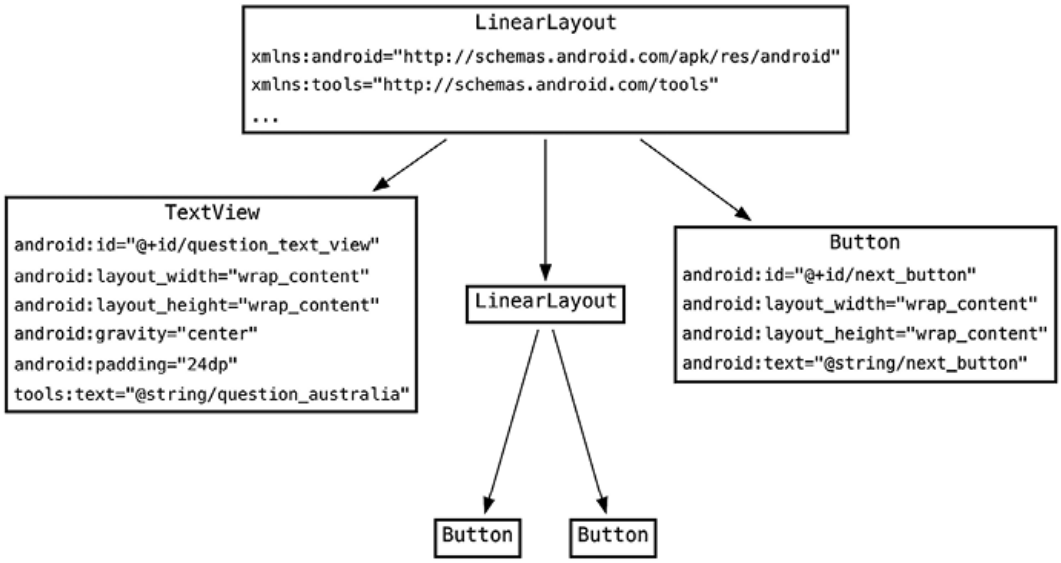


Рис. 2.5. Новая кнопка!

Итак, на уровне представления необходимо внести следующие изменения:

- Назначьте виджету `TextView` атрибут `android:id`. Этому виджету понадобится идентификатор ресурса, чтобы вы могли задать для него текст в коде `MainActivity`. Расположите текст `TextView` в центре текстового поля, присвоив свойству `gravity` значение `center`.

- Удалите атрибут `android:text` виджета `TextView`. Теперь не нужно, чтобы строго определенный вопрос был частью определения. Дело в том, что вы будете задавать текст вопроса динамически, по мере того как пользователь будет щелкать по вопросам.

- Укажите строку по умолчанию для вкладки `Design`, которая будет отображаться в `TextView` в предварительном просмотре макета. Для этого добавьте атрибут `tools:text` виджету `TextView` и задайте ему ссылку на строковый ресурс с вопросом, используя `@string/`.

  Также необходимо добавить пространство имен `tools` в корневой тег макета, чтобы программа Android Studio смогла понять атрибут `tools:text`. Это пространство имен позволяет переопределить любой атрибут виджета для предварительного просмотра в Android Studio. Атрибуты инструментов игнорируются при отрисовке виджетов на устройстве во время выполнения. Вы можете использовать `android:text` и просто перезаписать значение во время исполнения, но использование `tools:text` говорит о том, что предоставляемое вами значение предназначено только для предварительного просмотра.

- Добавьте новый виджет `Button` в качестве дочернего элемента корневого элемента `LinearLayout`.

Вернитесь к файлу `activity_main.xml` и выполните все перечисленные действия.

Листинг 2.2. Новая кнопка и изменения в TextView (res/layout/activity_main.xml)

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        ... >

    <TextView
        android:id="@+id/question_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:padding="24dp"
            android:text="@string/question_text"
        tools:text="@string/question_australia" />

    <LinearLayout ... >
        ...
    </LinearLayout>

    <Button
        android:id="@+id/next_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/next_button" />
```

```
</LinearLayout>
```

На экране появится знакомая ошибка с сообщением об отсутствующем строковом ресурсе.

Вернитесь к файлу `res/values/strings.xml`. Переименуйте `question_text` и добавьте строку для новой кнопки.

Листинг 2.3. Обновленные строки (res/values/strings.xml)

```
<string name="app_name">GeoQuiz</string>
    <string name="question_text">Canberra is the capital of Australia.</string>
<string name="question_australia">Canberra is the capital of Australia.</string>
<string name="true_button">True</string>
<string name="false_button">False</string>
<string name="next_button">Next</string>
...
```

Пока файл `strings.xml` открыт, добавьте строки с вопросами по географии, которые будут предлагаться пользователю.

Листинг 2.4. Добавление строк вопросов (res/values/strings.xml)

```
<string name="question_australia">Canberra is the capital of Australia.</string>
<string name="question_oceans">The Pacific Ocean is larger than the Atlantic Ocean.</string>
```

```
<string name="question_mideast">The Suez Canal connects the Red Sea and the Indian Ocean.</string>
<string name="question_africa">The source of the Nile River is in Egypt.</string>
<string name="question_americas">The Amazon River is the longest river in the Americas.</string>
<string name="question_asia">Lake Baikal is the world\'s oldest and deepest freshwater lake.</string>
...
```

Обратите внимание на использование служебной последовательности `\'` для включения апострофа в последнюю строку. В строковых ресурсах могут использоваться все стандартные служебные последовательности, включая `\n` для обозначения новой строки.

Вернитесь к файлу `activity_main.xml` и ознакомьтесь с изменениями макета в графическом конструкторе.

Пока это все, что относится к уровню представления GeoQuiz. Пора связать все воедино в классе контроллера `MainActivity`.

Обновление уровня контроллера

В предыдущей главе в единственном контроллере GeoQuiz — `MainActivity` — не происходило почти ничего. Он отображал макет, определенный в файле `activity_main.xml`, назначал слушателей для двух кнопок и организовывал выдачу уведомлений.

Теперь, когда у нас появились дополнительные вопросы, классу MainActivity придется приложить дополнительные усилия для связывания уровней модели и представления GeoQuiz.

Откройте файл MainActivity.kt. Создайте список объектов Question и переменную для индекса списка.

Листинг 2.5. Добавление списка Question (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    private lateinit var trueButton: Button
    private lateinit var falseButton: Button

    private val questionBank = listOf(
            Question(R.string.question_australi
a, true),
             Question(R.string.question_oceans,
true),
            Question(R.string.question_mideast,
false),
             Question(R.string.question_africa,
false),
            Question(R.string.question_americas
, true),
               Question(R.string.question_asia,
true))

    private var currentIndex = 0
    ...
}
```

Программа несколько раз вызывает конструктор `Question` и создает список объектов `Question`.

(В более сложном проекте этот список создавался бы и хранился в другом месте. Позднее мы рассмотрим более правильные варианты хранения данных модели. А пока для простоты список будет создаваться в контроллере.)

Мы собираемся использовать `questionBank`, `currentIndex` и свойства в `Question` для вывода на экран серии вопросов.

Сначала добавим свойства для `TextView` и новой кнопки. Затем получим ссылку на `TextView` и зададим текст из нее в вопрос по текущему индексу. Тем временем получим ссылку и на новую кнопку (позже мы настроим слушатель кликов для кнопки **NEXT**).

Листинг 2.6. Подключение виджета TextView (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    private lateinit var trueButton: Button
    private lateinit var falseButton: Button
    private lateinit var nextButton: Button
      private lateinit var questionTextView:
TextView
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
                                  trueButton    =
findViewById(R.id.true_button)
                                falseButton    =
findViewById(R.id.false_button)
```

```
                                    nextButton = findViewById(R.id.next_button)
                                    questionTextView = findViewById(R.id.question_text_view)

            trueButton.setOnClickListener { view: View ->
                ...
            }

            falseButton.setOnClickListener { view: View ->
                ...
            }

                        val questionTextResId = questionBank[currentIndex].textResId
            questionTextView.setText(questionTextResId)
        }
}
```

Сохраните файлы и проверьте возможные ошибки. Запустите программу GeoQuiz. Первый вопрос из массива должен отображаться в виджете `TextView`.

Теперь разберемся с кнопкой **NEXT**. Получите ссылку на кнопку, назначьте ей слушателя `View.OnClickListener`. Этот слушатель будет увеличивать индекс и обновлять текст `TextView`.

Листинг 2.7. Подключение новой кнопки (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    ...
    falseButton.setOnClickListener { view: View
->
        ...
    }

    nextButton.setOnClickListener {
          currentIndex = (currentIndex + 1) %
questionBank.size
                    val questionTextResId =
questionBank[currentIndex].textResId
        questionTextView.setText(questionTextRe
sId)

    }

                  val questionTextResId =
questionBank[currentIndex].textResId
    questionTextView.setText(questionTextResId)
}
```

Обновление переменной questionTextView осуществляется в двух разных местах. Лучше выделить этот код в функцию, как показано в листинге 2.8. Далее останется лишь вызвать эту функцию в слушателе nextButton и в конце onCreate(Bundle?) для исходного заполнения текста в представлении activity.

Листинг 2.8. Инкапсуляция с помощью функции (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        nextButton.setOnClickListener {
            currentIndex = (currentIndex + 1) %
questionBank.size
                        ~~val questionTextResId =~~
~~questionBank[currentIndex].textResId~~
                ~~questionTextView.setText(questi~~
~~onTextResId)~~
            updateQuestion()
        }

                        ~~val questionTextResId =~~
~~questionBank[currentIndex].textResId~~
            ~~questionTextView.setText(questionTe~~
~~xtResId)~~
        updateQuestion()
    }

    private fun updateQuestion() {
                    val questionTextResId =
questionBank[currentIndex].textResId
        questionTextView.setText(questionTextRe
sId)
    }
}
```

Запустите GeoQuiz и протестируйте новую кнопку **NEXT**.

Итак, с вопросами мы разобрались, пора обратиться к ответам. В текущем состоянии приложение GeoQuiz считает, что на все вопросы ответ должен быть положительным; исправим этот недостаток. И снова мы реализуем приватную функцию в `MainActivity` для инкапсуляции кода, вместо того чтобы вставлять одинаковый код в двух местах:

```
private fun checkAnswer(userAnswer: Boolean)
```

Функция получает логическую переменную, которая указывает, какую кнопку нажал пользователь: **TRUE** или **FALSE**. Ответ пользователя проверяется по ответу текущего объекта `Question`. Наконец, после определения правильности ответа функция создает `Toast` для вывода соответствующего сообщения.

Включите в файл `MainActivity.kt` реализацию `checkAnswer(Boolean)`, приведенную в листинге 2.9.

Листинг 2.9. Добавление функции checkAnswer(Boolean) (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
    private fun updateQuestion() {
        ...
    }

        private fun checkAnswer(userAnswer:
Boolean) {
                    val correctAnswer =
questionBank[currentIndex].answer
```

```
val messageResId = if (userAnswer == correctAnswer) {
            R.string.correct_toast
        } else {
            R.string.incorrect_toast
        }

            Toast.makeText(this, messageResId, Toast.LENGTH_SHORT)
                .show()
    }
}
```

Включите в слушателя кнопки вызов `checkAnswer(Boolean)`, как показано в листинге 2.10.

Листинг 2.10. Вызов функции checkAnswer(Boolean) (MainActivity.kt)

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    trueButton.setOnClickListener { view: View ->
            Toast.makeText(
                this,
                R.string.correct_toast,
                Toast.LENGTH_SHORT
            )
                .show()
        checkAnswer(true)
    }
```

```
        falseButton.setOnClickListener { view: View
    ->
                Toast.makeText(
                    this,
                    R.string.correct_toast,
                    Toast.LENGTH_SHORT
                )
                    .show()
            checkAnswer(false)
        }
        ...
    }
```

Программа GeoQuiz снова готова к работе. Давайте запустим ее на реальном устройстве.

Добавление значка

Приложение GeoQuiz работает, но пользовательский интерфейс смотрелся бы более привлекательно, если бы на кнопке **NEXT** была изображена стрелка, обращенная направо.

Изображение такой стрелки можно найти в файле решений этой книги (**www.bignerdranch.com/solutions/AndroidProgramming4e.zip**). Файл решений представляет собой набор проектов Android Studio для всех глав книги.

Загрузите файл и откройте каталог `02_MVC/GeoQuiz/app/src/main/res`. Найдите в нем подкаталоги `drawable-hdpi`, `drawable-mdpi`, `drawable-xhdpi`, `drawable-xxhdpi` и `drawable-xxxhdpi`.

Суффиксы имен каталогов обозначают экранную плотность пикселов устройства.

`mdpi` Средняя плотность (~160 dpi)

`hdpi` Высокая плотность (~240 dpi)

`xhdpi` Сверхвысокая плотность (~320 dpi)

`xxhdpi` Сверхсверхвысокая плотность (~480 dpi)

`xxxhdpi` Сверхсверхсверхвысокая плотность (~640 dpi)

(Также существуют другие категории устройств, включая ldpi и xxxhdpi, но в решениях они не используются.)

Каждый каталог содержит два графических файла: `arrow_right.png` и `arrow_left.png`. Эти файлы адаптированы для плотности пикселов, указанной в имени каталога.

Мы включим в решения GeoQuiz все файлы изображений. При запуске ОС выбирает файл изображения, наиболее подходящий для конкретного устройства, на котором выполняется приложение. Следует учитывать, что дублирование изображений увеличивает размер приложения. В данном примере это не создает серьезных проблем, потому что GeoQuiz — очень простое приложение.

Если приложение выполняется на устройстве, экранная плотность которого не соответствует ни одному признаку в именах каталогов, Android автоматически масштабирует изображение к подходящему размеру. Благодаря этому обстоятельству необязательно предоставлять изображения для всех категорий плотности. Для сокращения размера приложения можно сосредоточиться на одной или нескольких категориях высокой плотности и выборочно оптимизировать графику для уменьшенного разрешения, когда автоматическое масштабирование Android создает артефакты на устройствах с низким разрешением.

(Альтернативные варианты дублирования изображений с разной плотностью и каталог mipmap рассматриваются в главе 22.)

Добавление ресурсов в проект

Следующим шагом станет включение графических файлов в ресурсы приложения GeoQuiz.

Проверьте, чтобы на панели **Project** отображалось дерево представления Project. Раскройте содержимое узла GeoQuiz/app/src/main/res (рис. 2.6). В нем должны присутствовать папки с именами mipmap-hdpi и mipmap-xhdpi, показанные на рис. 2.6. (Также в списке присутствуют другие папки; пока не обращайте на них внимания.)

В файле решения выделите и скопируйте четыре каталога, упоминавшихся ранее: drawable-hdpi, drawable-mdpi, drawable-xhdpi, drawable-xxhdpi и drawable-xxxhdpi. В Android Studio вставьте скопированные каталоги в каталог app/src/main/res. В результате должны появиться четыре каталога для разной плотности, каждый из которых содержит файлы arrow_left.png и arrow_right.png (рис. 2.7).

Рис. 2.6. Просмотр ресурсов на панели Project

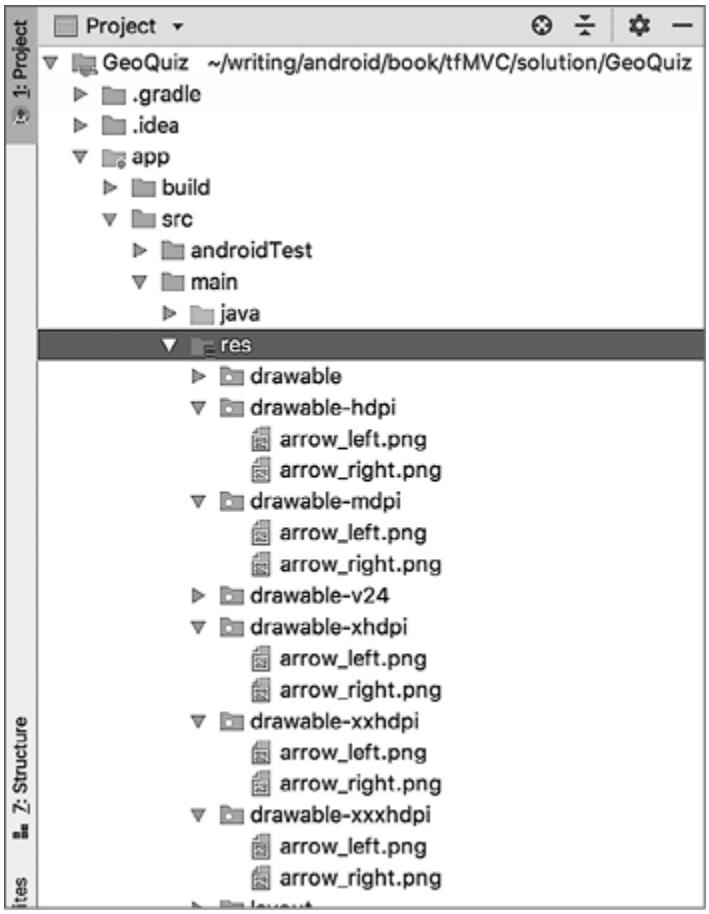


Рис. 2.7. Изображения стрелок в каталогах drawable проекта GeoQuiz

Переключив панель **Project** обратно в режим **Android**, вы увидите сводку добавленных файлов (рис. 2.8).

Рис. 2.8. Сводка изображений со стрелками в каталогах drawable приложения GeoQuiz

Процесс включения графики в приложение чрезвычайно прост. Любому файлу `.png`, `.jpg` или `.gif`, добавленному в папку `res/drawable`, автоматически назначается идентификатор ресурса. (Учтите, что имена файлов должны быть записаны в нижнем регистре и не могут содержать пробелов.)

Эти идентификаторы ресурсов не уточняются плотностью пикселов, так что вам не нужно определять плотность пикселов экрана во время выполнения; просто используйте идентификатор ресурса в коде. При запуске приложения ОС автоматически выберет изображение, подходящее для конкретного устройства.

Система ресурсов Android более подробно рассматривается в главе 3 и далее. А пока давайте заставим работать нашу стрелку.

Ссылки на ресурсы в XML

Для ссылок на ресурсы в коде используются идентификаторы ресурсов. Но мы хотим настроить кнопку **NEXT** так, чтобы в определении макета отображалась стрелка. Как включить ссылку на ресурс в разметку XML?

Да почти так же, только с немного измененным синтаксисом. Откройте файл `activity_main.xml` и добавьте два атрибута в определение виджета `Button`.

Листинг 2.11. Включение графики в кнопку NEXT (res/layout/activity_main.xml)

```
<LinearLayout ... >
    ...
    <LinearLayout ... >
        ...
    </LinearLayout>

    <Button
        android:id="@+id/next_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/next_button"
        android:drawableEnd="@drawable/arrow_ri
ght"
        android:drawablePadding="4dp" />

</LinearLayout>
```

В ресурсах XML вы ссылаетесь на другой ресурс по его типу и имени. Ссылка на строку начинается с префикса `@string/`. Ссылка на графический ресурс начинается с префикса `@drawable/`.

Имена ресурсов и структура каталогов `res` более подробно рассматриваются начиная с главы 3.

Сохраните приложение GeoQuiz, запустите его и насладитесь новым внешним видом кнопки. Протестируйте ее и убедитесь, что кнопка работает точно так же, как прежде.

Плотность пикселов

В файле `active_main.xml` вы задавали значения атрибутов в единицах `dp`. Давайте поясним, что это вообще такое.

Иногда приходится задавать значения атрибутов в определенных единицах измерения (обычно в пикселах, но иногда в точках, миллиметрах или дюймах). В большей степени это касается атрибутов, отвечающих за кегль текста, полей и отступов. Кегль текста — это высота пикселов текста на экране устройства. Отступы задают расстояния между виджетами, а поля — расстояние между внешними краями виджета и его содержимым.

Как вы видели в разделе «Добавление значка», Android автоматически масштабирует изображения под нужное значение плотности пикселов с помощью специальных папок (таких как `drawable-xhdpi`). А что произойдет, если изображения масштабируются, а поля нет? Или если пользователь установит какой-то свой, крупный размер текста?

Для решения этих проблем в Android предусмотрены плотностно-независимые единицы измерения, которые можно использовать для получения одного и того же размера на экранах с разной плотностью. Android преобразует эти единицы в пикселы во время выполнения программы, и никакой хитроумной математики тут нет (рис. 2.9).

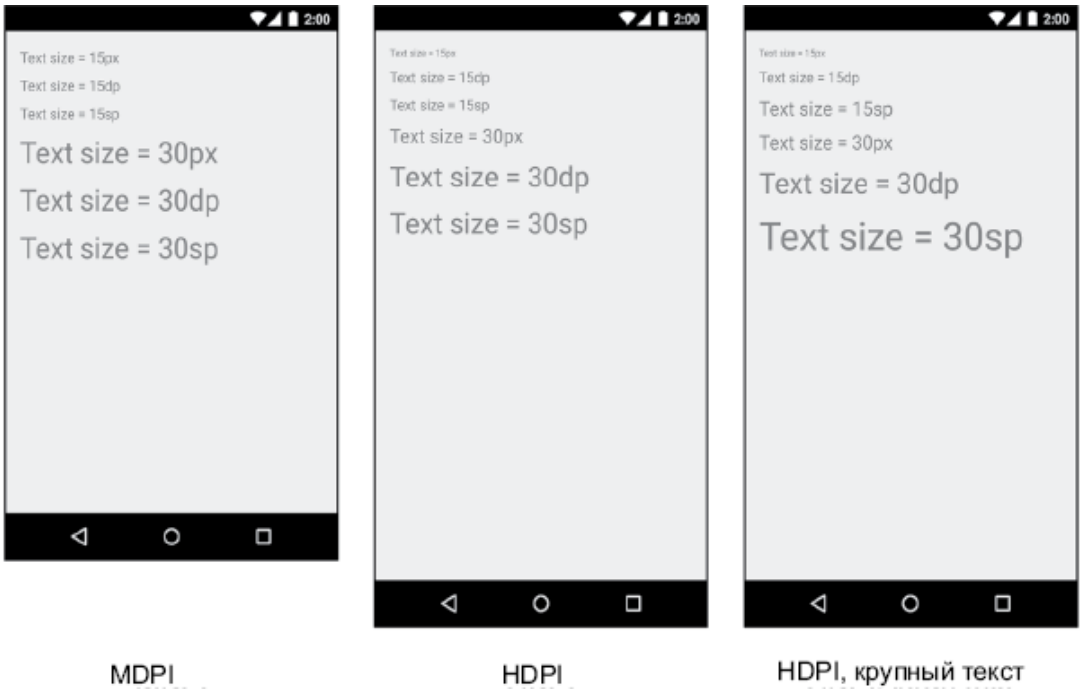


Рис. 2.9. Единицы измерения в activity в TextView

px

Сокращение от *pixel* — пиксел. Один пиксел в настройках соответствует одному пикселу на реальном экране, независимо от плотности отображения. Так как пикселы не масштабируются под плотность экрана устройства, не рекомендуется использовать такие единицы измерения.

dp

Сокращение от *density-independent pixel* (плотностно-независимый пиксел). Обычно эта единица измерения используется для полей, отступов или чего-либо еще, для чего в противном случае пришлось бы задавать размер в пикселах. Один dp всегда составляет 1/160 дюйма на экране устройства. Размер получится одинаковым, независимо от плотности экрана: при более высокой плотности экрана 1 dp просто будет включать больше пикселов, чтобы геометрический размер элемента сохранился.

sp

Сокращение от *scale-independent pixel* (масштабно-независимый пиксел). Это такие не зависящие от плотности пикселы, которые также учитывают предпочтения пользователя в отношении размера шрифта. Эта единица почти всегда используется для установки размера текста на экране.

`pt`, `mm`, `in`

Масштабируемые единицы, как и `dp`, которые позволяют задавать размеры интерфейса в точках (1/72 дюйма), миллиметрах или дюймах. Однако мы не рекомендуем их использовать: не все устройства будут правильно масштабировать такие единицы.

На практике и в этой книге вы почти всегда будете использовать `dp` и `sp`. Android будет переводить эти значения в пикселы во время выполнения.

Запуск на устройстве

Конечно, баловаться с приложением на эмуляторе — это забавно. Но еще больше удовольствия можно получить, взаимодействуя с приложением на физическом устройстве Android. В этом разделе мы настроим систему, устройство и приложение так, чтобы GeoQuiz работало на вашем устройстве.

Прежде всего подключите устройство к системе. Если разработка ведется в macOS, ваша система должна немедленно распознать устройство. Системе Windows может потребоваться установка драйвера `adb` (Android Debug Bridge). Если Windows не может найти драйвер `adb`, загрузите его с сайта производителя устройства.

Во-вторых, необходимо разрешить для устройства отладку через USB. Для этого вам нужно найти в меню флажок **Developeroptions**, который по умолчанию не отображается.

Чтобы включить его, откройте экран **Settings⇒AboutTablet/Phone** и быстро нажмите кнопку **BuildNumber** семь раз. Спустя несколько нажатий вы увидите, сколько шагов отделяет вас от становления разработчиком. Когда вам напишут «You are now a developer», можно будет прекратить. После этого вернитесь в меню **Settings**, найдите раздел **Developeroptions** и установите флажок **USBdebugging**.

Настройки разных устройств серьезно различаются. Если у вас возникнут проблемы с включением отладки на вашем устройстве, обратитесь за помощью по адресу **developer.android.com/tools/device.html**.

Чтобы убедиться в том, что устройство было успешно опознано, откройте инструмент **Logcat** в нижней части окна Android Studio. В левом верхнем углу панели находится раскрывающийся список подключенных устройств (рис. 2.10). В списке должны присутствовать как эмулятор AVD, так и физическое устройство.

Если у вас возникнут трудности с распознаванием устройства, прежде всего убедитесь в том, что устройство включено и для него включена отладка через USB.

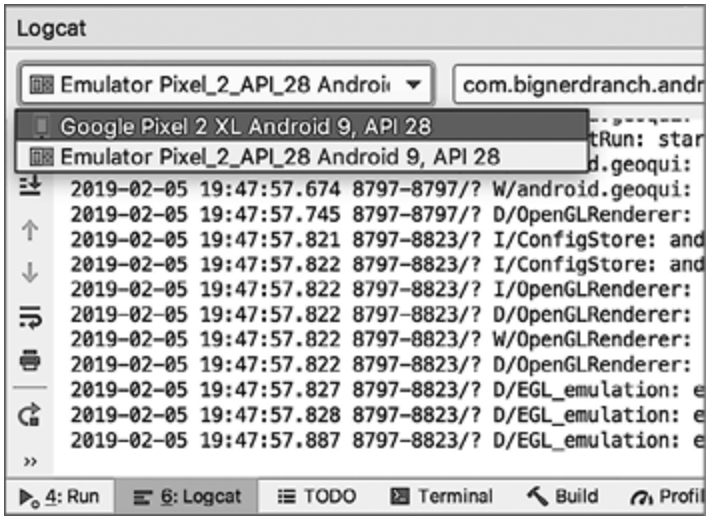


Рис. 2.10. Просмотр списка подключенных устройств

Если устройство так и не появилось в списке, дополнительную информацию можно найти на сайте разработчиков Android. Начните со страницы **developer.android.com/tools/device.html** или обратитесь на тематический форум за дополнительной информацией о решении проблемы.

Запустите GeoQuiz так, как это делалось ранее. Android Studio предложит выбор между запуском на виртуальном устройстве и на физическом устройстве, подключенном к системе. Выберите физическое устройство; GeoQuiz запустится на выбранном устройстве.

Если Android Studio не предлагает выбрать устройство, а GeoQuiz запускается в эмуляторе, проверьте описанную ранее процедуру и убедитесь в том, что устройство подключено. Затем проверьте правильность конфигурации запуска; чтобы изменить параметры конфигурации, выберите раскрывающийся список в верхней части окна (рис. 2.11).

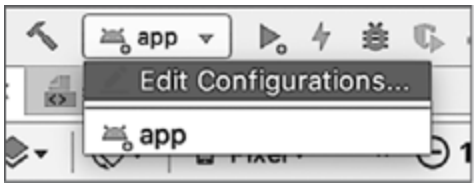


Рис. 2.11. Конфигурации запуска

Выберите в списке строку **EditConfigurations**; откроется новое окно с подробной информацией о текущей конфигурации (рис. 2.12).

Выберите в левой части окна строку **app** и убедитесь в том, что в разделе **DeploymentTargetOptions** выбран вариант **OpenSelectDeploymentTargetDialog**, а флажок **Usesamedeviceforfuturelaunches** сброшен. Нажмите кнопку **OK** и запустите приложение заново. На этот раз вам будет предложено выбрать устройство для запуска.

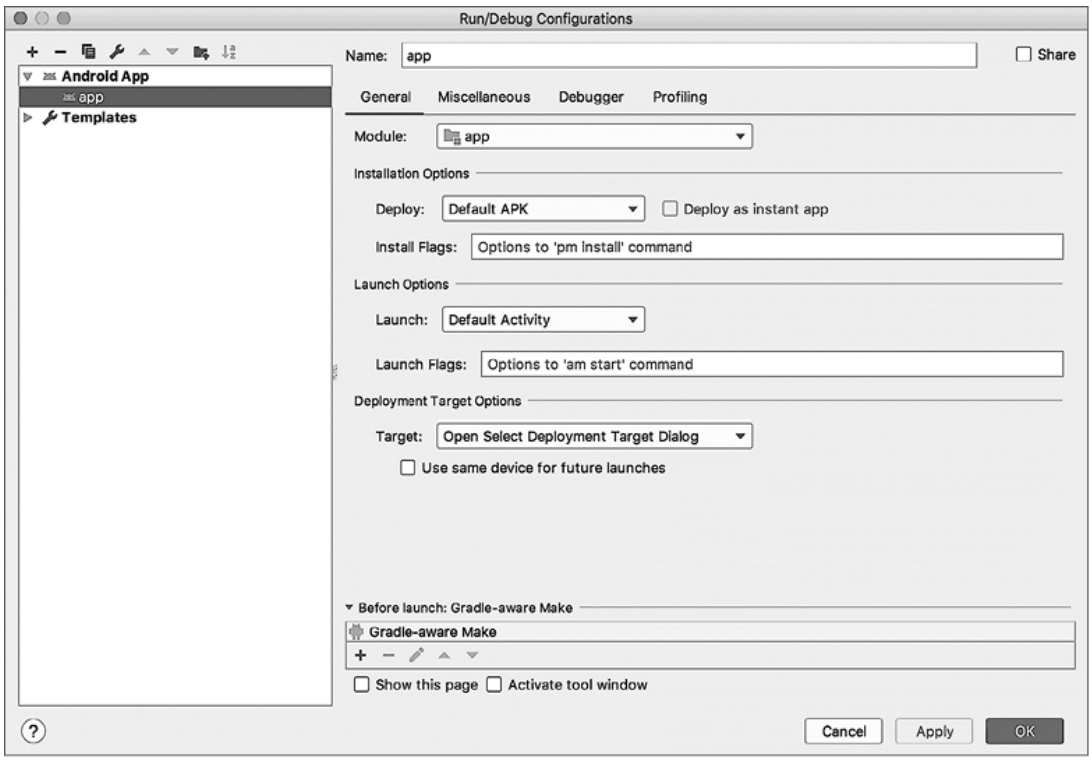


Рис. 2.12. Свойства конфигурации запуска

Упражнение. Добавление слушателя для TextView

Кнопка **NEXT** удобна, но было бы неплохо, если бы пользователь мог переходить к следующему вопросу простым нажатием на виджет `TextView`.

Подсказка. Для `TextView` можно использовать слушателя `View.OnClickListener`, который использовался с `Button`, потому что класс `TextView` также является производным от `View`.

Упражнение. Добавление кнопки возврата

Добавьте кнопку для возвращения к предыдущему вопросу. Пользовательский интерфейс должен выглядеть примерно так, как показано на рис. 2.13.

Это очень полезное упражнение. В нем вам придется вспомнить многое из того, о чем говорилось в двух предыдущих главах.

Рис. 2.13. Теперь с кнопкой возврата!

Рис. 2.14. Кнопки только со значками

Упражнение. От Button к ImageButton

Возможно, пользовательский интерфейс будет смотреться еще лучше, если на кнопках будут отображаться только значки, как

на рис. 2.14.

Для этого оба виджета должны относиться к типу ImageButton (вместо обычного Button).

Виджет ImageButton является производным от ImageView, в отличие от виджета Button, производного от TextView. Диаграммы их наследования изображены на рис. 2.15.

Атрибуты text и drawable кнопки **NEXT** можно заменить одним атрибутом ImageView:

```
<    Button ImageButton
  android:id="@+id/next_button"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
     android:text="@string/next_button"
      android:drawableEnd="@drawable/arrow_righ
t"
     android:drawablePadding="4dp"
  android:src="@drawable/arrow_right"
  />
```

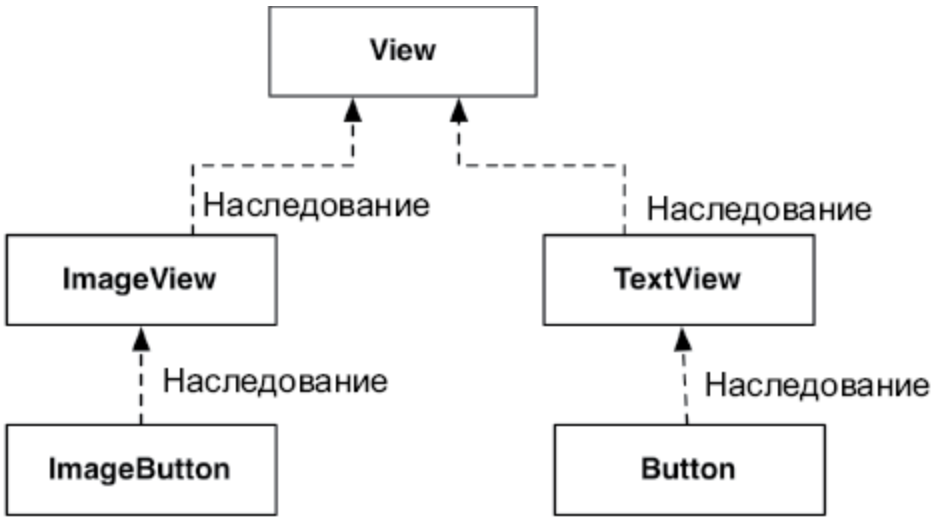


Рис. 2.15. Диаграмма наследования ImageButton и Button

Конечно, вам также придется внести изменения в `MainActivity`, чтобы этот класс работал с `ImageButton`.

После того как вы замените эти кнопки на кнопки `ImageButton`, Android Studio выдаст предупреждение об отсутствии атрибута `android:contentDescription`. Этот атрибут обеспечивает доступность контента для читателей с ослабленным зрением. Строка, заданная этому атрибуту, читается экранным диктором (при включении соответствующих настроек в системе пользователя).

Наконец, добавьте атрибут `android:contentDescription` в каждый элемент `ImageButton`.

3. Жизненный цикл activity

В этой главе вы научитесь решать ужасную — и чрезвычайно распространенную — «проблему поворота». Вы также научитесь работать с механизмами, лежащими в основе проблемы поворота, и отображать альтернативный макет, если устройство повернуто в альбомный режим (рис. 3.1).

Рис. 3.1. Макет GeoQuiz при повороте в альбомный режим

Поворот GeoQuiz

GeoQuiz работает нормально... пока вы не повернете устройство. Во время выполнения GeoQuiz нажмите кнопку **NEXT**, чтобы перейти к следующему вопросу, а затем поверните устройство. (Если программа выполняется в эмуляторе, нажмите кнопку поворота налево или направо на плавающей панели инструментов (рис. 3.2).)

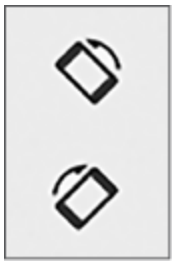


Рис. 3.2. Управление поворотом

Если эмулятор не вращается после нажатия одной из кнопок вращения, можно включить автоповорот. Проведите вниз от верхней части экрана, чтобы открыть панель быстрого доступа. Нажмите на значок автоповорота — он третий слева (рис. 3.3). Значок изменит цвет с серого на голубой, и функция автоповорота теперь включена.

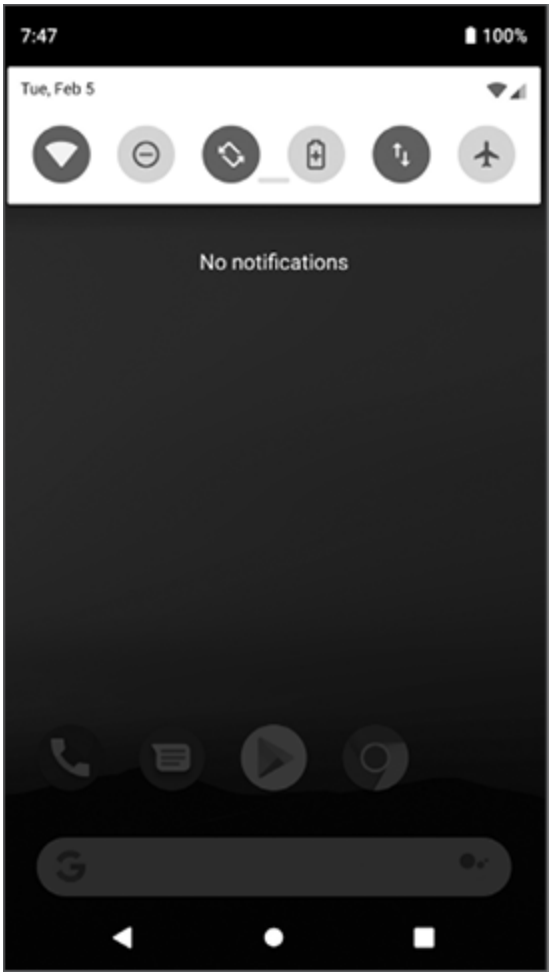


Рис. 3.3. Настройка автоповорота на панели быстрого доступа

После поворота мы снова видим первый вопрос. Почему это произошло и как исправить ошибку?

Решению этой проблемы посвящена глава 4, но сначала нам важно понять корень проблемы, чтобы в дальнейшем избежать кучи лишних багов.

Обратные вызовы состояния и жизненного цикла activity

У каждого экземпляра `Activity` имеется жизненный цикл. Во время этого жизненного цикла activity проходит четыре возможных состояния: выполнение, приостановка, остановка и несуществование. Для каждого перехода у `Activity` существует функция, которая оповещает activity об изменении состояния. На рис. 3.4 изображен жизненный цикл, состояния и функции activity.

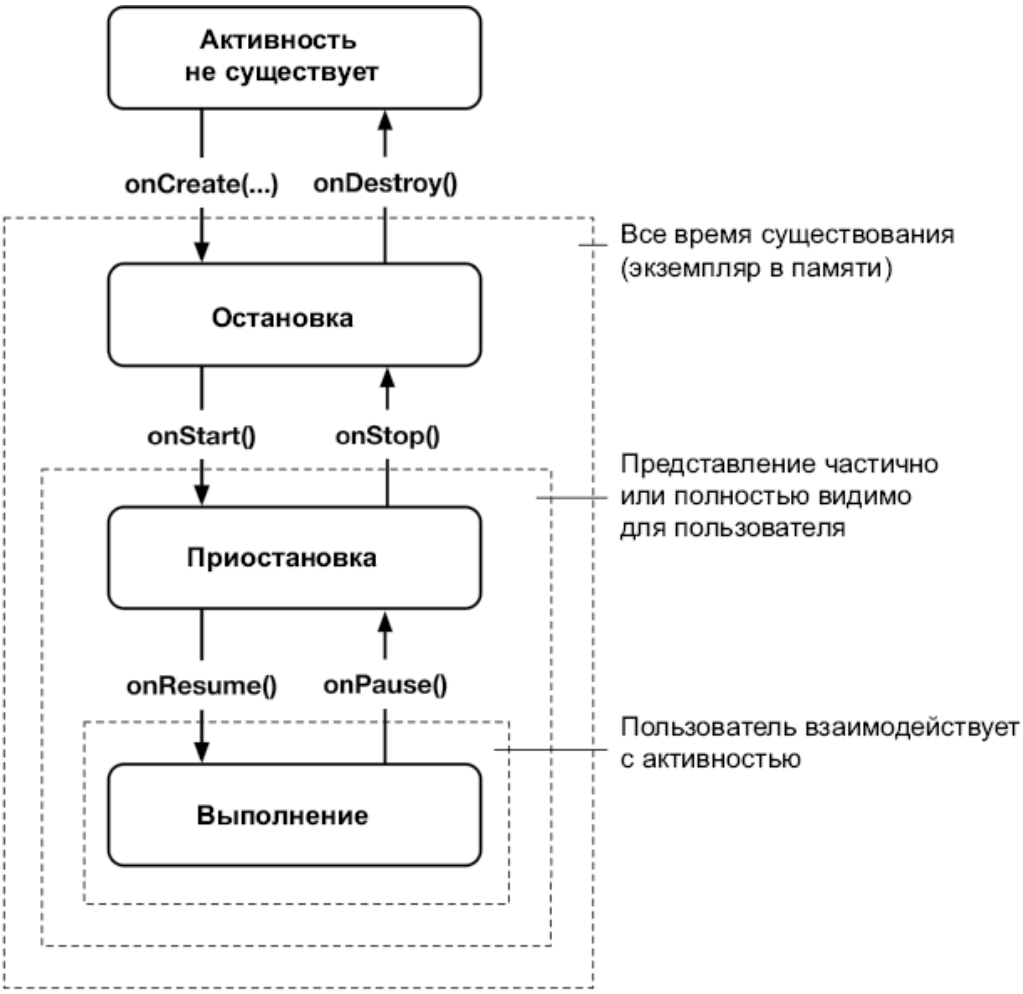


Рис. 3.4. Диаграмма состояния activity

На рис. 3.4 для каждого состояния указано, существует ли экземпляр activity в памяти, видим ли он для пользователя и находится ли он на переднем плане (получает ли ввод от пользователя). Сводка этой информации приведена в табл. 3.1.

Таблица 3.1. Состояния activity

Состояние	Находится в памяти?	Видима ли для пользователя?	На переднем плане?
Activity не существует	Нет	Нет	Нет
Activity остановлена	Да	Нет	Нет

Activity приостановлена	Да	Да/частично*	Нет
Activity выполняется	Да	Да	Да

* В зависимости от обстоятельств приостановленная activity может быть видима полностью или частично. Эта тема рассматривается более подробно в разделе «Анализ жизненного цикла activity на примере».

Несуществующей называется activity, которая еще не была запущена либо только что была уничтожена (например, если пользователь нажал кнопку «Назад»). Иногда такие activity еще называют «уничтоженными». В памяти нет экземпляра и нет связанного с ним представления, которое пользователь мог бы видеть или с которым он мог бы взаимодействовать.

Остановленной называется activity, экземпляр которой находится в памяти, но на экране ее не видно. Это переходное состояние возникает, когда activity запускается, и повторяется всякий раз, когда представление полностью закрыто (например, когда пользователь выводит другую activity на передний план, нажимает кнопку «Главная» или использует список приложений для переключения задач).

Приостановленная activity видна, хоть и не на переднем плане. Это возможно, например, когда пользователь запускает новое диалоговое окно с темой или прозрачное действие поверх нее. Activity также может быть полностью видимой, но не на переднем плане, если пользователь просматривает две activity в многооконном режиме (в «режиме разделенного экрана»).

Выполняемая activity находится в памяти, полностью видима и находится на переднем плане. Это та activity, с которой пользователь в данный момент взаимодействует. В любой момент времени только одна activity во всей системе может находиться в выполняемом состоянии. Это означает, что если

одна activity переходит в возобновленное состояние, то другая, скорее всего, приостанавливается.

Подклассы `Activity` могут использовать функции, представленные на рис. 3.4, для выполнения необходимых действий во время критических переходов жизненного цикла activity. Эти функции часто называются *обратными вызовами жизненного цикла*.

Одна из таких функций вам уже знакома — `onCreate(Bundle?)`. ОС вызывает эту функцию после создания экземпляра activity, но до его отображения на экране.

Как правило, activity переопределяет `onCreate(Bundle?)` для подготовки пользовательского интерфейса в следующем порядке:

- заполнение виджетов и их вывод на экран (вызов `setContentView(int)`);
- получение ссылок на заполненные виджеты;
- назначение слушателей виджетам для обработки взаимодействия с пользователем;
- подключение к внешним данным модели.

Важно понимать, что вы никогда не вызываете `onCreate(Bundle?)` или другие функции жизненного цикла `Activity` в своих приложениях: вы переопределяете их в подклассах activity, а Android вызывает их в нужный момент времени (в зависимости от того, что делает пользователь и что происходит в системе), для того чтобы оповестить приложение об изменении состояния.

Регистрация событий жизненного цикла activity

В этом разделе мы переопределим функции жизненного цикла, чтобы отслеживать основные переходы жизненного цикла `MainActivity`. Реализации ограничиваются регистрацией в журнале сообщения о вызове функции. Эти сообщения помогут вам наблюдать за изменениями состояния `MainActivity` при выполнении действий пользователем.

Создание сообщений в журнале

В Android класс `android.util.Log` отправляет журнальные сообщения в общий журнал системного уровня. Класс `Log` предоставляет несколько функций регистрации сообщений. Следующая функция чаще всего встречается в этой книге:

```
public static Int d(String tag, String msg)
```

Буква `d` означает «debug» (отладка) и относится к уровню регистрации сообщений. (Уровни `Log` более подробно рассматриваются в последнем разделе этой главы.) Первый параметр определяет источник сообщения, а второй — его содержимое.

Первая строка обычно содержит константу `TAG`, значением которой является имя класса. Это позволяет легко определить источник конкретного сообщения.

Откройте файл `MainActivity.kt` и добавьте константу `TAG` в `MainActivity`.

Листинг 3.1. Добавление константы TAG (MainActivity.kt)

```
import ...
```

```
private const val TAG = "MainActivity"

class MainActivity : AppCompatActivity() {
    ...
}
```

Включите в `onCreate(Bundle?)` вызов `Log.d(...)` для регистрации сообщения.

Листинг 3.2. Включение команды регистрации сообщения в onCreate(Bundle?) (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d(TAG, "onCreate(Bundle?) called")
    setContentView(R.layout.activity_main)
    ...
}
```

Теперь переопределите еще пять функций в `MainActivity`; для этого добавьте следующие определения после `onCreate(Bundle?)`.

Листинг 3.3. Переопределение функций жизненного цикла (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
```

```
}

override fun onStart() {
    super.onStart()
    Log.d(TAG, "onStart() called")
}

override fun onResume() {
    super.onResume()
    Log.d(TAG, "onResume() called")
}

override fun onPause() {
    super.onPause()
    Log.d(TAG, "onPause() called")
}

override fun onStop() {
    super.onStop()
    Log.d(TAG, "onStop() called")
}

override fun onDestroy() {
    super.onDestroy()
    Log.d(TAG, "onDestroy() called")
}

private fun updateQuestion() {
    ...
}
...
```

```
}
```

Обратите внимание на вызовы реализаций суперкласса перед регистрацией сообщений. Эти вызовы обязательны. Вызов реализации суперкласса должен располагаться в первой строке реализации переопределения любой функции обратного вызова.

Возможно, вы заметили ключевое слово `override`. Оно приказывает компилятору проследить за тем, чтобы класс действительно содержал переопределяемую функцию. Например, компилятор сможет предупредить вас об опечатке в имени функции:

```
override fun onCreat(savedInstanceState: Bundle?) {
    ...
}
```

Родительский класс `AppCompatActivity` не содержит функцию `onCreat(Bundle?)`, поэтому компилятор сообщает об ошибке. Это позволит вам исправить опечатку, вместо того чтобы догадываться об ошибке по странному поведению запущенного приложения.

Панель LogCat

При запуске GeoQuiz данные **LogCat** появляются в нижней части окна Android Studio (рис. 3.5). Если панель **LogCat** не видна, в нижней части окна Android Studio перейдите на вкладку **LogCat**.

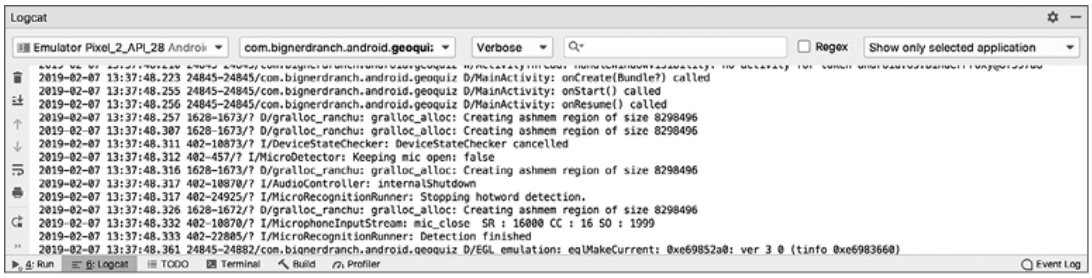


Рис. 3.5. Android Studio с панелью LogCat

Вы увидите свои собственные сообщения, а также системный вывод. Чтобы облегчить поиск сообщений, вы можете отфильтровать вывод, используя значение, установленное для константы `TAG`. На панели **Logcat** откройте раскрывающийся список в правом верхнем углу с надписью **Showonlyselectedapplication**. Это фильтр, который в настоящее время настроен на показ сообщений только от вашего приложения.

Выберите в раскрывающемся списке пункт **EditFilterConfiguration**. Нажмите кнопку **+** для создания нового фильтра. Введите имя фильтра **MainActivity** и строку **MainActivity** в поле **LogTag** (рис. 3.6).

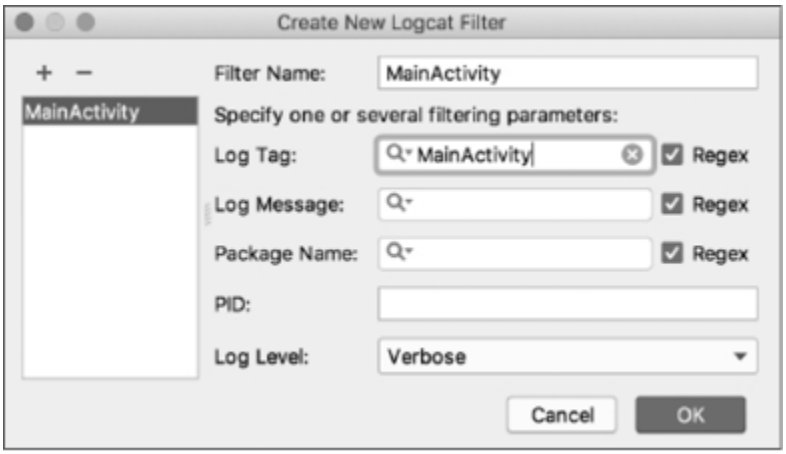


Рис. 3.6. Создание фильтра на панели LogCat

Нажмите кнопку **OK**; после этого будут видны только сообщения с тегом **MainActivity** (рис. 3.7).

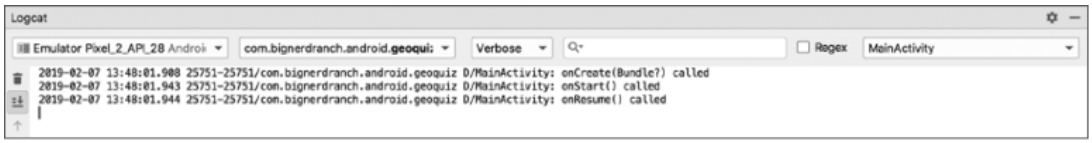


Рис. 3.7. При запуске GeoQuiz происходит создание, запуск и возобновление activity

Анализ жизненного цикла activity на примере

Как видите, после запуска GeoQuiz и создания исходного экземпляра `MainActivity` были вызваны три функции жизненного цикла: `onCreate(Bundle?)`, `onStart()` и `onResume()` (рис. 3.7). Экземпляр `MainActivity` после этого находится в состоянии выполнения (он находится в памяти, виден пользователю и находится на переднем плане).

По мере изучения этой книги вы будете переопределять различные функции жизненного цикла activity, чтобы решить те или иные задачи для вашего приложения. По ходу дела вы будете узнавать все больше об использовании каждой функции. А пока немного позабавимся и посмотрим, как ведет себя жизненный цикл в стандартных сценариях использования, поработаем с вашим приложением и посмотрим журналы на панели **Logcat**.

Временная остановка activity

Если приложение GeoQuiz не запущено на вашем устройстве, запустите его в программе Android Studio. Нажмите на устройстве кнопку «Главный экран», и `MainActivity` полностью исчезнет с экрана. В каком состоянии теперь `MainActivity`? Взгляните на панель **LogCat**. Activity приложения получила вызовы `onPause()` и `onStop()`, но не `onDestroy()` (рис. 3.8).

Рис. 3.8. Нажатие кнопки «Главный экран» приводит к остановке activity

Нажатие кнопки «Главный экран» сообщает Android: «Я сейчас займусь другим делом, но потом могу вернуться. Этот экран мне еще понадобится». Android приостанавливает, а в конечном итоге и останавливает activity.

Это означает, что после нажатия кнопки «Главный экран» ваш экземпляр `MainActivity` пребывает в состоянии остановки (находится в памяти, не виден, не активен). Android делает это для того, чтобы быстро и легко перезапустить `MainActivity` с того момента, на котором была прервана работа, если вы решите вернуться к GeoQuiz в будущем.

(Это не полное описание работы кнопки «Главный экран». Остановленные activity могут уничтожаться по усмотрению ОС — за более полным описанием обращайтесь к главе 4.)

Вернитесь к приложению GeoQuiz, выбрав его *сводный экран* в диспетчере задач. Для этого нажмите кнопку «Обзор приложений» рядом с кнопкой «Главный экран» (рис. 3.9).

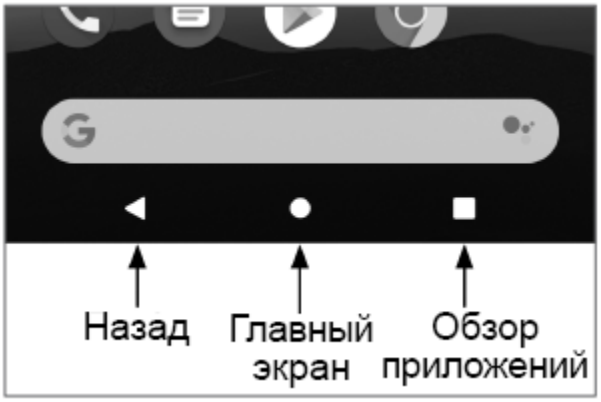


Рис. 3.9. Кнопки «Назад», «Главный экран» и «Обзор приложений»

Если на вашем устройстве нет кнопки «Обзор приложений», а лишь одна кнопка «Главный экран» (как показано на рис. 3.10), проведите пальцем вверх от нижней части экрана, чтобы открыть диспетчер задач. Если на устройстве не работает ни один из этих методов, обратитесь к руководству пользователя устройства.

Рис. 3.10. Одиночная кнопка «Главный экран»

Каждая миниатюра экрана в диспетчере задач представляет приложение, с которым пользователь взаимодействовал в прошлом (рис. 3.11). (В документации разработчика этот экран называется «сводным экраном», *overview screen*.)

Рис. 3.11. Диспетчер задач

Коснитесь сводного экрана приложения GeoQuiz; `MainActivity` заполнит экран.

Из информации на панели **Logcat** следует, что activity получила вызовы `onStart()` и `onResume()`. Обратите внимание: функция `onCreate(...)` при этом не вызывалась. Дело в том, что activity `MainActivity` находилась в состоянии остановки после нажатия пользователем кнопки «Главный экран». Так как экземпляр activity все еще находится в памяти, создавать его не нужно. Вместо этого достаточно запустить activity (перевести ее в видимое приостановленное состояние),

а затем продолжить ее выполнение (перевести в состояние выполнения с получением ввода от пользователя).

Ранее мы говорили, что activity может «зависнуть» в состоянии паузы, будучи либо частично видимой (например, при запуске новой activity с прозрачным фоном или не в полном экране), либо полностью видимой (в многооконном режиме). Остановимся подробнее на многооконном режиме.

Многооконный режим доступен только на Android 7.0 Nougat и выше, поэтому если ваше устройство работает на более ранней версии Android, для его тестирования потребуется эмулятор. Откройте диспетчер задач еще раз и коснитесь и удерживайте значок в верхней части GeoQuiz. Выберите режим **Splitscreen** (показано слева на рис. 3.12), и в нижней части экрана появится новое окно со списком приложений (в центре на рис. 3.12). Щелкните по одному из сводных экранов, чтобы запустить соответствующее приложение.

Запустится многооконный режим, при этом GeoQuiz окажется в верхнем окне, а второе приложение — в нижнем окне (показано справа на рис. 3.12).

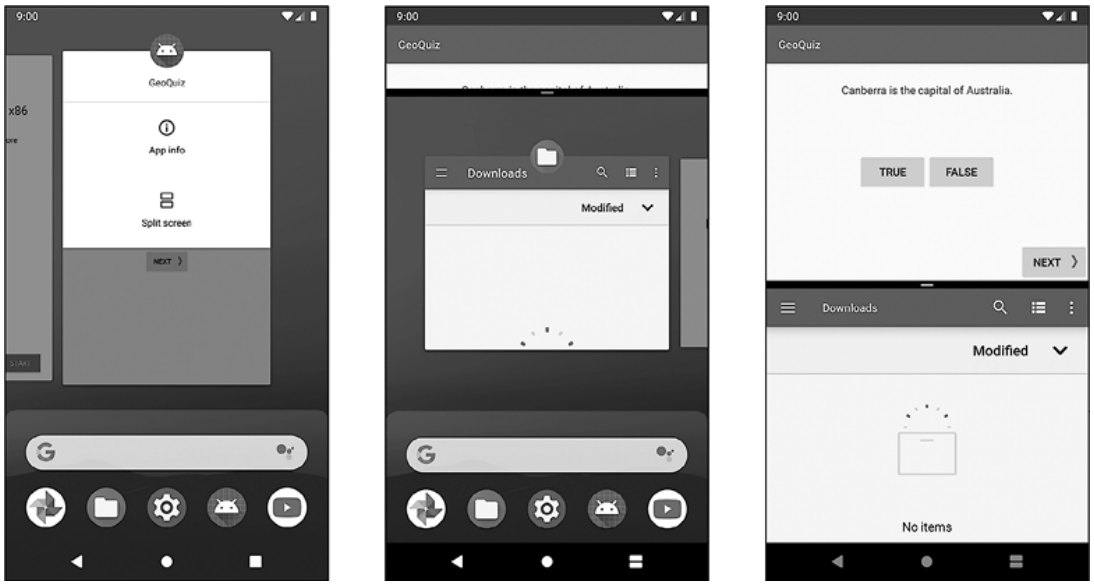


Рис. 3.12. Открытие двух приложений в многооконном режиме

Теперь щелкните по приложению в нижнем окне и посмотрите на сообщения журнала на панели **Logcat**. В `MainActivity` была вызвана функция `onPause()`, поэтому activity теперь находится в состоянии паузы. Нажмите на GeoQuiz в верхнем окне. В `MainActivity` была вызвана функция `onResume()`, поэтому activity теперь находится в возобновленном состоянии.

Чтобы выйти из многооконного режима, проведите разделителем окон в середине экрана вниз, чтобы закрыть нижнее окно. (Проведите разделителем вверх, чтобы закрыть верхнее окно.)

Завершение activity

Нажмите на устройстве кнопку «Назад», а затем проверьте вывод на панели **LogCat**. Activity приложения получила вызовы `onPause()`, `onStop()` и `onDestroy()` (рис. 3.13). Экземпляр `MainActivity` после этого уже не существует (он не находится в памяти, не виден и, конечно, не находится на переднем плане.)

Рис. 3.13. Нажатие кнопки «Назад» приводит к уничтожению activity

Нажимая кнопку «Назад», вы сообщаете Android: «Я *завершилработу* с activity, и она мне больше не нужна». Android уничтожает activity и удаляет все следы ее существования из

памяти, чтобы избежать неэффективного расходования ограниченных ресурсов устройства.

То же самое можно сделать, смахнув приложение в сторону из диспетчера задач. Вы можете также завершать activity программно с помощью функции `Activity.finish()`.

Поворот activity

А теперь вернемся к ошибке, обнаруженной в начале этой главы. Запустите GeoQuiz, нажмите кнопку **NEXT** для перехода к следующему вопросу, а затем поверните устройство. (Чтобы имитировать поворот в эмуляторе, нажмите на кнопки поворота на панели инструментов.)

После поворота GeoQuiz снова выведет первый вопрос. Чтобы понять, почему это произошло, просмотрите вывод **LogCat**. Он выглядит примерно так, как показано на рис. 3.14.

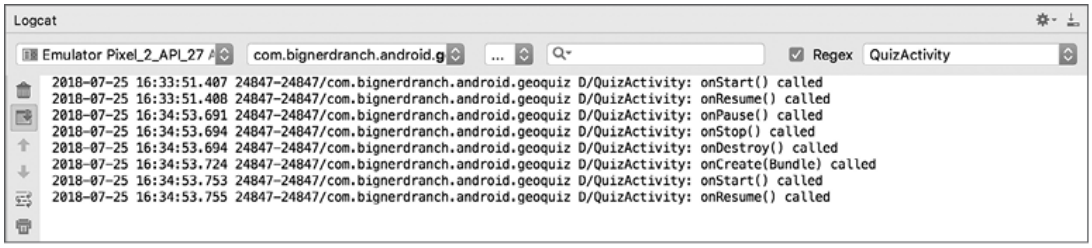


Рис. 3.14. MainActivity мертв. Да здравствует MainActivity!

Когда вы поворачиваете устройство, экземпляр `MainActivity`, который вы видели, уничтожается, и вместо него создается новый экземпляр. Снова поверните устройство — произойдет еще один цикл уничтожения и возрождения.

Из-за этого и возникает ошибка. Каждый раз, когда вы поворачиваете устройство, экземпляр `MainActivity` полностью уничтожается. Значение, хранившееся в переменной `currentIndex` этого экземпляра, стирается из памяти. Таким образом, при повороте устройства приложение

GeoQuiz «забывает», какой вопрос вы смотрели, а после завершения поворота Android создает новый экземпляр `MainActivity` с нуля. Переменная `currentIndex` инициализируется с `0` в вызове `onCreate(Bundle?)`, а пользователь начинает с первого вопроса.

В главе 4 мы исправим ошибку, но сначала подробнее разберемся, почему это происходит.

Конфигурации устройств и жизненный цикл ресурса

Поворот приводит к изменению *конфигурации устройства*. Конфигурация устройства представляет собой набор характеристик, описывающих текущее состояние конкретного устройства. К числу характеристик, определяющих конфигурацию, относится ориентация экрана, плотность пикселов, размер экрана, тип клавиатуры, режим стыковки, язык и многое другое.

Как правило, приложения предоставляют альтернативные ресурсы для разных конфигураций устройств. Пример такого рода нам уже встречался: вспомните, как мы включали в проект несколько изображений стрелки для разной плотности пикселов.

При *изменении конфигурации во время выполнения* может оказаться, что приложение содержит ресурсы, лучше подходящие для новой конфигурации. По этой причине Android уничтожает activity, ищет ресурсы, которые лучше всего подходят для новой конфигурации, и заново строит экземпляр activity с этими ресурсами. Чтобы увидеть, как работает этот механизм, мы создадим альтернативный ресурс, который Android найдет и использует при изменении ориентации экрана.

Создание макета для альбомной ориентации

На панели **Project** щелкните правой кнопкой мыши по каталогу `res` и выберите команду **New⇒Androidresourcefile** в контекстном меню. Открывается окно со списками типов ресурсов и квалификаторами этих типов (наподобие изображенного на рис. 3.15). Введите **activity_main** в поле **Filename**. Выберите в раскрывающемся списке **Resourcetype** строку **layout**. Оставьте в списке **Sourceset** значение **main**.

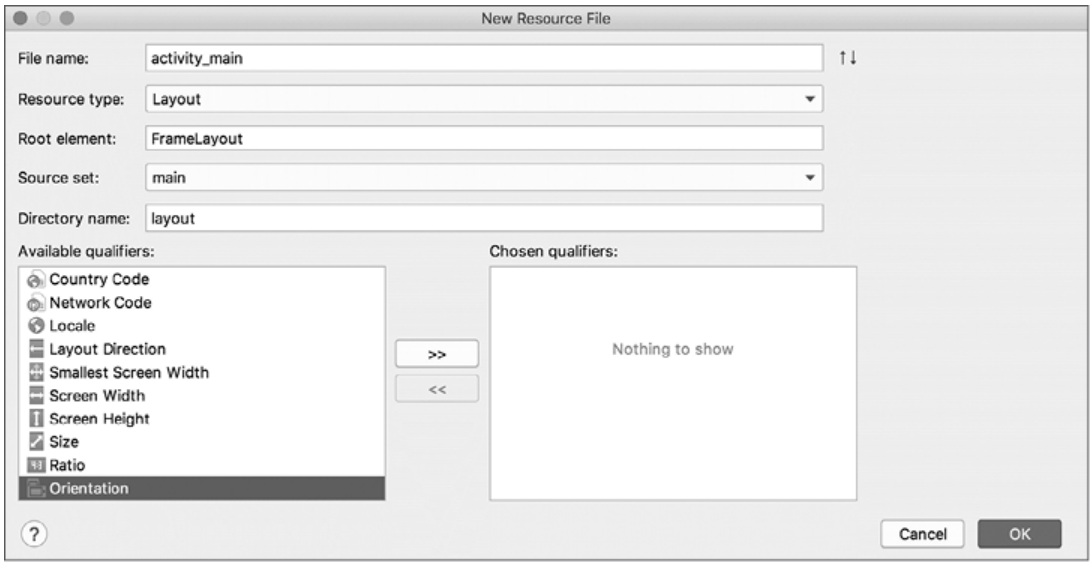


Рис. 3.15. Создание нового каталога ресурсов

Теперь необходимо выбрать способ уточнения ресурсов макета. Выберите в списке **Availablequalifiers** строку **Orientation** и нажмите кнопку **>>**, чтобы переместить значение **Orientation** в поле **Chosenqualifiers**.

Наконец, убедитесь в том, что в списке **ScreenOrientation** выбрано значение **Landscape** (рис. 3.16). Проверьте, указано ли в поле **Directoryname** имя каталога **layout-land**. Окно выглядит несколько экзотично, но единственная его цель — задание имени каталога. Нажмите кнопку **OK**.

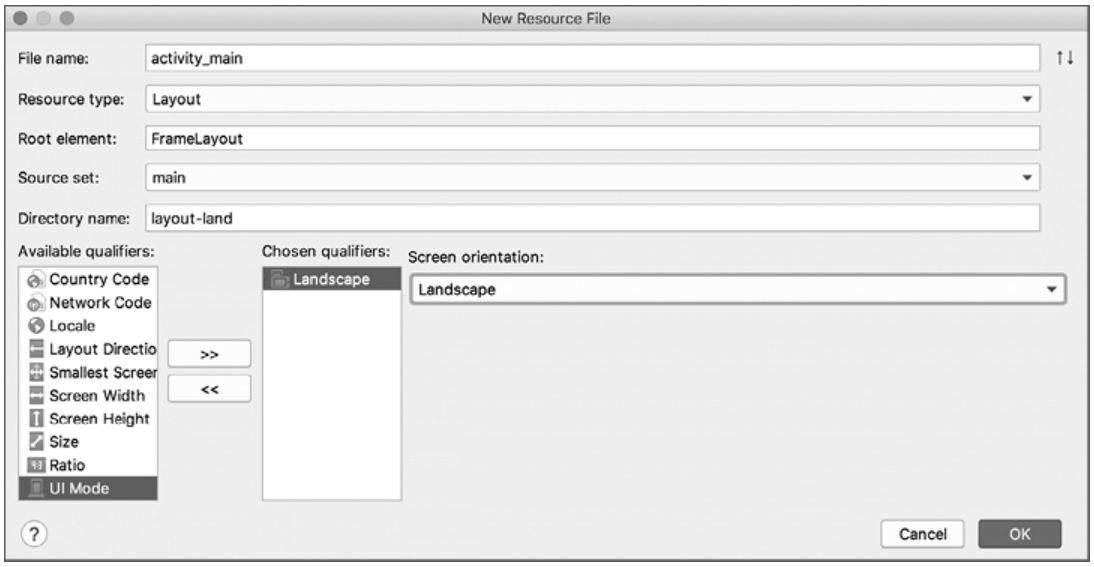


Рис. 3.16. Создание файла res/layout-land/activity_main.xml

Android Studio создаст в вашем проекте папку `res/layout-land/` и поместит в нее новый файл макета под названием `activity_main.xml`. Переключите панель **Project** в режим **Project**, чтобы увидеть файлы и папки в вашем проекте такими, какие они есть на самом деле. Переключитесь обратно в режим **Android**, чтобы увидеть сводку файлов.

Как видно, конфигурационные квалификаторы в подпапках `res` позволяют Android определять, какие ресурсы лучше соответствуют конфигурации устройства. Суффикс `-land` — еще один пример конфигурационного квалификатора. По конфигурационным квалификаторам подкаталогов `res` Android определяет, какие ресурсы лучше всего подходят для текущей конфигурации устройства. Список конфигурационных квалификаторов, поддерживаемых Android, и обозначаемых ими компонентов конфигурации устройств находится по адресу **developer.android.com/guide/topics/resources/providing-resources.html**.

Теперь у вас есть макет для альбомной ориентации и макет по умолчанию. Когда устройство находится в альбомной

ориентации, Android использует ресурсы макета из папки res/layout-land. В противном случае он будет использовать ресурсы по умолчанию в каталоге res/layout/. Обратите внимание, что файлы макета должны иметь одно и то же имя, чтобы на них можно было ссылаться с одним и тем же идентификатором ресурса.

В настоящее время на макете res/layout-land/activity_main.xml есть лишь пустой экран. Чтобы исправить это, скопируйте все содержимое, кроме открывающего и закрывающего тегов корневого элемента LinearLayout, из файла res/layout/activity_main.xml. Вставьте код в файл res/layout-land/activity_main.xml между открывающим и закрывающим тегами элемента FrameLayout.

Теперь внесите необходимые изменения, как показано в листинге 3.4, в альбомный макет, чтобы он отличался от макета по умолчанию.

Листинг 3.4. Настройка альбомного макета (res/layout-land/activity_main.xml)

```
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:id="@+id/question_text_view"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
            android:gravity="center"
        android:layout_gravity="center_horizont
al"
        android:padding="24dp"
        tools:text="@string/question_australia"
/>

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
            android:orientation="horizontal"
        android:layout_gravity="center_vertical
|center_horizontal">

        <Button
            .../>

        <Button
            .../>

    </LinearLayout>

    <Button
        android:id="@+id/next_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|right"
        android:text="@string/next_button"
        android:drawableEnd="@drawable/arrow_ri
ght"
```

```
            android:drawablePadding="4dp"/>

    </FrameLayout>
```

FrameLayout — это самый простой вариант ViewGroup, в котором дочерние объекты не организованы каким-то конкретным образом. В этом макете дочерние виды располагаются в соответствии со значением атрибутов android:layout_gravity. Именно поэтому вы добавили атрибут android:layout_gravity в TextView, LinearLayout и Button — они все являются дочерними элементами FrameLayout. Дочерние Button у вложенного LinearLayout можно не редактировать, так как они не являются прямыми дочерними объектами FrameLayout.

Снова запустите GeoQuiz. Поверните устройство в альбомную ориентацию, чтобы увидеть новый макет (рис. 3.17). Конечно, в программе используется не только новый макет, но и новый экземпляр MainActivity.

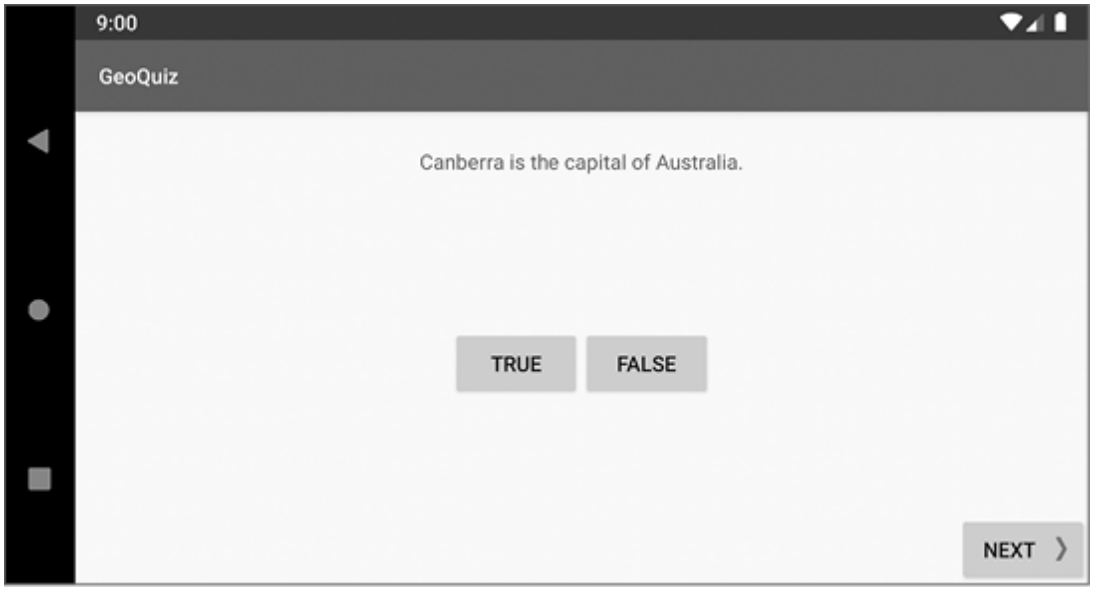


Рис. 3.17. MainActivity в альбомной ориентации

Снова поверните устройство в книжную ориентацию, чтобы увидеть макет по умолчанию и другой экземпляр `MainActivity`.

Что хорошего в приложении, которое перезагружается при повороте устройства, даже если в нем отличный макет? В главе 4 вы узнаете, как исправить ошибку GeoQuiz, вызванную проблемой вращения.

Для любознательных: обновление UI и мультиоконный режим

До версии Android 7.0 Nougat большинство activity очень мало времени проводило в приостановленном состоянии. Это состояние было лишь промежуточным между состоянием выполнения и остановки. Из-за этого многие разработчики предполагали, что обновлять интерфейс нужно только тогда, когда activity находится в возобновленном состоянии. Обычной практикой было использование функций `onResume()` и `onPause()` для запуска или остановки любых текущих обновлений, связанных с пользовательским интерфейсом (таких как анимация или обновление данных).

Когда в версии Nougat был введен многооконный режим, это нарушило принцип, гласящий, что на экране видна только выполняемая activity. А это, в свою очередь, сломало предполагаемое поведение многих приложений. Теперь, когда пользователь находится в многооконном режиме, приостановленные activity могут быть полностью видны в течение длительного периода времени. Разумеется, пользователь ожидает, что эти приостановленные activity будут вести себя так, как если бы они выполнялись.

Простой пример — видео. Предположим, у вас есть приложение pre-Nougat, предназначенное для

воспроизведения видео. Вы запускаете (или возобновляете) воспроизведение видео в функции `onResume()` и приостанавливаете в `onPause()`. И вот появляется многооконный режим, но ваше приложение останавливает воспроизведение при переключении пользователя на приложение во втором окне. Пользователи недовольны, ведь они хотят смотреть видео, *пока* пишут сообщение в отдельном окне.

К счастью, исправить это просто: нужно переместить ваше возобновление воспроизведения и паузу в функции `onStart()` и `onStop()`. Это относится к любым данным, обновляемым в реальном времени, например фотогалереям, которые прокручивают фотографии, когда они попадают в поток Flickr (как вы увидите позже в этой книге).

С момента появления Nougat ваша activity должна обновлять пользовательский интерфейс на протяжении всего видимого жизненного цикла, от `onStart()` до `onStop()`.

К сожалению, не все усвоили этот урок, и многие приложения до сих пор некорректно работают в многооконном режиме. Чтобы исправить это, команда разработчиков Android в ноябре 2018 года представила спецификацию для поддержки *мультивозобновления* в многооконном режиме. Этот режим означает, что полностью видимая activity в каждом из окон будет находиться в состоянии выполнения, когда устройство находится в многооконном режиме, независимо от того, с каким из окон пользователь взаимодействует в данный момент.

Вы можете использовать многооконный режим для своего приложения, когда оно работает на Android 9.0 Pie, добавив метаданные в манифест Android: `<meta-dataandroid:name="android.allow_multiple_resumed_activities"android:value="true"/>`. Вы узнаете об этом больше в главе 6.

Стоит отметить, что *мультивозобновление* будет работать только на тех устройствах, производитель которых реализовал соответствующую спецификацию. И на момент написания этой статьи ни одно устройство на рынке так не умеет. Однако ходят слухи, что спецификация станет обязательной в следующей версии Android Q.

Пока *мультивозобновление* не станет легкодоступным стандартом для большинства устройств на рынке, вам придется использовать ваши знания о жизненном цикле activity, чтобы понимать, где разместить код обновления пользовательского интерфейса. По мере изучения этой книги мы будем много практиковаться в этом.

Для любознательных: уровни регистрации

Когда вы используете класс `android.util.Log` для регистрации сообщений в журнале, вы задаете не только содержимое сообщения, но и *уровень* регистрации, определяющий важность сообщения. Android поддерживает пять уровней регистрации (табл. 3.2). Каждому уровню соответствует своя функция класса `Log`. Регистрация данных в журнале сводится к вызову соответствующей функции `Log`.

Таблица 3.2. Функции и уровни регистрации

Уровень регистрации	Функция	Примечания
ERROR	`Log.e(...)`	Ошибки
WARNING	`Log.w(...)`	Предупреждения
INFO	`Log.i(...)`	Информационные сообщения
DEBUG	`Log.d(...)`	Отладочный вывод (может фильтроваться)
VERBOSE	`Log.v(...)`	Только для разработчиков

Каждая функция регистрации существует в двух вариантах: один получает строку TAG и строку сообщения, а второй получает эти же аргументы и экземпляр Throwable, упрощающий регистрацию информации о конкретном исключении, которое может быть выдано вашим приложением. В листинге 3.5 представлены примеры сигнатуры функций регистрации.

Листинг 3.5. Различные способы регистрации в Android

```
// Регистрация сообщения с уровнем регистрации "debug"
Log.d(TAG, "Current question index: $currentIndex")

try {
    val question = questionBank[currentIndex]
} catch (ex: ArrayIndexOutOfBoundsException) {
        // Регистрация сообщения с уровнем регистрации "error" с трассировкой стека исключений
    Log.e(TAG, "Index was out of bounds", ex)
}
```

Упражнение. Предотвращение ввода нескольких ответов

После того как пользователь введет ответ на вопрос, заблокируйте кнопки этого вопроса, чтобы предотвратить возможность ввода нескольких ответов.

Упражнение. Вывод оценки

После того как пользователь введет ответ на все вопросы, отобразите уведомление с процентом правильных ответов. Удачи!

4. Сохранение состояния интерфейса

В Android много делается для своевременного обеспечения альтернативных ресурсов. Однако уничтожение и повторное создание activity связаны с большими трудностями. Например, в GeoQuiz может возникнуть баг при возврате к первому вопросу при повороте устройства.

Чтобы исправить эту ошибку, экземпляр `MainActivity` должен после поворота знать старое значение атрибута `currentIndex`. Вам нужен способ сохранять эти данные на случай изменения конфигурации, например вращения.

В этой главе мы исправим проблему GeoQuiz с потерей состояния при вращении путем сохранения данных пользовательского интерфейса в `ViewModel`. Кроме того, мы поработаем с более сложной для обнаружения, но не менее проблематичной ошибкой — потерей состояния пользовательского интерфейса при завершении процесса. Для этого мы воспользуемся механизмом сохранения состояния Android.

Добавление зависимостей ViewModel

Мы добавим в проект класс `ViewModel`. Класс `ViewModel` взят из библиотеки Android Jetpack под названием `lifecycle-extensions`, одной из многих библиотек, которые вы будете использовать на протяжении всей книги. (Подробнее о Jetpack поговорим позже в этой главе.) Чтобы использовать класс, сначала необходимо включить библиотеку в список *зависимостей* вашего проекта.

Зависимости вашего проекта находятся в файле под названием `build.gradle` (напомним, что Gradle — это

инструмент для сборки Android). На панели **Project** в режиме **Android** разверните раздел **Gradlescripts** и изучите его содержимое. В вашем проекте есть два файла `build.gradle`: один для проекта в целом и один для модуля приложения. Откройте файл `build.gradle`, расположенный в модуле приложения. Вы увидите что-то похожее на листинг 4.1.

Листинг 4.1. Зависимости Gradle (app/build.gradle)

```
apply plugin: 'com.android.application'

apply plugin: 'kotlin-android'

apply plugin: 'kotlin-android-extensions'

android {
    ...
}

dependencies {
        implementation fileTree(dir: 'libs',
include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-
stdlib-jdk7:$kotlin_version"
                                    implementation
'androidx.appcompat:appcompat:1.0.0-beta01'
    ...
}
```

Первая строка в разделе `dependencies` гласит, что проект зависит от всех `.jar` файлов в каталоге `libs`. В других строках приведены библиотеки, которые были автоматически

включены в проект в соответствии с настройками, выбранными при создании.

Gradle также позволяет задавать новые зависимости. Во время компиляции приложения Gradle будет находить, загружать и включать нужные зависимости. Все, что вам нужно, это указать точное название, а Gradle сделает все остальное.

Добавьте зависимость `lifecycle-extensions` в ваш файл `app/build.gradle`, как показано в листинге 4.2. Точное размещение строки в разделе не имеет значения, но лучше держать код в порядке и помещать новые зависимости под последней зависимостью в `implementation`.

Листинг 4.2. Добавление зависимости lifecycle-extensions (app/build.gradle)

```
dependencies {
    ...
                                    implementation
'androidx.constraintlayout:constraintlayout:1.1
.2'
                                    implementation
'androidx.lifecycle:lifecycle-extensions:2.0.0'
    ...
}
```

Когда вы вносите какие-либо изменения в файл `build.gradle`, Android Studio предложит синхронизировать файл (рис. 4.1).

Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly. Sync now

Рис. 4.1. Подсказка по синхронизации Gradle

Компилятор просит Gradle обновить сборку после внесения изменений или добавления/удаления зависимостей. Чтобы выполнить синхронизацию, щелкните мышью по ссылке **SyncNow** в запросе или выберите команду **File⇒SyncProjectwithGradleFiles**.

Добавление ViewModel

Теперь вернемся к `ViewModel`. Объект `ViewModel` связан с одним конкретным экраном и отлично подходит для размещения логики форматирования данных для отображения на этом экране. Объект `ViewModel` связан с модельным объектом и «декорирует» модель, добавляя ей функциональность для отображения на экране, которая в самой модели вам не нужна. Использование `ViewModel` позволяет собрать все данные о том, что должно быть на экране, в одном месте, отформатировать данные и легко получить доступ к конечному результату.

`ViewModel` входит в состав пакета `androidx.lifecycle`, в котором также есть связанные с жизненным циклом API, в том числе зависимые от жизненного цикла. Такие компоненты наблюдают за жизненным циклом другого компонента, например activity, и используют эту информацию в своих целях.

Google создал пакет `androidx.lifecycle` и его содержимое именно для работы с жизненным циклом activity (и другими жизненными циклами, о которых вы узнаете позже в этой книге). Например, вы узнаете о `LiveData`, еще одном компоненте жизненного цикла, из главы 11. А в главе 25 мы

объясняем, как создавать свои зависимые от жизненного цикла компоненты.

Теперь вы можете создать свой подкласс `ViewModel` под названием `QuizViewModel`, для чего на панели **Project** щелкните правой кнопкой мыши по пакету `com.bignerdranch.android.geoquiz` и выберите команду **New⇒KotlinFile/Class** в контекстном меню. Введите имя **QuizViewModel** и выберите **Class** в раскрывающемся списке **Kind**.

В файле `QuizViewModel.kt` добавьте блок `init` и перепишите функцию `onCleared()`. Запишите в журнал создание и уничтожение экземпляра `QuizViewModel` — см. пример в листинге 4.3.

Листинг 4.3. Создание класса ViewModel (QuizViewModel.kt)

```
private const val TAG = "QuizViewModel"

class QuizViewModel : ViewModel() {

    init {
                Log.d(TAG, "ViewModel instance
created")
    }

    override fun onCleared() {
        super.onCleared()
        Log.d(TAG, "ViewModel instance about to
be destroyed")
    }
}
```

Функция onCleared() вызывается непосредственно перед уничтожением ViewModel. В этом месте удобно выполнять уборку мусора, например снимать наблюдение с источника данных. Сейчас нам важен тот факт, что объект ViewModel вот-вот будет уничтожен, так что вы можете изучить его жизненный цикл (как изучали жизненный цикл MainActivity в главе 3).

Откройте файл MainActivity.kt и в функции OnCreate(...) свяжите activity с экземпляром QuizViewModel.

Листинг 4.4. Доступ к ViewModel (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        setContentView(R.layout.activity_main)

            val provider: ViewModelProvider =
ViewModelProviders.of(this)
                        val quizViewModel =
provider.get(QuizViewModel::class.java)
             Log.d(TAG, "Got a QuizViewModel:
$quizViewModel")

                                trueButton =
findViewById(R.id.true_button)
        ...
    }
    ...
```

```
}
```

Класс `ViewModelProviders` (обратите внимание на множественное число в слове Providers) предоставляет экземпляры класса `ViewModelProvider`. Ваш вызов `ViewModelProviders.of(this)` создает и возвращает `ViewModelProvider`, связанный с activity.

`ViewModelProvider`, в свою очередь, передает activity экземпляр `ViewModel`. Вызов `provider.get(QuizViewModel::class.java)` возвращает экземпляр `QuizViewModel`. Обычно эти функции идут вместе, например:

```
ViewModelProviders.of(this).get(QuizViewModel::
class.java)
```

`ViewModelProvider` работает как реестр `ViewModel`. Когда activity делает первый запрос `QuizViewModel`, `ViewModelProvider` создает и возвращает новый экземпляр `QuizViewModel`. Когда activity запрашивает `QuizViewModel` после изменения конфигурации, экземпляр, который был создан изначально, возвращается. Когда activity завершается (например, когда пользователь нажимает кнопку «Назад»), пара `ViewModel-Activity` удаляется из памяти.

Жизненный цикл ViewModel и ViewModelProvider

Из главы 3 вы узнали, что activity может иметь четыре состояния: выполнение, остановка, приостановка и несуществование. Вы также узнали о различных способах уничтожения activity: пользователем, завершающим activity, или с помощью системы, разрушающей ее в результате изменения конфигурации.

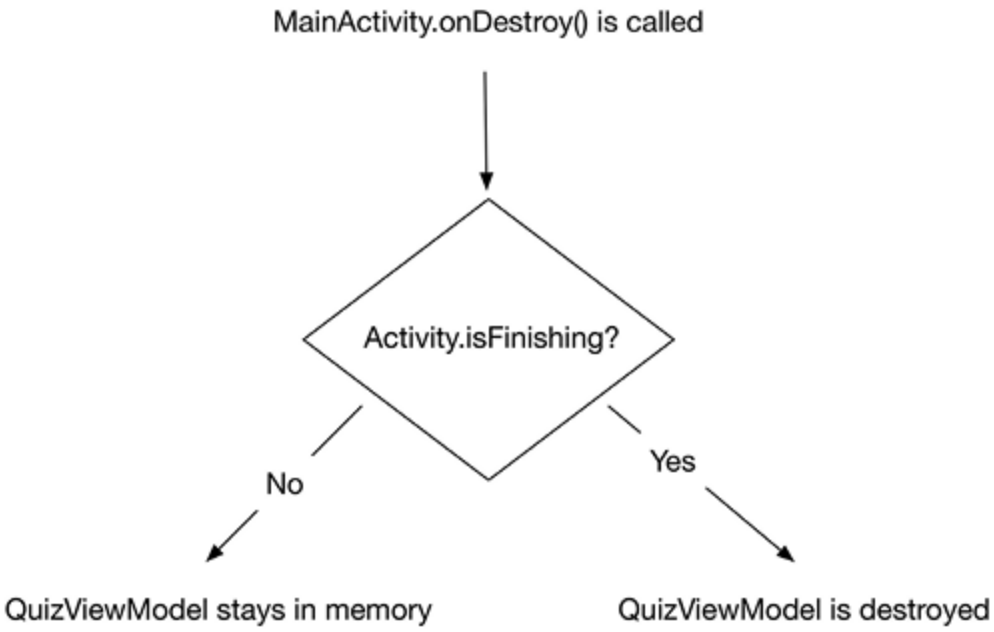
Когда пользователь завершает activity, ожидается, что состояние UI сбросится на исходное. Когда пользователь поворачивает activity, ожидается, что состояние пользовательского интерфейса окажется таким же, как и до поворота.

Вы можете определить, какой из этих двух сценариев будет иметь место, с помощью свойства activity `isFinishing`. Если свойство `isFinishing` истинно, activity уничтожается, когда пользователь завершает ее (например, нажав кнопку «Назад» или смахнув карточку приложения из диспетчера задач). Если свойство `isFinishing` ложно, activity уничтожается системой из-за изменения конфигурации.

Но вам не нужно следить за свойством `isFinishing` и сохранять состояние пользовательского интерфейса вручную при значении `false`, чтобы оправдать ожидания пользователя. Вместо этого вы можете сохранять данные о состоянии интерфейса с помощью `ViewModel`, даже если конфигурация изменяется. Жизненный цикл `ViewModel` лучше отражает ожидания пользователя: он нормально переносит изменения конфигурации и уничтожается только тогда, когда связанная с ним activity будет завершена.

Нужно проассоциировать экземпляр `ViewModel` с жизненным циклом activity, как вы это делали в листинге 4.4. С этого момента `ViewModel`*привязана* к жизненному циклу activity. Это означает, что `ViewModel` останется в памяти, независимо от состояния activity, пока она не будет завершена. После завершения activity (например, путем нажатия пользователем кнопки «Назад») экземпляр `ViewModel` тоже уничтожается (рис. 4.2).

QuizViewModel остается в памяти



MainActivity.onDestroy() вызывается

нет

Активность завершается?

QuizViewModel уничтожается

да

Рис. 4.2. QuizViewModel, привязанная к MainActivity

Это означает, что `ViewModel` остается в памяти во время изменения конфигурации, например, при вращении. Во время изменения конфигурации экземпляр activity уничтожается и воссоздается вновь, но привязанные к activity `ViewModel`-и остаются в памяти. Это изображено на рис. 4.3, на примере `MainActivity` и `QuizViewModel`.

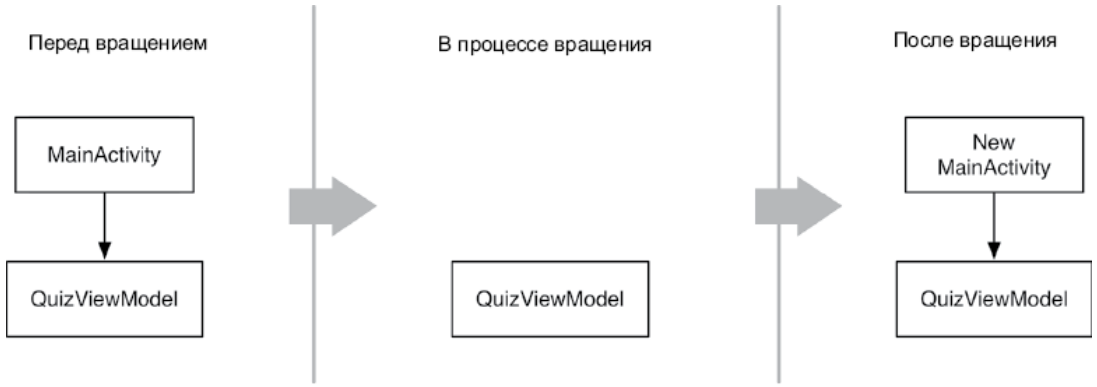


Рис. 4.3. Поведение MainActivity и QuizViewModel при вращении

Чтобы посмотреть на это в действии, запустите GeoQuiz. На панели **LogCat** выберите пункт **EditFilterConfiguration** в раскрывающемся списке, чтобы создать новый фильтр. В поле **LogTag** введите **QuizViewModel|MainActivity** (то есть два имени класса с логическим ИЛИ в виде значка | между ними), чтобы выводились только записи, связанные с определенными классами. Назовите фильтр **ViewModelAndActivity** (можно использовать и другое имя, которое для вас имеет смысл) и нажмите **OK** (рис. 4.4).

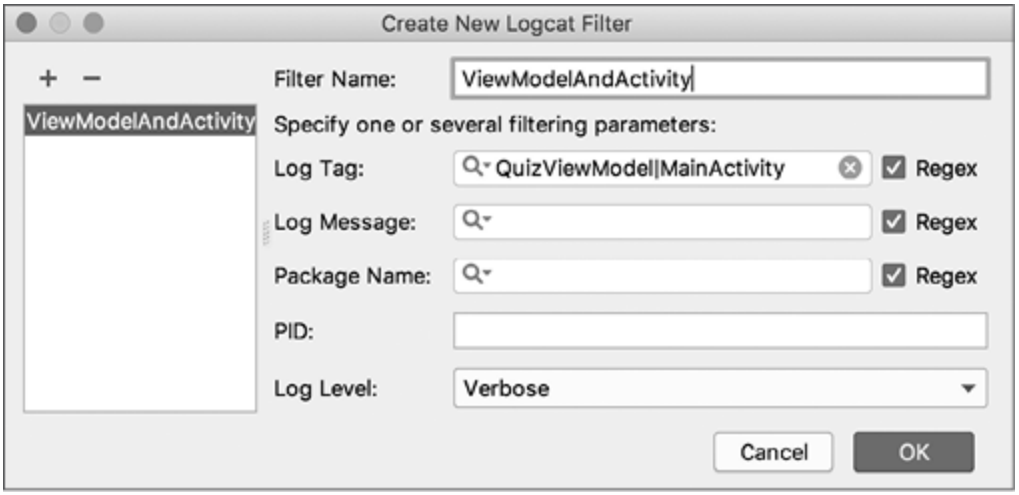


Рис. 4.4. Фильтрация журналов QuizViewModel и MainActivity

Посмотрите в журнал. Когда `MainActivity` впервые запускается и делает запрос на `ViewModel` в `OnCreate(...)`,

создается экземпляр QuizViewModel, что видно по журналу (рис. 4.5).

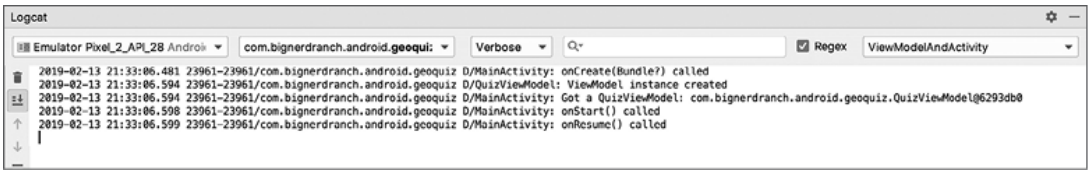


Рис. 4.5. Созданный экземпляр QuizViewModel

Поверните устройство. В журнале видно, что activity уничтожена (рис. 4.6), а QuizViewModel — нет. Когда после поворота создается новый экземпляр MainActivity, он запрашивает QuizViewModel. Так как оригинал QuizViewModel все еще находится в памяти, ViewModelProvider возвращает тот же экземпляр, а не создает новый.

В завершение нажмите кнопку «Назад». Вызывается функция QuizViewModel.onCleared(), указывающая, что экземпляр QuizViewModel скоро будет уничтожен, как видно из журналов (рис. 4.7). QuizViewModel уничтожается вместе с экземпляром MainActivity.

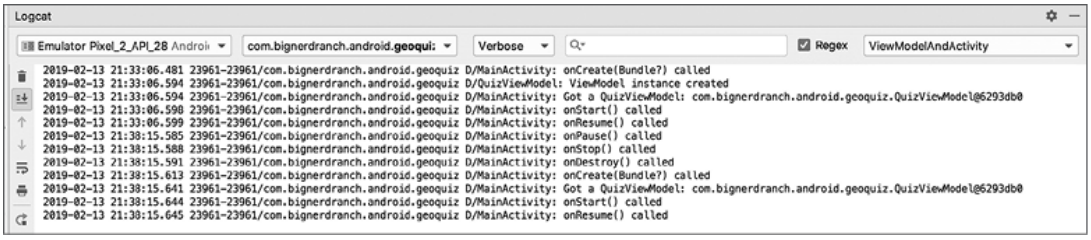


Рис. 4.6. Уничтожение MainActivity после пересоздания. QuizViewModel сохраняется в памяти

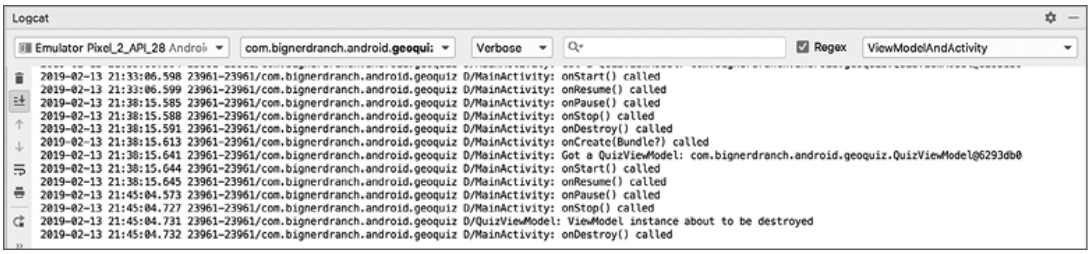


Рис. 4.7. Уничтожение MainActivity и QuizViewModel

Взаимоотношения между `MainActivity` и `QuizViewModel` однонаправленные. Activity ссылается на `ViewModel`, но `ViewModel` не имеет доступа к activity. `ViewModel` никогда не должен ссылаться на activity или виджет, иначе получится *утечкапамяти*.

Утечка памяти возникает тогда, когда один объект содержит сильную ссылку на другой объект, который должен быть разрушен. Сильная ссылка не дает сборщику мусора удалить объект из памяти. Утечки памяти из-за изменения конфигурации возникают часто. Подробный разговор о сильных ссылках и сборщиках мусора уже выходит за пределы этой книги. Если вы не уверены в этих понятиях, мы рекомендуем почитать о них в книгах по Kotlin или Java.

Ваш экземпляр `ViewModel` сохраняется в памяти после вращения, в то время как исходный экземпляр activity уничтожается. Если в `ViewModel` содержалась сильная ссылка на экземпляр исходной activity, возникнут две проблемы: во-первых, первоначальный экземпляр activity не будет удален из памяти и таким образом возникает утечка, во-вторых, `ViewModel` будет содержать ссылку на уже неактуальную activity. Если `ViewModel` попытается обновить ее, возникнет исключение `IllegalStateException`.

Добавление данных в ViewModel

Сейчас, наконец, пришло время исправить ошибку, связанную с поворотом GeoQuiz. `QuizViewModel` не уничтожается при повороте, и `MainActivity` сохраняет данные о состоянии интерфейса своей activity в экземпляре `QuizViewModel`, который после поворота тоже сохраняется.

Мы копируем вопрос и текущие данные об индексах из вашей activity в ваш `ViewModel` вместе со всей сопутствующей логикой. Начнем с удаления свойств `currentIndex` и `questionBank` из `MainActivity` (листинг 4.5).

Листинг 4.5. Извлечение данных модели из activity (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
        private val questionBank = listOf(
            Question(R.string.question_australia, true),
            Question(R.string.question_oceans, true),
            Question(R.string.question_mideast, false),
            Question(R.string.question_africa, false),
            Question(R.string.question_americas, true),
            Question(R.string.question_asia, true)
        )

        private var currentIndex = 0
    ...
```

```
}
```

Теперь нужно вставить свойства `currentIndex` и `questionBank` в `QuizViewModel`, как показано в листинге 4.6.

Удалите модификатор доступа `private` из `currentIndex`, чтобы значение свойства было доступно внешним классам вроде `MainActivity`. При этом модификатор `private` нужно оставить в `questionBank`, так как `MainActivity` не будет взаимодействовать с `questionBank` напрямую. Вместо этого он будет вызывать функции и вычисляемые свойства, которые вы добавляете в `QuizViewModel`. Во время редактирования `QuizViewModel` удалите регистрацию `init` и `onCleared()`, так как они повторно использоваться не будут.

Листинг 4.6. Вставка данных модели в QuizViewModel (QuizViewModel.kt)

```
class QuizViewModel : ViewModel() {

        init {
                Log.d(TAG, "ViewModel instance
created")
        }

        override fun onCleared() {
            super.onCleared()
                Log.d(TAG, "ViewModel instance
about to be destroyed")
        }

        private var currentIndex = 0
```

```
    private val questionBank = listOf(
            Question(R.string.question_australia,
true),
                Question(R.string.question_oceans,
true),
               Question(R.string.question_mideast,
false),
                Question(R.string.question_africa,
false),
              Question(R.string.question_americas,
true),
          Question(R.string.question_asia, true)
    )
}
```

Теперь добавим в QuizViewModel функцию перехода к следующему вопросу. Кроме того, добавим вычисляемые свойства, чтобы вернуть текст и ответ на текущий вопрос.

Листинг 4.7. Добавление бизнес-логики в QuizViewModel (QuizViewModel.kt)

```
class QuizViewModel : ViewModel() {

    var currentIndex = 0

    private val questionBank = listOf(
        ...
    )

    val currentQuestionAnswer: Boolean
```

```
                                        get()    =
questionBank[currentIndex].answer

    val currentQuestionText: Int
                                        get()    =
questionBank[currentIndex].textResId

    fun moveToNext() {
            currentIndex = (currentIndex + 1) %
questionBank.size
    }
}
```

Ранее мы говорили, что `ViewModel` хранит все данные, связанные с потребностями экрана, форматирует их и облегчает доступ. Это позволяет удалить логику представления из activity, что, в свою очередь, делает саму activity проще — а это хорошо. Любая логика, которую вы можете описать в activity, может быть непреднамеренно затронута жизненным циклом activity. Кроме того, это позволяет activity отвечать за обработку только того, что появляется на экране, а не внутреннюю обработку данных.

Несмотря на вышесказанное, нужно будет оставить функции `updateQuestion()` и `checkAnswer(Boolean)` в `MainActivity`. Эти функции мы скоро обновим, чтобы они вызывали свойства нового `QuizViewModel`, которые вы добавили. Они находятся именно в `MainActivity`, чтобы она была более организованной.

Затем добавим лениво инициализированное свойство, чтобы хранить экземпляры `QuizViewModel`, связанные с activity.

Листинг 4.8. Ленивая инициализация QuizViewModel (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
    private val quizViewModel: QuizViewModel by
lazy {
        ViewModelProviders.of(this).get(QuizVie
wModel::class.java)
    }

     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
            ~~val provider: ViewModelProvider =~~
~~ViewModelProviders.of(this)~~
                        ~~val quizViewModel =~~
~~provider.get(QuizViewModel::class.java)~~
              ~~Log.d(TAG, "Got a QuizViewModel:~~
~~$quizViewModel")~~
        ...
    }
    ...
}
```

Использование `lazy` допускает применение свойства `quizViewModel` как `val`, а не `var`. Это здорово, потому что вам нужно захватить и сохранить `QuizViewModel`, лишь когда создается экземпляр activity, поэтому `quizViewModel` получает значение только один раз.

Что еще более важно, использование `lazy` означает, что расчет и назначение `quizViewModel` не будет происходить, пока вы не запросите доступ к `quizViewModel` впервые. Это

хорошо, потому что вы не можете безопасно получить доступ к ViewModel до выполнения Activity.onCreate(...). Если вы пытаетесь вызвать ViewModelProviders.of(this).get(QuizViewModel::class.java) до Activity.onCreate(...), ваше приложение вылетит с исключением IllegalStateException.

В заключение обновим MainActivity для отображения содержимого и взаимодействия с недавно обновленным QuizViewModel.

Листинг 4.9. Обновление вопроса через QuizViewModel (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        nextButton.setOnClickListener {
                ~~currentIndex = (currentIndex +~~
~~1) % questionBank.size~~
            quizViewModel.moveToNext()
            updateQuestion()
        }
        ...
    }
    ...
    private fun updateQuestion() {
                    ~~val questionTextResId =~~
~~questionBank[currentIndex].textResId~~
                    val questionTextResId =
quizViewModel.currentQuestionText
```

```
        questionTextView.setText(questionTextRe
sId)
    }

         private fun checkAnswer(userAnswer:
Boolean) {
                            ~~val correctAnswer =~~
~~questionBank[currentIndex].answer~~
                           val correctAnswer =
quizViewModel.currentQuestionAnswer
        ...
}
```

Запустите GeoQuiz, нажмите **NEXT**, а затем поверните устройство или эмулятор. Независимо от того, сколько раз вы выполните поворот, новый экземпляр `MainActivity` будет «помнить», на каком вопросе вы остановились. Можете порадоваться, ведь вы решили проблему потери прогресса при повороте.

Но не слишком сильно радуйтесь. В программе есть еще одна незаметная ошибка.

Сохранение данных после завершения процесса

Конфигурация изменяется не только тогда, когда операционная система может уничтожить activity, даже если пользователь не собирался этого делать. Каждое приложение получает свой собственный процесс (а именно процесс Linux), содержащий один поток для работы с интерфейсом и память для хранения объектов.

Процесс приложения может быть завершен ОС, если пользователь долго не пользуется приложением, и Android

нужно освободить оперативную память. Когда процесс приложения уничтожается, все объекты, хранящиеся в памяти этого процесса, тоже уничтожаются. (Подробнее о процессах приложений вы узнаете из главы 23.)

Процессы, содержащие выполняющиеся или приостановленные activity, имеют более высокий приоритет по сравнению с другими процессами. Когда ОС нужно высвободить ресурсы, она сначала будет завершать процессы с низким приоритетом. На практике это означает, что процесс, содержащий видимую activity, уничтожаться не будет. Если это произошло — значит, с устройством что-то не так (и завершение приложения в этом случае — меньшая из проблем).

Но вот остановленные activity вполне можно уничтожить. Так, например, если пользователь нажимает кнопку «Главный экран», после чего смотрит видео или играет в игру, процесс вашего приложения может быть уничтожен.

(На момент написания отдельно взятая activity не уничтожается при недостатке памяти, даже если в документации говорится, что это так. Вместо этого Android удаляет из памяти весь процесс приложения, а заодно и все activity.)

Когда операционная система уничтожает процесс в приложении, все activity приложения и `ViewModel`-и будут удалены. И ОС насчет аккуратного завершения церемониться не будет и вызывать никакие activity и функции *обратного вызова жизненного цикла*`ViewModel` тоже не станет.

Итак, как можно сохранить состояние данных пользовательского интерфейса и использовать их для восстановления activity, чтобы пользователь вообще не замечал, что activity была уничтожена? Один из способов сделать это — хранить данные в виде *сохраненного состояния экземпляра*. Сохраненное состояние экземпляра — это данные,

которые ОС временно сохраняет за пределами activity. Вы можете добавить значения в сохраненное состояние экземпляра путем переопределения `Activity.onSaveInstanceState(Bundle)`.

ОС вызывает `Activity.onSaveInstanceState(Bundle)` всякий раз, когда незавершенная activity переходит в остановленное состояние (например, когда пользователь нажимает кнопку «Главный экран», а затем запускает другое приложение). Это время очень важно, потому что остановленные activity отмечаются как *killable*. Если процесс приложения будет завершен из-за низкого приоритета, вы можете быть уверены, что функция `Activity.onSaveInstanceState(Bundle)` уже была вызвана.

Реализация по умолчанию функции `onSaveInstanceState(Bundle)` сохраняет все представления и состояние activity в объекте `Bundle`. Это такая структура, которая отображает строковые ключи к значениям определенных типов.

Вы уже видели объект `Bundle` ранее. Он передается в функцию `OnCreate(Bundle?)`:

```
override fun onCreate(savedInstanceState:
Bundle?) {
    super.onCreate(savedInstanceState)
    ...
}
```

При переопределении функции `onCreate(Bundle?)` вы вызываете функцию `OnCreate(Bundle?)` на суперклассе своей activity и передаете набор, который только что получили. В реализации суперкласса сохраненное состояние представлений

извлекается и используется для повторного создания иерархии видов activity.

Переопределение onSaveInstanceState(Bundle)

Вы можете переопределить `onSaveInstanceState(Bundle)` для сохранения в набор дополнительных данных, которые затем могут быть считаны обратно в `OnCreate (Bundle?)`. Именно так можно сохранить значение `currentIndex` после завершения процесса.

Во-первых, в файл `MainActivity.kt` нужно добавить константу, которая будет ключом для пары «ключ — значение», которые будут храниться в наборе.

Листинг 4.10. Добавление ключа для значения (MainActivity.kt)

```
private const val TAG = "MainActivity"
private const val KEY_INDEX = "index"

class MainActivity : AppCompatActivity() {
    ...
}
```

Затем переопределите `onSaveInstanceState(Bundle)`, чтобы записать значение `currentIndex` в пакет с помощью константы как ключ.

Листинг 4.11. Переопределение onSaveInstanceState(...) (MainActivity.kt)

```
override fun onPause() {
    ...
}
```

```
override fun onSaveInstanceState(savedInstanceState: Bundle)
{
    super.onSaveInstanceState(savedInstanceStat
e)
    Log.i(TAG, "onSaveInstanceState")
            savedInstanceState.putInt(KEY_INDEX,
quizViewModel.currentIndex)
}

override fun onStop() {
    ...
}
```

Наконец, проверим это значение в `OnCreate(Bundle?)`. Если оно существует, назначьте его `currentIndex`. Если значение с ключом `index` в наборе не существует или если набор пустой, присвойте значение `0`.

Листинг 4.12. Проверка наборов в OnCreate(Bundle?) (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d(TAG, "onCreate(Bundle?) called")
    setContentView(R.layout.activity_main)

                    val currentIndex =
savedInstanceState?.getInt(KEY_INDEX, 0) ?: 0
    quizViewModel.currentIndex = currentIndex
```

```
        ...
    }
```

`OnCreate` принимает в качестве входных данных обнуляемый набор. Это связано с тем, что не существует состояния, когда новый экземпляр activity запускается пользователем в первый раз, так что в этом случае набор будет пустым. Когда activity пересоздается после поворота или завершения процесса, набор будет ненулевым. Он будет содержать пары «ключ — значение», добавляемые в `onSaveInstanceState(Bundle)`. Набор может также содержать дополнительную информацию, добавленную фреймворком, например содержание `EditText` или другого виджета пользовательского интерфейса.

Реакцию на поворот проверить легко. И, к счастью, с низким уровнем памяти все так же просто. Попробуйте — и убедитесь сами.

На устройстве или эмуляторе нажмите на значок настроек в списке приложений. Вам нужно получить доступ к опциям разработчиков, которые по умолчанию скрыты. Если вы используете аппаратное устройство, вам потребуется включить опции разработчика, как описано в главе 2. Если вы используете эмулятор (или еще не включили режим для разработчиков), выберите команду/экран **System⇒Aboutemulateddevice** (или **System⇒AboutTablet/Phone**). Прокрутите вниз и нажмите/коснитесь **Buildnumber** семь раз подряд.

Когда появится сообщение You are now a developer!, нажмите кнопку «Назад», чтобы вернуться в настройки системы. Прокрутите вниз и найдите раздел **Developeroptions** (возможно, потребуется развернуть раздел **Advanced**). На экране **Developeroptions** вы увидите множество возможных настроек.

Прокрутите вниз до раздела **Apps** и установите переключатель **Don'tkeepactivities** в активное положение, как показано на рис. 4.8.

Теперь запустите приложение GeoQuiz, нажмите кнопку **NEXT**, чтобы перейти к следующему вопросу, и кнопку «Главный экран». Нажатие кнопки «Главный экран», как вы уже знаете, приведет к остановке activity. Журналы говорят, что остановленная activity была завершена, так же как если бы ОС Android удалила ее из памяти. Однако журналы также говорят, что была вызвана функция `onSaveInstanceState(Bundle)`, — поэтому надежда есть.

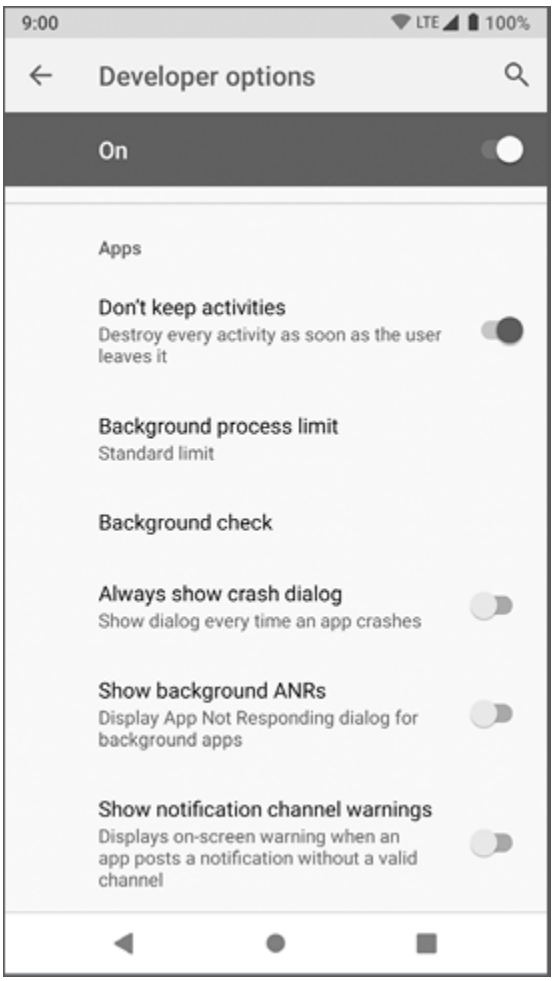


Рис. 4.8. Опция Don't keep activities

Восстановите приложение (используя список приложений на устройстве или эмуляторе), чтобы проверить, сохранилось ли состояние так, как вы ожидали. Поздравьте себя, если приложение GeoQuiz открылось на том вопросе, который вы видели в последний раз.

Обязательно выключите опцию **Don'tkeepactivities**, когда закончите тестирование, так как иначе она приведет к снижению производительности. Помните, что нажатие кнопки «Назад» вместо кнопки «Главный экран» всегда уничтожает activity, независимо от того, включена ли эта опция. Нажатие кнопки «Назад» сообщает ОС, что пользователь закончил работу с activity.

Сохраненное состояние экземпляра и записи activity

Как данные, которые вы сохраняете в `onSaveInstanceState(Bundle)`, сохраняются после уничтожения activity (и процесса)? Когда вызывается `onSave-InstanceState(Bundle)`, данные сохраняются в объекте `Bundle`. Этот объект `Bundle` затем сохраняется в *записи activity* в ОС.

Чтобы понимать, как устроена запись activity, давайте добавим состояние *stashed* (сохранена) в жизненный цикл activity (рис. 4.9).

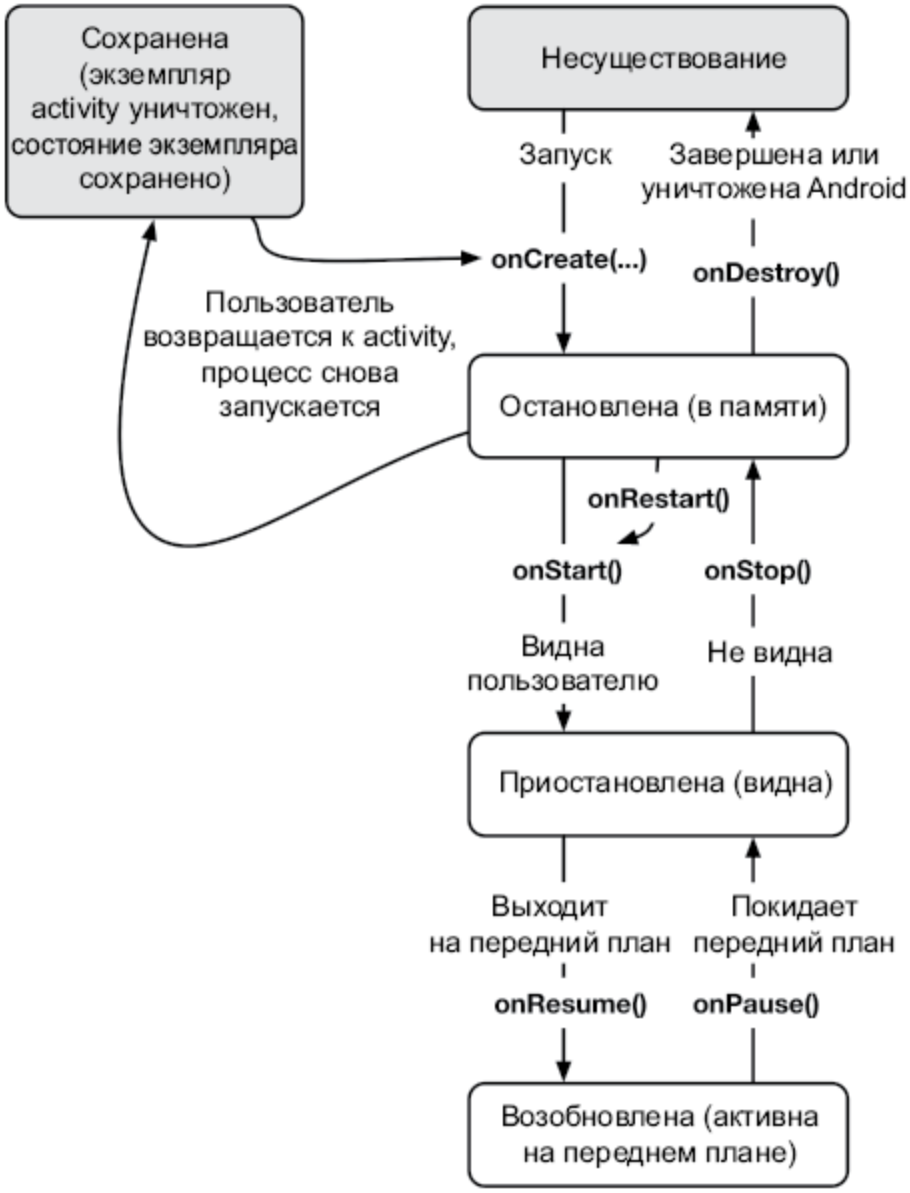


Рис. 4.9. Полный жизненный цикл activity

Когда activity сохранена, сам объект `Activity` не существует, но запись об activity сохранена в ОС. ОС может восстановить activity с помощью этой записи, если необходимо.

Обратите внимание, что ваша activity может перейти в сохраненное состояние без вызова функции `onDestroy()`. Вы можете быть уверены, что произойдет вызов `OnStop()` и `onSaveInstanceState(Bundle)` (если что-то на устройстве

пошло не по плану). Как правило, вы переопределяете функцию `onSaveInstanceState(Bundle)`, чтобы она собирала небольшие переходные данные, относящиеся к текущей activity в вашем наборе. Переопределение `OnStop()` позволяет сохранить постоянные данные, такие как настройки пользователя, так как ваша activity может быть завершена в любой момент после завершения этой функции.

Когда эти записи удаляются? После завершения activity они уничтожаются раз и навсегда. В этот момент ваша запись activity удаляется. Также запись удаляется при перезагрузке (вспомните, что подразумевается под завершением activity, повторив главу 3).

Сравнение ViewModel и сохраненного состояния экземпляра

В сохраненном состоянии экземпляра запись activity хранится после уничтожения или изменения конфигурации. При первом запуске activity сохраненное состояние экземпляра нулевое. При повороте устройства операционная система вызывает функцию `onSaveInstanceState(Bundle)` от вашей activity. Затем ОС передает данные, которые были сохранены в наборе, функции `OnCreate(Bundle?)`.

Если сохраненное состояние экземпляра защищает и от изменений конфигурации и от уничтожения процесса, зачем вообще нужен `ViewModel`? Приложение GeoQuiz настолько простое, что можно было бы обойтись одним сохраненным состоянием экземпляра.

Однако в большинстве приложений нельзя ограничиться небольшим, жестко заданным набором данных, как в GeoQuiz. Большинство приложений динамически используют данные из базы данных, из интернета или из двух мест одновременно. Эти операции являются асинхронными, часто бывают

медленными и тратят заряд аккумулятора и трафик. Связывать эти операции с жизненным циклом activity сложно и чревато ошибками.

`ViewModel` проявляет себя во всей красе, когда вы используете его, чтобы организовать динамическую работу с данными в главах 11 и 24. Например, `ViewModel` позволяет возобновить операцию загрузки после изменения конфигурации. В нем также есть простой способ сохранить данные, которые не получалось сохранить в памяти после изменения конфигурации. И, как вы уже видели, `ViewModel` автоматически очищается, когда пользователь завершает activity.

`ViewModel` не так хорошо работает при уничтожении процесса, так как он удаляется из памяти вместе с процессом. Здесь уже на сцену выходит сохраненное состояние экземпляра. Но и у него есть свои ограничения. Поскольку сохраненное состояние экземпляра сериализуется на диск, не стоит сохранять крупные или сложные объекты.

На момент написания этой главы команда Android активно работала над улучшением опыта разработчиков при работе с `ViewModel`. Появилась новая библиотека `lifecycle-viewmodel-savedstate`, выпущенная с целью позволить `ViewModel` сохранять свое состояние после завершения процесса. Это должно облегчить некоторые из трудностей использования `ViewModel`.

Вопрос, что лучше, `ViewModel` или сохраненное состояние экземпляра, вообще не стоит. Грамотные разработчики используют оба эти инструмента, в зависимости от ситуации.

Используйте сохраненное состояние экземпляра, чтобы сохранить минимальное количество информации, необходимой для воссоздания состояния пользовательского интерфейса (например, номер текущего вопроса). Используйте

ViewModel, чтобы кэшировать множество данных, необходимых для восстановления интерфейса после изменения конфигурации. Когда activity воссоздается после смерти процесса, используйте информацию о сохраненном состоянии экземпляра, чтобы настроить ViewModel, как будто ViewModel и activity не были уничтожены.

На момент написания книги не существует простого способа определить, была ли activity воссоздана после уничтожения процесса или изменения конфигурации. Почему это важно? ViewModel остается в памяти во время изменения конфигурации. Поэтому если вы используете сохраненное состояние экземпляра данных для обновления ViewModel после изменения конфигурации, приложение делает лишнюю работу, которая заставляет пользователя ждать или тратить лишние ресурсы.

Один из способов решить эту проблему — сделать ваш ViewModel немного умнее. Если установка значения ViewModel может привести к лишней работе, сначала проверьте, достаточно ли актуальны данные, прежде чем выполнять работу по их извлечению:

```
class SomeFancyViewModel : ViewModel() {
    ...
    fun setCurrentIndex(index: Int) {
        if (index != currentIndex) {
            currentIndex = index
            // Загрузка текущего вопроса из БД
        }
    }
}
```

Ни `ViewModel`, ни сохраненное состояние экземпляра не подходят для длительного хранения. Если вам нужно хранить данные в течение длительного срока, пока приложение установлено на устройстве, независимо от состояния вашей activity, нужно более постоянное решение. Вы узнаете о двух локальных вариантах постоянного хранения данных: базы данных из главы 11 и общие настройки из главы 26. Общие настройки отлично подходят для необъемных и простых данных. Локальная база данных — лучший вариант для более крупных и сложных данных. Помимо локальной памяти, можно хранить данные на удаленном сервере. Вы узнаете, как получить доступ к данным веб-сервера, из главы 24.

Если в GeoQuiz много вопросов, возможно, имеет смысл хранить их в базе данных или на веб-сервере, а не жестко прописывать в `ViewModel`. Поскольку вопросы являются константами, имеет смысл хранить их независимо от состояния жизненного цикла activity. Однако доступ к базам данных — это относительно медленная операция по сравнению с доступом к значениям в памяти. Так что имеет смысл загрузить то, что вам нужно для отображения пользовательского интерфейса, и сохранить это в памяти, а интерфейс выводить через `ViewModel`.

В этой главе мы исправили ошибки потери состояния у приложения GeoQuiz путем правильного учета изменений конфигурации и уничтожения процесса. В следующей главе вы узнаете, как использовать инструменты отладки Android Studio, чтобы устранить другие ошибки, которые также имеют место быть.

Для любознательных: компоненты Jetpack, AndroidX и компоненты архитектуры

Библиотека `lifecycle-extensions`, в которой находится `ViewModel`, входит в состав набора компонентов Android Jetpack Components, называемого для краткости Jetpack. Это набор библиотек, созданных Google, чтобы упростить различные аспекты разработки на Android. Полный список Jetpack-библиотек можно посмотреть по ссылке **developer.android.com/jetpack**. Вы можете включить любую из них в проект, добавив соответствующую зависимость в файл `build.gradle`, как уже делали в этой главе.

Каждая из Jetpack-библиотек находится в пакете, который начинается с `androidx`. По этой причине термины «AndroidX» и «Jetpack» используются как синонимы.

Вы, возможно, помните, что в мастере создания проекта (рис. 4.10) есть флажок **UseAndroidXartifacts**. Почти всегда эту опцию нужно включать, как вы делали для GeoQuiz. Это позволит добавить в ваш проект некоторые начальные Jetpack-библиотеки и установить приложение, чтобы использовать их по умолчанию.

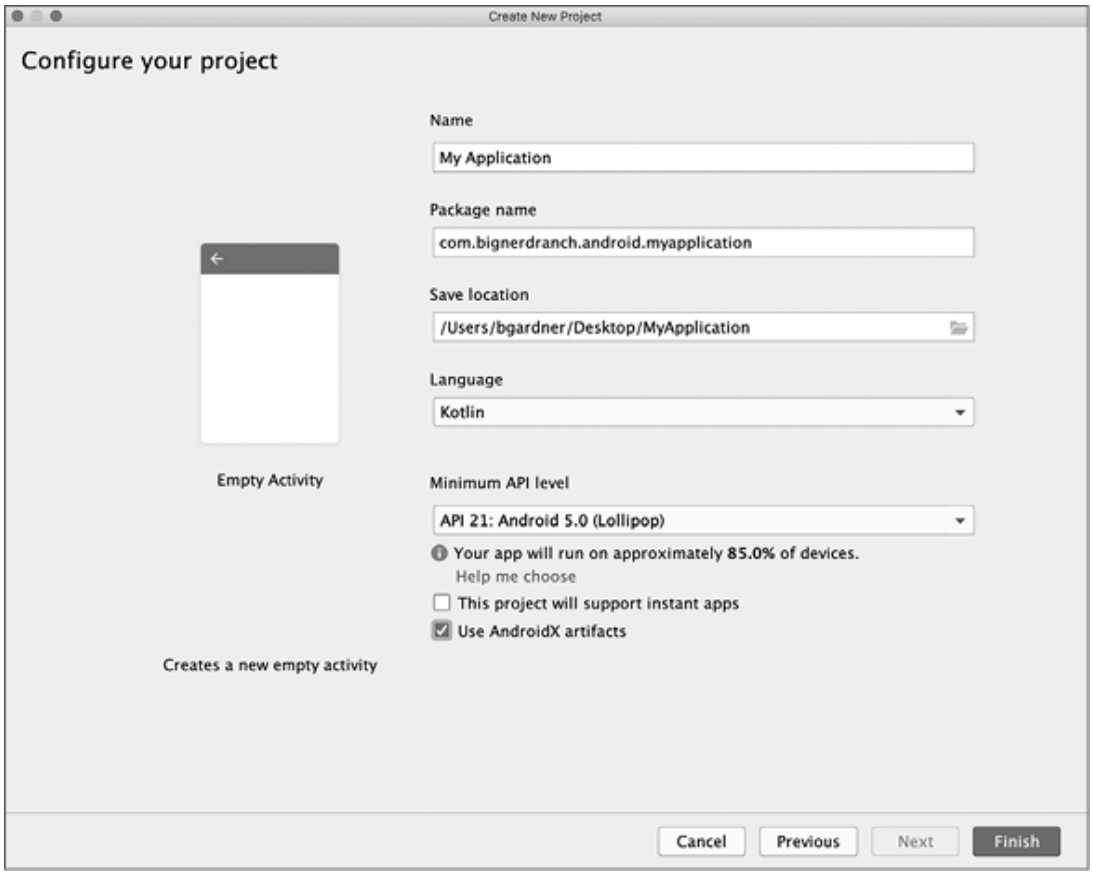


Рис. 4.10. Добавление Jetpack-библиотек

Jetpack-библиотеки разбиты на четыре категории: фундамент, архитектура, поведение и пользовательский интерфейс. Библиотеки категории архитектуры часто упоминаются как *компоненты архитектуры*. `ViewModel` — один из таких компонентов. Вы узнаете о других основных компонентах архитектуры позже из этой книги, а именно о компоненте Room (глава 11), Data Binding (глава 19), и WorkManager (глава 27).

Вы также узнаете о некоторых фундаментных библиотеках, в том числе AppCompat (глава 14), Test (глава 20) и Android KTX (глава 26). В главе 27 поработаете с поведенческой библиотекой

Notifications. Вы также будете использовать некоторые библиотеки интерфейсов Jetpack, включая Fragment (глава 8) и Layouts (глава 9 и глава 10).

Некоторые компоненты Jetpack совершенно новые. Некоторые созданы уже давно, но раньше были сосредоточены в наборе гораздо более крупных библиотек под названием Support Library. Теперь вместо всего этого можно использовать Jetpack (AndroidX).

Для любознательных: как избежать костылей

Некоторые разработчики пытаются решить потерю состояния пользовательского интерфейса при изменении конфигурации, попросту отключив поворот. Ведь если пользователь не может повернуть приложение, то и состояние пользовательского интерфейса он не потеряет, не так ли? Это правда, но, к сожалению, этот подход открывает дорогу другим ошибкам. Возникнут другие проблемы жизненного цикла, с которыми пользователи наверняка столкнутся, и при этом эти ошибки могут остаться незамеченными во время разработки и тестирования.

Кроме того, существуют и другие изменения конфигурации, которые могут возникнуть во время выполнения, например изменение размеров окна или переход на ночной режим. И да, вы можете также захватить и игнорировать или обрабатывать эти изменения. Но отключать функцию системы, которая автоматически выбирает правильные ресурсы, нехорошо.

К тому же обработка изменения конфигурации или отключение вращения не решают проблему уничтожения процесса.

Если вы хотите заблокировать приложение в книжном или альбомном режиме, *потому что такова идея приложения*, все

равно придется защитить программу от изменений конфигурации и уничтожения процесса. В этом вам помогут новые знания о `ViewModel` и сохраненном состояния экземпляра.

Запрет на изменения конфигурации с целью уберечься от потери интерфейса — это плохая идея. Упомянули мы этот способ только для того, чтобы вы могли его распознать, если встретите.

5. Отладка Android-приложений

Из этой главы вы узнаете, что делать, если в приложение закралась ошибка. В частности, вы научитесь пользоваться панелью **LogCat**, Android Lint и отладчиком среды Android Studio.

Чтобы потренироваться в починке, нужно сначала что-нибудь сломать. В файле `MainActivity.kt` закомментируйте строку кода `onCreate(Bundle?)`, в которой мы получаем `questionTextView` по идентификатору.

Листинг 5.1. Из программы исключается важная строка (MainActivity.kt)

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    trueButton = findViewById(R.id.true_button)
                                falseButton = findViewById(R.id.false_button)
    nextButton = findViewById(R.id.next_button)
                //        questionTextView = findViewById(R.id.question_text_view)
    ...
}
```

Запустите GeoQuiz и посмотрите, что получится. Приложение мгновенно сломается.

На версиях, предшествующих Android Pie (API 28), появляется сообщение об ошибке, говорящее о том, что приложение сломалось. На устройстве, работающем под

Android Pie, вы увидите, что приложение на короткий промежуток времени появляется, а затем молча останавливается. В этом случае запустите приложение заново, нажав на значок GeoQuiz на начальном экране. На этот раз, когда приложение выйдет из строя, вы увидите сообщение, подобное тому, которое показано на рис. 5.1.

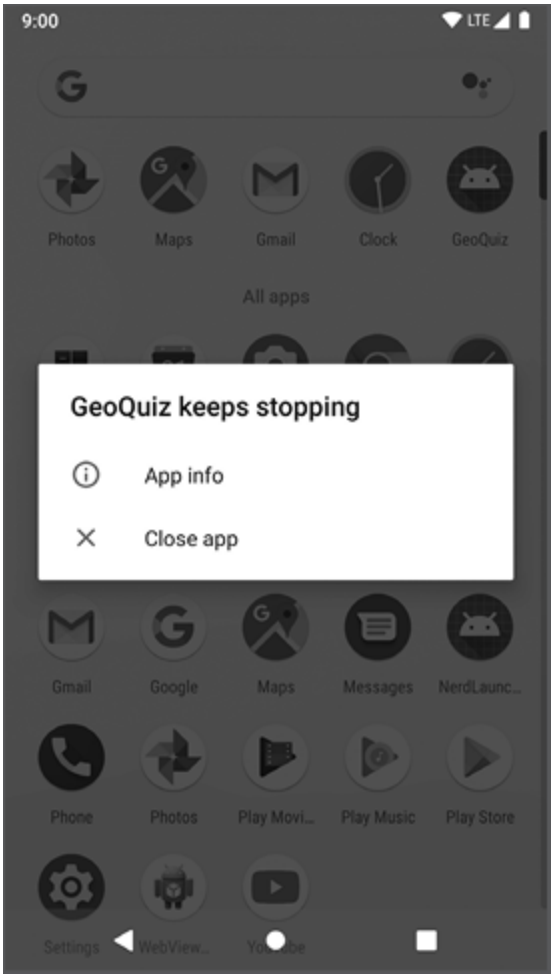


Рис. 5.1. Сбой в приложении GeoQuiz

Конечно, вы и так знаете, что случилось с приложением, но если бы не знали, было бы полезно взглянуть на приложение в другой перспективе.

Исключения и трассировка стека

Откройте панель **Logcat**, чтобы понять, что же произошло. Прокрутите содержимое **LogCat** и найдите текст, выделенный красным шрифтом (рис. 5.2). Это стандартный отчет об исключениях `AndroidRuntime`.

Если информация об исключении отсутствует в **LogCat**, возможно, необходимо изменить фильтры **LogCat**: выберите в раскрывающемся списке фильтров вариант **NoFilters**. С другой стороны, если данных в **LogCat** слишком много, также можно выбрать в списке **LogLevel** значение **Error**, при котором отображаются только наиболее критичные сообщения. Также поищите строку **fatalexception**, которая приведет вас прямо к исключению, вызвавшему сбой приложения.

В отчете приводится исключение верхнего уровня и данные трассировки стека; затем исключение, которое привело к этому исключению, и его трассировка стека; и так далее, пока не будет найдено исключение, не имеющее причины.

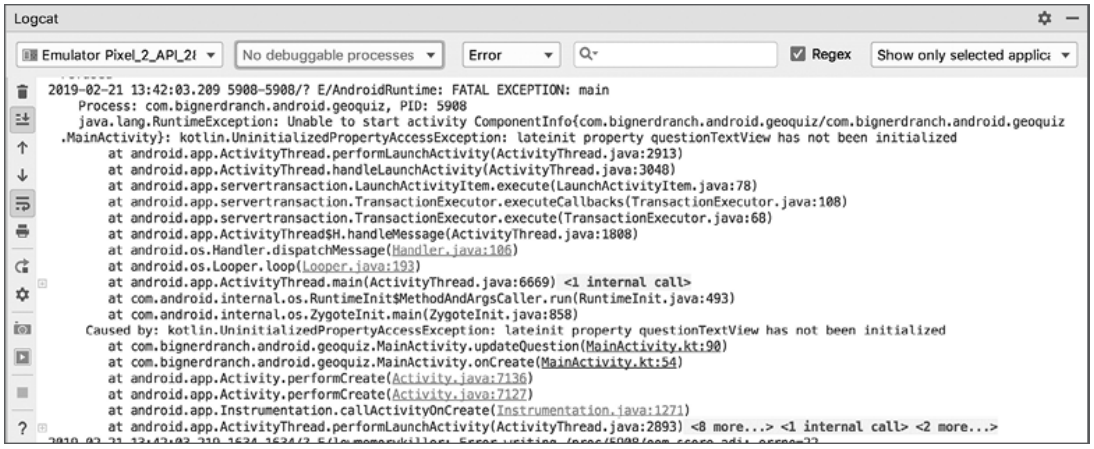


Рис. 5.2. Исключения и трассировка стека в LogCat

Может показаться странным видеть исключение `java.lang`, так как вы пишете код на Kotlin. При сборке для Android код Kotlin компилируется в тот же низкоуровневый

байткод, в который компилируется Java. В ходе этого процесса многие исключения Kotlin отображаются в классы исключений java.lang из-за механизма сглаживания типов. kotlin.RuntimeException — это суперкласс kotlin.UninitializedPropertyAccessException, который при запуске на Android получает псевдоним java.lang.RuntimeException.

Как правило, в написанном вами коде интерес представляет именно последнее исключение. В данном примере это исключение kotlin.UninitializedPropertyAccess-Exception. Строка непосредственно под именем исключения содержит начало трассировки стека. В ней указываются класс и функция, в которой произошло исключение, а также имя файла и номер строки кода. Щелкните мышью по синей ссылке; Android Studio открывает указанную строку кода.

В открывшейся строке программа впервые обращается к переменной questionTextView в функции updateQuestion(). Имя исключения UninitializedPropertyAccessException подсказывает суть проблемы: переменная не была инициализирована.

Раскомментируйте строку с инициализацией questionTextView, чтобы исправить ошибку.

Листинг 5.2. Раскомментирование важной строки (MainActivity.kt)

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    trueButton = findViewById(R.id.true_button)
                                falseButton = findViewById(R.id.false_button)
```

```
    nextButton = findViewById(R.id.next_button)
                    //questionTextView    =
findViewById(R.id.question_text_view)
    ...
}
```

Когда в вашей программе возникают исключения времени выполнения, следует искать последнее исключение на панели **LogCat** и первую строку трассировки стека со ссылкой на написанный вами код. Именно здесь возникает проблема, и именно здесь следует искать ответы.

Даже если сбой происходит на неподключенном устройстве, не все потеряно. Устройство сохраняет последние строки, выводимые в журнал. Длина и срок хранения журнала зависят от устройства, но обычно можно рассчитывать на то, что результаты будут храниться минимум 10 минут. Подключите устройство и выберите его на панели **Devices**. Панель **LogCat** заполняется данными из сохраненного журнала.

Диагностика ошибок поведения

Проблемы с приложениями не всегда приводят к сбоям — в некоторых случаях приложение просто начинает некорректно работать. Допустим, пользователь нажимает кнопку **NEXT**, а в приложении ничего не происходит. Такие ошибки относятся к категории ошибок поведения.

В файле `MainActivity.kt` внесите изменение в слушателя `nextButton` и закомментируйте код, увеличивающий индекс вопроса.

Листинг 5.3. Из программы исключается важная строка (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    ...
    nextButton.setOnClickListener {
        //    quizViewModel.moveToNext()
        updateQuestion()
    }
    ...
}
```

Запустите GeoQuiz и нажмите кнопку **NEXT**. Ничего не произойдет.

Эта ошибка коварнее предыдущей. Она не приводит к выдаче исключения, поэтому исправление ошибки не сводится к простому устранению исключения. Кроме того, некорректное поведение может быть вызвано разными причинами: то ли в программе не изменяется индекс, то ли не вызывается функция `updateQuestion()`.

Если бы вы не ввели эту ошибку сами, было бы необходимо выяснить, откуда она взялась. Далее мы покажем два основных приема диагностики: сохранение трассировки стека и использование отладчика для назначения точки останова.

Сохранение трассировки стека

Включите команду сохранения отладочного вывода в функцию `updateQuestion()` класса `MainActivity`:

Листинг 5.4. Использование Exception (MainActivity.kt)

```
private fun updateQuestion() {
        Log.d(TAG, "Updating question text",
Exception())
```

```
                val questionTextResId =
quizViewModel.currentQuestionText
    questionTextView.setText(questionTextResId)
}
```

Версия `Log.d` с сигнатурой `Log.d(String,String,Throwable)` регистрирует в журнале все данные трассировки стека, как уже встречавшееся ранее исключение `UninitializedPropertyAccessException`. По данным трассировки стека вы сможете определить, в какой момент произошел вызов функции `updateQuestion()`.

При вызове `Log.d(String,String,Throwable)` вовсе не обязательно передавать перехваченное исключение. Вы можете создать новый экземпляр `Exception` и передать его функции без инициирования исключения. В журнале будет сохранена информация о том, где было создано исключение.

Запустите приложение GeoQuiz, нажмите кнопку **NEXT** и проверьте вывод на панели **LogCat** (рис. 5.3).

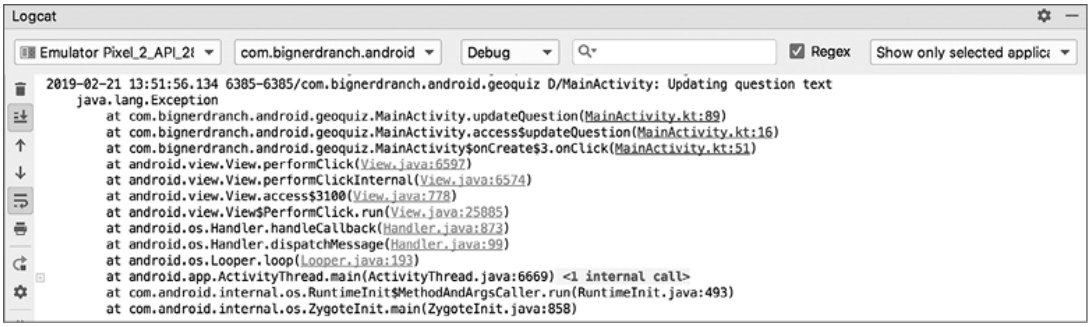


Рис. 5.3. Результаты

Верхняя строка трассировки стека соответствует строке, в которой в журнале были зарегистрированы данные `Exception`. Следующая строка показывает, где был вызван `updateQuestion()`, — в реализации `onClick(View)`. Щелкните мышью по ссылке в этой строке; откроется позиция,

в которой была закомментирована строка с увеличением индекса заголовка. Но пока не торопитесь исправлять ошибку; сейчас мы найдем ее повторно при помощи отладчика.

Регистрация в журнале данных трассировки стека — мощный инструмент отладки, но он выводит довольно большой объем данных. Оставьте в программе несколько таких команд, и вскоре вывод на панели **LogCat** превратится в невразумительную мешанину. Кроме того, по трассировке стека конкуренты могут разобраться, как работает ваш код, и похитить ваши идеи.

С другой стороны, в некоторых ситуациях нужна именно трассировка стека с информацией, показывающей, что делает ваш код. Если вы обратитесь за помощью на сайты **stackoverflow.com** и **forums.bignerdranch.com**, включите в вопрос трассировку стека. Информацию можно скопировать прямо из **LogCat**.

Прежде чем продолжать, удалите команду сохранения отладочного вывода из `MainActivity.kt`.

Листинг 5.5. Прощай, старый друг (MainActivity.kt)

```
private fun updateQuestion() {
        ~~Log.d(TAG, "Updating question text", Exception())~~
        val questionTextResId = quizViewModel.currentQuestionText
    questionTextView.setText(questionTextResId)
}
```

Установка точек останова

Попробуем найти ту же ошибку при помощи отладчика среды Android Studio. Мы установим *точку останова* в `updateQuestion()`, чтобы увидеть, была ли вызвана эта функция. Точка останова останавливает выполнение программы в заданной позиции, чтобы вы могли в пошаговом режиме проверить, что происходит далее.

В файле `MainActivity.kt` вернитесь к функции `updateQuestion()`. В первой строке функции щелкните мышью по серой полосе слева от кода. На месте щелчка появляется красный кружок; он обозначает точку останова (рис. 5.4). Также точки останова можно установить с помощью комбинации клавиш **⌘+F8** (**Ctrl+F8**).

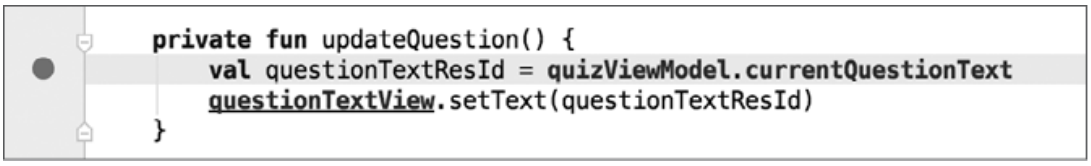


Рис. 5.4. Точка останова

Чтобы задействовать отладчик и активизировать точку останова, необходимо запустить приложение в отладочном режиме (в отличие от обычного запуска). Для этого нажмите кнопку отладки (кнопка с зеленым жуком) рядом с кнопкой выполнения. Также можно выполнить команду меню **Run⇒Debug'app'**. Устройство сообщит, что оно ожидает подключения отладчика, а затем продолжит работу как обычно.

Иногда вам может потребоваться отладить запущенное приложение без его перезапуска. Вы можете прикрепить отладчик к работающему приложению, нажав кнопку **AttachDebuggertoAndroidProcess**, показанную на рис. 5.5, или выбрав команду **Run⇒Attachtoprocess**. Выберите процесс вашего приложения в появившемся диалоговом окне и нажмите **OK**, после чего отладчик прикрепится. Обратите внимание, что

точки останова будут работать только тогда, когда подключен отладчик, поэтому любые точки останова, которые будут удалены до того, как вы подключите отладчик, будут проигнорированы.

Отладка app

Прикрепить отладчик к работающему приложению

Рис. 5.5. Кнопки отладки

Нам нужно отлаживать GeoQuiz с самого начала кода, поэтому мы использовали опцию **Debug'app'**. Вскоре после того как ваше приложение будет запущено с прикрепленным отладчиком, оно остановится. Запуск GeoQuiz вызывает `MainActivity.onCreate(Bundle?)`, которая вызвала функцию `updateQuestion()`, активирующую точку останова. (Если бы вы прикрепили отладчик к процессу после его запуска, то приложение, скорее всего, не делало бы паузы, так как `MainActivity.onCreate(Bundle?)` выполняется до того, как вы смогли бы прикрепить отладчик.)

На рис. 5.6 видно, что файл `MainActivity.kt` теперь открыт в окне инструментов редактора и выделена строка с точкой останова, в которой выполнение приостановлено. Теперь видно окно инструмента отладки в нижней части экрана. Тут есть панели **Frames** и **Variables**. (Если окно инструмента отладки не открылось автоматически, его можно открыть, нажав на кнопку **Debug** в нижней части окна Android Studio.)

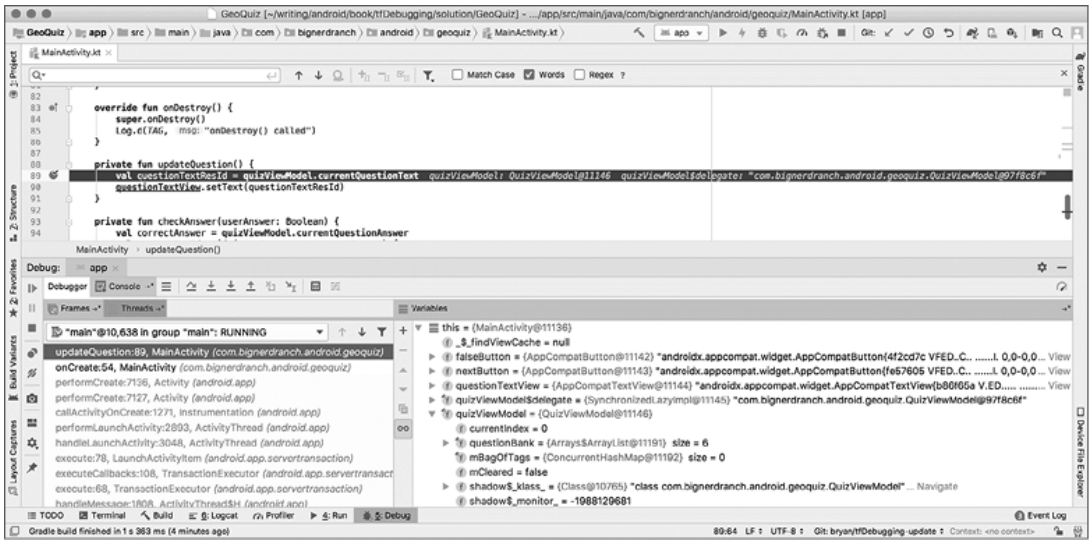


Рис. 5.6. Стоять!

Вы можете использовать кнопки со стрелками в верхней части панели **Debug** (рис. 5.7) для перемещения по вашей программе, а кнопку **EvaluateExpression** — для выполнения простых операторов Kotlin во время отладки, что тоже очень удобно.

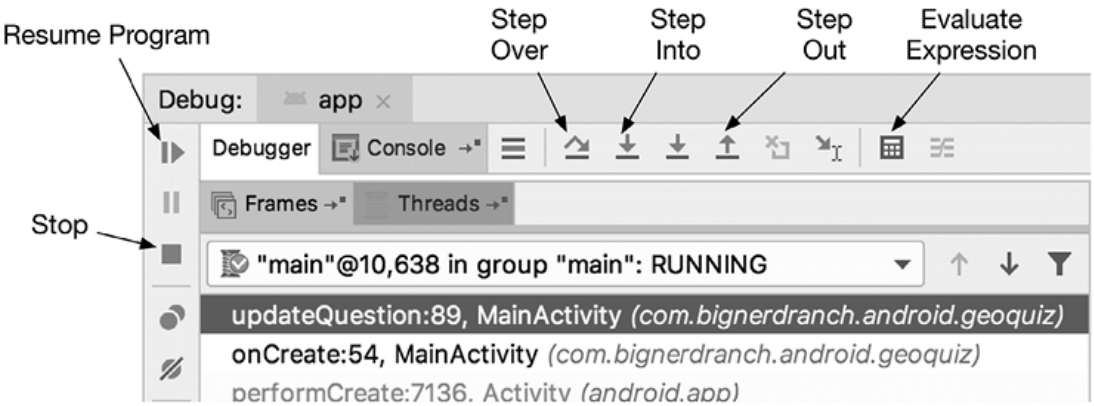


Рис. 5.7. Панель Debug

Из трассировки стека видно, что функция `updateQuestion()` была вызвана из `onCreate(Bundle?)`. Так как нас интересует поведение кнопки **NEXT**, нажмите кнопку **Resume**, чтобы продолжить выполнение программы. Затем снова нажмите кнопку **NEXT**, чтобы увидеть, активизируется ли точка останова (она должна активизироваться).

Теперь, когда мы добрались до интересного момента выполнения программы, можно немного осмотреться. Представление **Variables** используется для просмотра текущих значений объектов вашей программы. На нем должны отображаться переменные, созданные в `MainActivity`, а также еще одно значение: `this` (сам объект `MainActivity`).

Разверните переменную `this`, чтобы увидеть все переменные, объявленные в `MainActivity`, в суперклассе `MainActivity`, в суперклассе `Activity`, в суперсуперклассе и так далее. А пока посмотрим на переменные, которые вы создали.

Вас интересует только одно значение: `quizViewModel.currentIndex`. Прокрутите вниз панель переменных, пока не увидите `quizViewModel`. Разверните `quizViewModel` и найдите `currentIndex` (рис. 5.8).

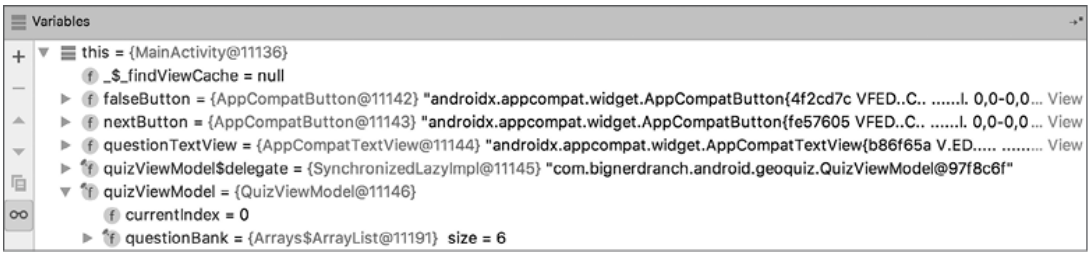


Рис. 5.8. Проверка значений переменных во время выполнения

Вы ожидали, что `currentIndex` будет иметь значение `1`. Вы нажали кнопку **NEXT**, в результате чего значение `currentIndex` должно было увеличиться с `0` до `1`. Однако, как показано на рис. 5.8, `currentIndex` все еще имеет значение `0`.

Проверьте код в окне редактора. Код в `MainActivity.updateQuestion()` просто обновляет текст вопроса на основе содержимого `QuizViewModel`. Никаких проблем. Так откуда же ошибка?

Чтобы продолжить исследование, необходимо выйти из этой функции и определить, какой код был выполнен непосредственно перед `MainActivity.updateQuestion()`. Для этого нажмите кнопку **StepOut**.

Проверьте окно редактора. Теперь вы перешли к `OnClickListener` компонента `nextButton` сразу после вызова функции `updateQuestion()`. Довольно легко.

Как вы уже знали, проблематичное поведение вызвано тем, что функция `quizViewModel.moveToNext()` ни разу не вызывалось (потому что вы ее закомментировали). Это нужно исправить, но перед тем как вносить изменения в код, вы должны прекратить отладку вашего приложения. Если вы редактируете свой код во время отладки, то код, запущенный с прикрепленным отладчиком, будет устаревшим и отладчик может заработать не совсем правильно.

Вы можете остановить отладку двумя способами: остановить программу или просто отключить отладчик. Чтобы остановить программу, нажмите кнопку **Stop**, показанную на рис. 5.7.

Верните `OnClickListener` в прежнее состояние.

Листинг 5.6. Возвращение к исходному состоянию (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    ...
    nextButton.setOnClickListener {
            // quizViewModel.moveToNext()
        updateQuestion()
    }
    ...
}
```

Мы рассмотрели два способа поиска проблемной строки кода: сохранение в журнале трассировки стека и установку точки останова в отладчике. Какой способ лучше? Каждый находит свои применения, и, скорее всего, один из них станет для вас основным.

Преимущество трассировки стека заключается в том, что трассировки из нескольких источников просматриваются в одном журнале. С другой стороны, чтобы получить новую информацию о программе, вы должны добавить новые команды регистрации, заново построить приложение, развернуть его и добраться до нужной точки. С отладчиком работать проще. Запустив приложение с подключенным отладчиком, вы сможете установить точку останова во время работы приложения, а потом поэкспериментировать для получения информации сразу о разных проблемах.

Особенности отладки Android

Как правило, процесс отладки в Android не отличается от обычной отладки кода Kotlin. Тем не менее у вас могут возникнуть проблемы в областях, специфических для Android (например, ресурсы), о которых компилятор Kotlin ничего не знает. В этом разделе мы узнаем о двух видах специальных вопросов Android: Android Lint и проблемы с классом `R`.

Android Lint

Android Lint — *статический анализатор* кода Android. Статические анализаторы проверяют код на наличие дефектов, не выполняя его. Android Lint использует свое знание инфраструктуры Android для проверки кода и выявления

проблем, которые компилятор обнаружить не может. Как правило, к рекомендациям Android Lint стоит прислушиваться.

В главе 7 вы увидите, как Android Lint выдает предупреждение о проблеме совместимости. Кроме того, Android Lint может выполнять проверку типов для объектов, определенных в XML.

Вы можете запустить Lint вручную, чтобы получить список всех потенциальных проблем в приложении (в том числе и не столь серьезных, как эта). Выполните команду меню **Analyze⇒InspectCode**. Вам будет предложено выбрать анализируемые части проекта. Выберите вариант **Wholeproject** и нажмите кнопку **OK**. Android Studio запустит Lint, а также несколько других статических анализаторов кода.

После завершения анализа выводится список потенциальных проблем, разбитый на несколько категорий. Раскройте категорию **AndroidLint**, чтобы просмотреть информацию Lint о вашем проекте (рис. 5.9).

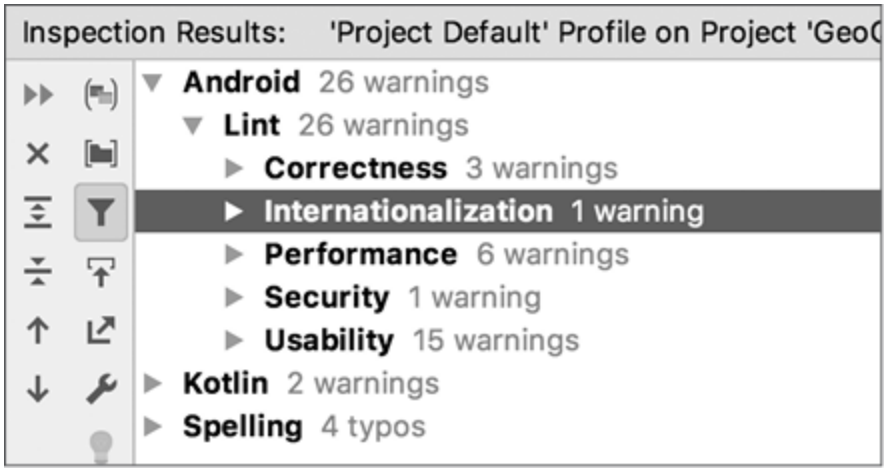


Рис. 5.9. Предупреждения Lint

(Не беспокойтесь, если увидите другое количество предупреждений. Инструментарий Android постоянно развивается, и в Lint могли быть добавлены новые проверки, а

в Android — новые ограничения. Кроме того, появляются новые версии инструментов и зависимостей.)

Разверните список **Internationalization**, а затем под ним — список **BidirectionalText**, чтобы увидеть более подробную информацию по этому вопросу в вашем проекте. Нажмите кнопку **Usingleft/rightinsteadofstart/endattributes**, чтобы узнать об этом конкретном предупреждении (рис. 5.10).

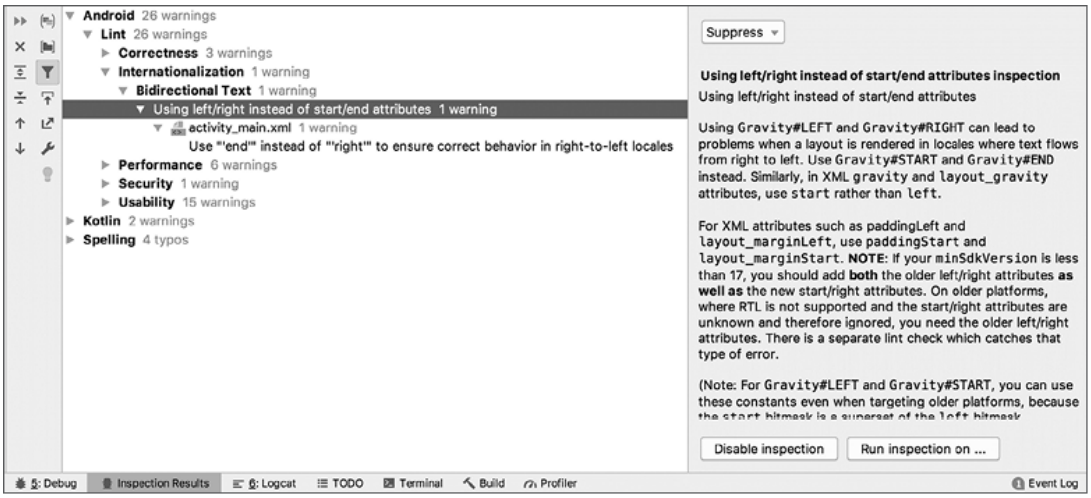


Рис. 5.10. Описание предупреждения

Lint предупреждает, что использование правого и левого значений для атрибутов макета может вызвать проблемы, если ваше приложение используется на устройстве, язык которого подразумевает чтение справа налево, а не слева направо. (О том, как подготовить приложение готовым к международному использованию, вы узнаете из главы 17.)

Далее вам нужно найти, какой файл и строка или строки кода вызвали предупреждение. Разверните пункт **Usingleft/rightinsteadofstart/endattributes**. Нажмите на проблемный файл `activity_main.xml`, чтобы увидеть неправильный фрагмент кода (рис. 5.11).

Рис. 5.11. Просмотр кода, который вызвал предупреждение

Дважды щелкните по описанию предупреждения, которое появляется под именем файла. Файл `res/layout/land/activity_main.xml` откроется в окне редактора, и курсор поместится на строку, вызывающую предупреждение:

```
<Button
        android:id="@+id/next_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
            android:layout_gravity="bottom|righ
t"
        android:text="@string/next_button"
        android:drawableEnd="@drawable/arrow_ri
ght"
        android:drawablePadding="4dp"/>
```

Строка Lint задает гравитацию кнопки **NEXT**, чтобы она появлялась в правом нижнем углу макета. Чтобы исправить предупреждение, измените значение `bottom|right` на `bottom|end`. Тогда кнопка будет расположена в нижнем левом углу, если язык устройства читается справа налево, а не слева направо.

Листинг 5.7. Предупреждение для двунаправленного текста (res/layout/land/activity_main.xml)

```
...
<Button
        android:id="@+id/next_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
            android:layout_gravity="bottom|right"
        android:layout_gravity="bottom|end"
        android:text="@string/next_button"
        android:drawableEnd="@drawable/arrow_right"
        android:drawablePadding="4dp"/>
...
```

Перезапустите Lint, чтобы подтвердить, что исправленный двунаправленный текст больше не отображается в результатах Lint. Чтобы увидеть, как макет будет выглядеть на устройствах с разными языками, откройте вкладку **Design** в окне редактора. Измените значение **Default(en-us)** в раскрывающемся списке **LocaleforPreview** на **PreviewRighttoLeft** (рис. 5.12). Если вы не видите раскрывающегося списка **LocaleforPreview**, нажмите кнопку >> в верхней части панели предварительного просмотра, чтобы развернуть ее.

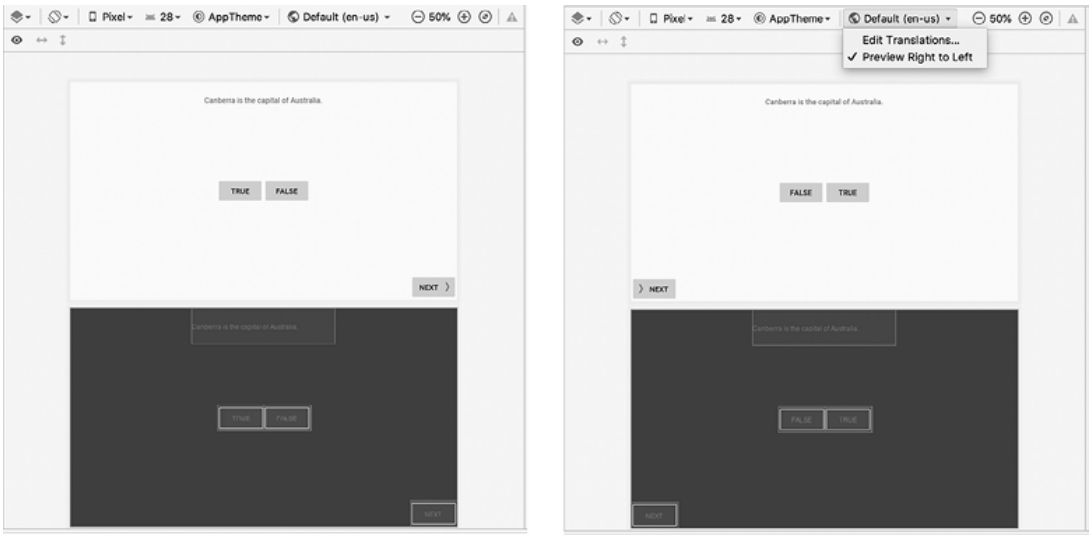


Рис. 5.12. Просмотр слева направо и справа налево

В основном ваше приложение будет работать отлично, даже если вы не исправите то, о чем предупреждает Lint. Зачастую, однако, обращение к предупреждениям Lint может помочь вам предотвратить проблемы в будущем или улучшить работу пользователей. Мы рекомендуем серьезно относиться ко всем предупреждениям Lint, даже если в результате вы решили не обращать на них внимания (как в данном случае). В противном случае вы можете привыкнуть игнорировать Lint и из-за этого пропустите серьезные проблемы.

Инструмент Lint выдает подробную информацию о каждой найденной проблеме и предлагает способы ее решения. Мы оставим это вам в качестве упражнения по решению проблем, найденных Lint в GeoQuiz. Вы можете игнорировать проблемы, исправлять их, как рекомендует Lint, или нажать кнопку **Suppress** на панели описания проблемы, чтобы отключить предупреждения в будущем. В оставшихся главах о GeoQuiz мы

будем считать, что проблемы Lint оказались нерассмотренными.

Проблемы с классом R

Всем известны ошибки сборки, которые возникают из-за ссылок на ресурсы до их добавления (или удаления ресурсов, на которые ссылаются другие файлы). Обычно повторное сохранение файла после добавления ресурса или удаления ссылки приводит к тому, что Android Studio проводит сборку заново, а проблемы исчезают.

Однако иногда такие ошибки остаются или появляются из ниоткуда. Если вы столкнетесь с подобной ситуацией, примите следующие меры.

Проверьте разметку XML в файлах ресурсов.

Если файл `R.java` не был сгенерирован при последней сборке, то все ссылки на ресурс будут сопровождаться ошибками. Часто ошибки вызваны опечатками в разметке одного из файлов XML. Разметка макета не проверяется, поэтому среда не сможет привлечь ваше внимание к опечаткам в таких файлах. Если вы найдете ошибку и заново сохраните файл, код `R.java` будет сгенерирован заново.

Выполните чистку проекта.

Выполните команду **Build⇒CleanProject**. Android Studio сформирует проект заново, а результат сборки избавится от ошибок. Глубокую чистку проекта полезно проводить время от времени.

Синхронизируйте проект с Gradle.

Если вы вносите изменения в файл `build.gradle`, эти изменения необходимо синхронизировать с настройками сборки вашего проекта. Выполните команду **File⇒SyncProjectwithGradleFiles**. Android Studio построит

проект заново с правильными настройками; при этом проблемы, связанные с изменением конфигурации Gradle, могут исчезнуть.

Запустите Android Lint.

Обратите внимание на предупреждения, полученные от Android Lint. При запуске этой программы нередко выявляются совершенно неожиданные проблемы.

Если у вас все еще остаются проблемы с ресурсами (или иные проблемы), отдохните и просмотрите сообщения об ошибках и файлы макета на свежую голову. В запале легко пропустить ошибку. Также проверьте все ошибки и предупреждения Lint. При спокойном повторном рассмотрении сообщений нередко выявляются ошибки или опечатки.

Наконец, если вы зашли в тупик или у вас возникли другие проблемы с Android Studio, обратитесь к архивам **stackoverflow.com** или посетите форум по адресу **forums.bignerdranch.com**.

Упражнение. Исследуем Layout Inspector

Для диагностики проблем с файлами макетов и интерактивного анализа визуализации макета на экране можно воспользоваться инструментом **LayoutInspector**. Убедитесь в том, что GeoQuiz выполняется в эмуляторе, и нажмите кнопку **LayoutInspector** на левой панели окна Android Monitor. Далее вы сможете исследовать свойства своего макета, щелкая на элементах в представлении **LayoutInspector**.

Упражнение. Profiler

С помощью инструмента **Profiler** создаются подробные отчеты о том, как ваше приложение использует ресурсы Android-

устройства, а именно процессор и память. Это полезно при оценке и настройке производительности вашего приложения.

Для просмотра окна **Profiler** запустите приложение на подключенном Android-устройстве или эмуляторе, в строке меню выберите команду **View⇒ToolWindows⇒Profiler**. В открывшемся окне **Profiler** отобразится временная шкала с показаниями по использованию сети, процессора, памяти и заряда аккумулятора.

Щелкните по разделу, чтобы увидеть более подробную информацию об использовании этого ресурса вашим приложением. В режиме просмотра процессора нажмите кнопку **Record**, чтобы получить более подробную информацию об использовании процессора. После выполнения любых взаимодействий с приложением, которые вы хотите записать, нажмите кнопку **Stop**, чтобы остановить запись.

6. Вторая activity

В этой главе мы добавим в приложение GeoQuiz вторую activity. Как было сказано ранее, activity управляет информацией на экране; новая activity добавит в приложение второй экран, на котором пользователю будет предложено увидеть ответ на текущий вопрос.

Если пользователь решает просмотреть ответ, а затем возвращается к `MainActivity` и отвечает на вопрос, он получает новое сообщение. Новая activity изображена на рис. 6.1.

Если пользователь решает просмотреть ответ, а затем возвращается к `MainActivity` и отвечает на вопрос, он получает новое сообщение (рис. 6.2).

Почему эта задача является хорошим упражнением по программированию Android? Потому что вы научитесь:

- создавать новую activity и новый макет;
- *запускать* activity из другой activity (вы приказываете ОС создать экземпляр activity и вызвать ее функцию `onCreate(Bundle?)`);
- передавать данные между родительской (запускающей) и дочерней (запущенной) activity.

Подготовка к внедрению второй activity

В этой главе нам предстоит многое сделать. К счастью, часть рутинной работы будет выполнена за вас мастером **NewActivity** среды Android Studio.

Но сначала откройте файл `res/values/strings.xml` и добавьте строки, необходимые для этой главы.

Рис. 6.1. CheatActivity позволяет подсмотреть ответ на вопрос

Рис. 6.2. MainActivity знает, что вы жульничаете

Листинг 6.1. Добавление строк (res/values/strings.xml)

```
<resources>
    ...
      <string name="incorrect_toast">Incorrect!
</string>
```

```
    <string name="warning_text">Are you sure you want to do this?</string>
    <string name="show_answer_button">Show Answer</string>
    <string name="cheat_button">Cheat!</string>
    <string name="judgment_toast">Cheating is wrong.</string>

</resources>
```

Создание второй activity

При создании activity обычно изменяются по крайней мере три файла: файл класса Kotlin, макет XML и манифест приложения. Но если эти файлы будут изменены некорректно, Android будет протестовать. Чтобы избежать возможных проблем, воспользуйтесь мастером **NewActivity** среды Android Studio.

Запустите мастер **NewActivity**: щелкните правой кнопкой мыши по папке `app/java` на панели **Project** и выберите команду **New⇒Activity⇒EmptyActivity** в контекстном меню (рис. 6.3).

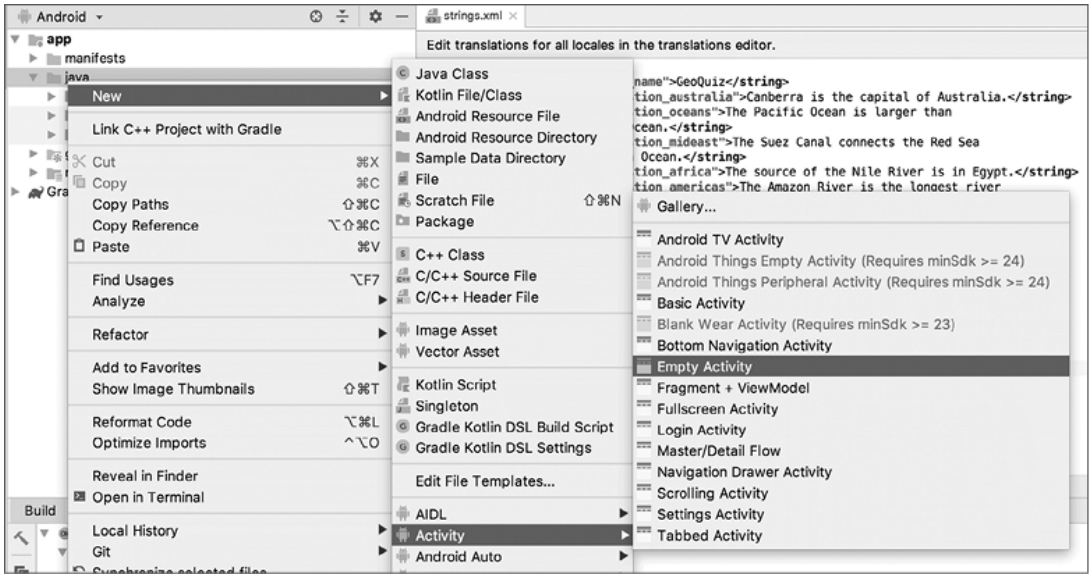


Рис. 6.3. Меню мастера New Activity

На экране появится диалоговое окно, изображенное на рис. 6.4. Введите в поле **ActivityName** строку **CheatActivity** — это имя вашего подкласса `Activity`. В поле **LayoutName** автоматически сгенерируется имя **activity_cheat**. Это базовое имя файла макета, создаваемого мастером.

Остальным полям можно оставить значения по умолчанию, но не забудьте убедиться в том, что имя пакета выглядит так, как ожидается. Оно определяет местонахождение `CheatActivity.kt` в файловой системе. Нажмите кнопку **Finish**, чтобы мастер завершил свою работу.

Теперь можно обратиться к пользовательскому интерфейсу. Снимок экрана в начале главы показывает, как должно выглядеть представление `CheatActivity`. На рис. 6.5 приведены определения виджетов.

Откройте файл `activity_cheat.xml` из каталога `res/layout` и перейдите в режим **Text**.

Создайте разметку XML для макета по образцу на рис. 6.5. Замените простой макет элементом `LinearLayout`, и так далее по дереву представлений. Сравните свою работу с листингом 6.2.

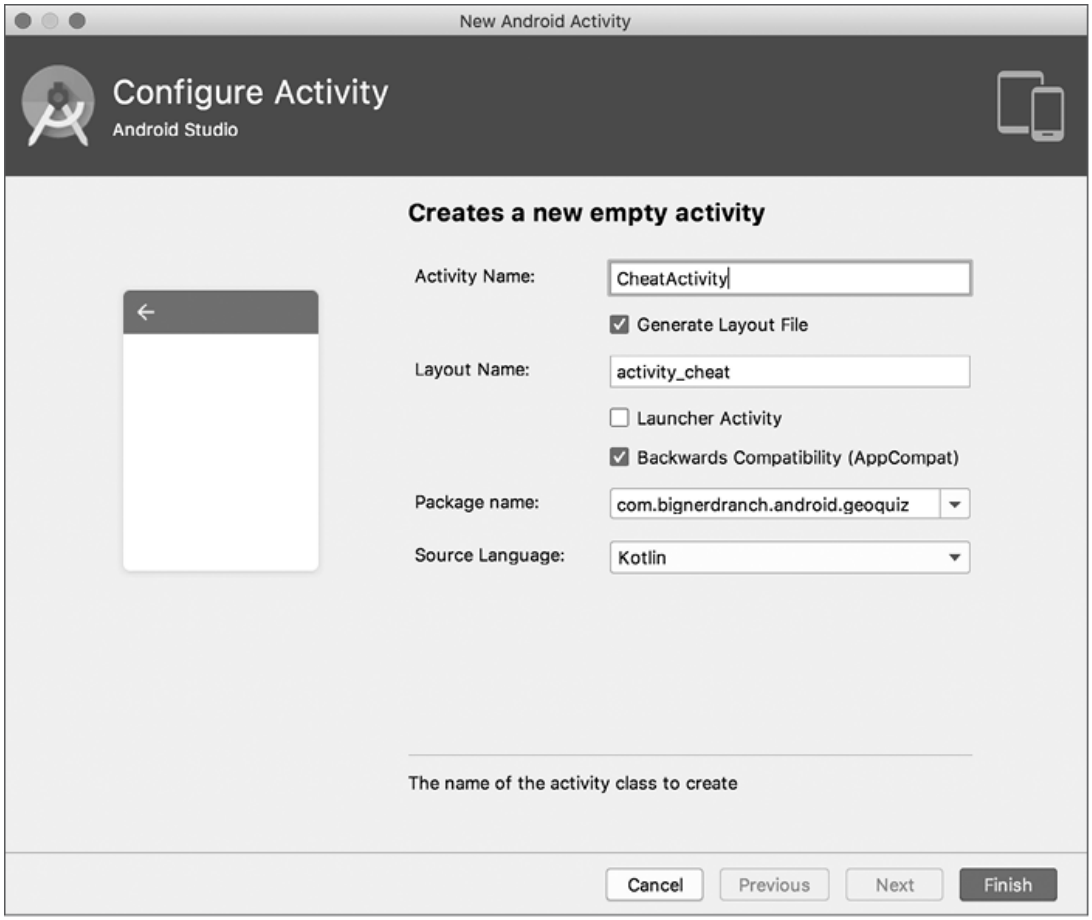


Рис. 6.4. Мастер New Empty Activity

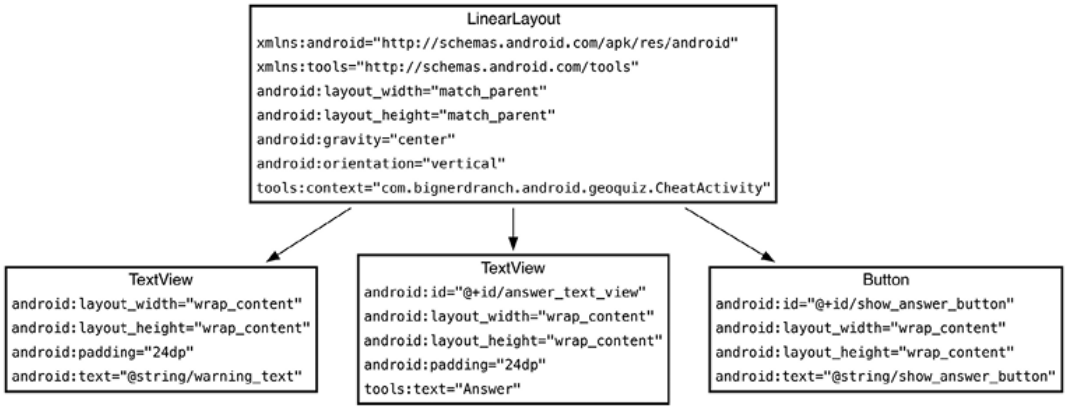


Рис. 6.5. Схема макета CheatActivity

Листинг 6.2. Заполнение макета второй activity (res/layout/activity_cheat.xml)

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical"
    tools:context="com.bignerdranch.android.geoquiz.CheatActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        android:text="@string/warning_text"/>

    <TextView
        android:id="@+id/answer_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        tools:text="Answer"/>

    <Button
        android:id="@+id/show_answer_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
            android:text="@string/show_answer_butto
n"/>

    </LinearLayout>
```

Мы не будем создавать для `activity_cheat.xml` альтернативный макет с альбомной ориентацией, однако есть возможность увидеть, как макет по умолчанию будет отображаться в альбомном режиме.

На вкладке **Design** окна редактора найдите на панели инструментов над областью предварительного просмотра кнопку, на которой изображено устройство с изогнутой стрелкой. Нажмите эту кнопку, чтобы изменить ориентацию макета (рис. 6.6).

Макет по умолчанию достаточно хорошо смотрится в обеих ориентациях; можно переходить к созданию подкласса activity.

Создание подкласса новой activity

Откройте в редакторе файл `CheatActivity.kt` с помощью панели **Project**, если он не открылся автоматически.

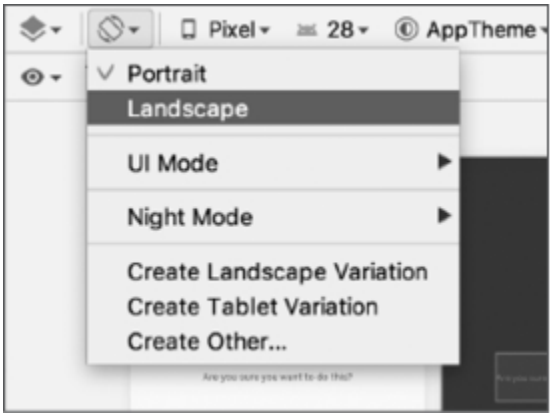


Рис. 6.6. Просмотр макета activity_cheat.xml в альбомной ориентации

Класс `CheatActivity` уже содержит простейшую реализацию `onCreate(Bundle?)`, которая передает идентификатор ресурса макета, определенного в `activity_cheat.xml` при вызове `setContentView(...)`.

Со временем функциональность функции `onCreate(Bundle?)` в `CheatActivity` будет расширена. А пока посмотрим, что еще мастер **NewActivity** сделал за вас: в манифест приложения было включено объявление `CheatActivity`.

Объявление activity в манифесте

Манифест (*manifest*) представляет собой XML-файл с метаданными, описывающими ваше приложение для ОС Android. Файл манифеста всегда называется `AndroidManifest.xml` и располагается в каталоге `app/manifests` вашего проекта.

На панели **Project** найдите и откройте файл `manifests/AndroidManifest.xml`. Также в Android Studio можно воспользоваться диалоговым окном **QuickOpen** — нажмите сочетание клавиш ⌘+⇧+O (**Ctrl+Shift+N**) и начните вводить имя файла. Когда в окне будет предложен правильный файл, нажмите клавишу ⏎ (**Enter**), чтобы открыть этот файл.

Каждая activity приложения должна быть объявлена в манифесте, чтобы она стала доступной для ОС.

Когда вы использовали мастер **NewProject** для создания `MainActivity`, мастер объявлял activity за вас. Аналогичным образом мастер **NewActivity** объявил `CheatActivity`, добавив разметку XML, выделенную в листинге 6.3.

Листинг 6.3. Объявление CheatActivity в манифесте (manifests/AndroidManifest.xml)

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.geoquiz">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
                                    <activity android:name=".CheatActivity">
            </activity>
        <activity android:name=".MainActivity">
            <intent-filter>
                                        <action android:name="android.intent.action.MAIN" />

                                    <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Атрибут android:name обязателен. Точка в начале значения атрибута сообщает ОС, что класс этой activity находится в пакете, который задается атрибутом package в элементе manifest в начале файла.

Иногда встречается полностью уточненный атрибут android:name вида android:name="com.bignerdranch.android.geoquiz.CheatActivity". Длинная форма записи идентична версии в листинге 6.3.

Манифест содержит много интересной информации, но сейчас мы хотим как можно быстрее наладить работу CheatActivity. Другие части манифеста будут рассмотрены позднее.

Добавление кнопки CHEAT! в MainActivity

Итак, пользователь должен нажать кнопку в MainActivity, чтобы вызвать на экран экземпляр CheatActivity. Следовательно, мы должны включить новые кнопки в res/layout/activity_main.xml и res/layout-land/activity_main.xml.

На рис. 6.2 вы видели, что над кнопкой **NEXT** появилась новая кнопка **CHEAT!**. В макете по умолчанию добавьте новую кнопку как прямого потомка корневого элемента LinearLayout. Ее определение должно непосредственно предшествовать кнопке **NEXT**.

Листинг 6.4. Добавление кнопки CHEAT! в макет по умолчанию (res/layout/activity_main.xml)

```
        ...
    </LinearLayout>
```

```
    <Button
        android:id="@+id/cheat_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp"
android:text="@string/cheat_button" />

    <Button
        android:id="@+id/next_button"
        .../>

</LinearLayout>
```

Листинг 6.5. Добавление кнопки CHEAT! в альбомный макет (res/layout-land/activity_main.xml)

```
        ...
    </LinearLayout>

    <Button
        android:id="@+id/cheat_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|center_h
orizontal"
        android:text="@string/cheat_button" />

    <Button
        android:id="@+id/next_button"
        .../>
```

```
</FrameLayout>
```

Снова откройте файл MainActivity.kt. Добавьте переменную, получите ссылку и назначьте заглушку View.OnClickListener для кнопки **CHEAT!**.

Листинг 6.6. Подключение кнопки CHEAT! (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    private lateinit var trueButton: Button
    private lateinit var falseButton: Button
    private lateinit var nextButton: Button
    private lateinit var cheatButton: Button
       private lateinit var questionTextView:
TextView
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
                                  nextButton    =
findViewById(R.id.next_button)
                                 cheatButton    =
findViewById(R.id.cheat_button)
                             questionTextView   =
findViewById(R.id.question_text_view)
        ...
        nextButton.setOnClickListener {
            quizViewModel.moveToNext()
            updateQuestion()
        }
```

```
            cheatButton.setOnClickListener {
                // Начало CheatActivity
            }

            updateQuestion()
        }
        ...
    }
```

Теперь можно переходить к запуску `CheatActivity`.

Запуск activity

Чтобы запустить одну activity из другой, проще всего воспользоваться функцией `startActivity(Intent)`.

Напрашивается предположение, что `startActivity(Intent)` — статическая функция, которая должна вызываться для запускаемого подкласса `Activity`. Тем не менее это не так. Когда activity вызывает `startActivity(Intent)`, этот вызов передается ОС.

Точнее, он передается компоненту ОС, который называется `ActivityManager`. `ActivityManager` создает экземпляр `Activity` и вызывает его функцию `onCreate(Bundle?)` (рис. 6.7).

Откуда `ActivityManager` узнает, какую activity следует запустить? Эта информация передается в параметре `Intent`.

Передача информации через интенты

Интент (*intent*) — объект, который может использоваться *компонентом* для взаимодействия с ОС. Пока что из компонентов нам встречались только activity, но еще

существуют службы (services), широковещательные приемники (broadcast receivers) и поставщики контента (content providers).

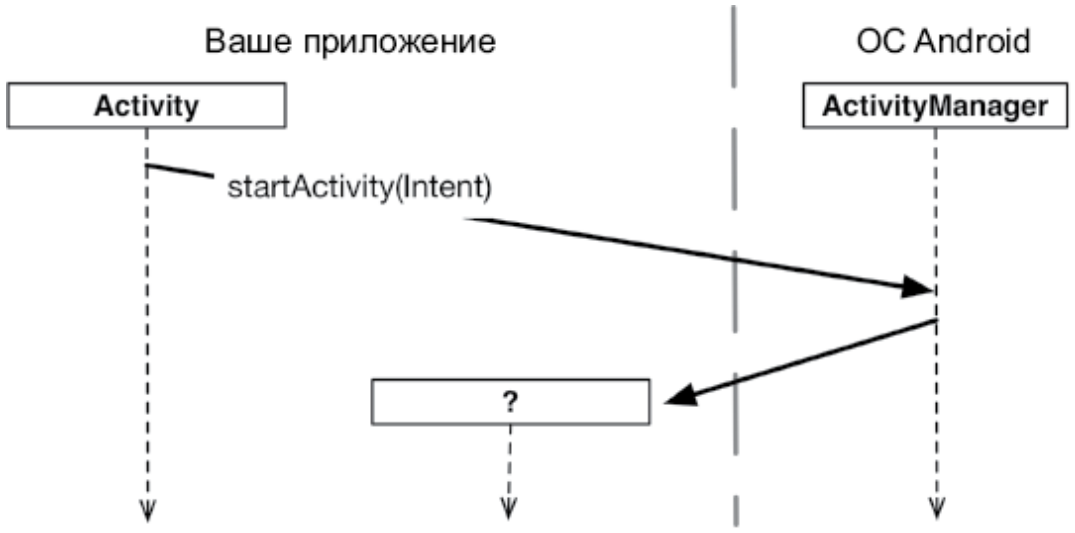


Рис. 6.7. Запуск activity

Интенты представляют собой многоцелевые средства передачи информации, а класс `Intent` предоставляет разные конструкторы в зависимости от того, для чего должен использоваться интент.

В данном случае интент сообщает `ActivityManager`, какую activity следует запустить, поэтому мы используем следующий конструктор:
`Intent(packageContext:Context,class:Class<?>)`.

В слушателе `cheatButton` создайте объект `Intent`, включающий класс `CheatActivity`, а затем передайте его при вызове `startActivity(Intent)` (листинг 6.7).

Листинг 6.7. Запуск CheatActivity (MainActivity.kt)

```
cheatButton.setOnClickListener {
    // Начало CheatActivity
```

```
            val intent = Intent(this,
CheatActivity::class.java)
    startActivity(intent)
}
```

Аргумент `Class` передает конструктору `Intent` инструкции, какой класс activity должен запустить `ActivityManager`. Аргумент `Context` определяет, в каком пакете приложения `ActivityManager` должен искать класс activity.

Прежде чем запустить activity, `ActivityManager` ищет в манифесте пакета объявление с именем, соответствующим заданному объекту `Class`. Если такое объявление будет найдено, activity запускается — все хорошо. Если объявление не найдено, выдается неприятное исключение `ActivityNotFoundException`, которое может привести к аварийному завершению приложения. Вот почему все activity должны объявляться в манифесте.

Запустите приложение GeoQuiz. Нажмите кнопку **CHEAT!**; на экране появится новая activity. Теперь нажмите кнопку «Назад». Activity `CheatActivity` уничтожится, а приложение возвратится к `MainActivity`.

Интенты явные и неявные

При создании объекта `Intent` с объектом `Context` и `Class` вы создаете *явный* (*explicit*) *интент*. Явные интенты используются для запуска activity в приложениях.

Может показаться странным, что две activity внутри приложения должны взаимодействовать через компонент `ActivityManager`, находящийся вне приложения. Тем не менее такая схема позволяет activity одного приложения легко работать с activity другого приложения.

Когда activity в вашем приложении должна запустить activity в другом приложении, вы создаете *неявный* (*implicit*) *интент*. Неявные интенты будут использоваться в главе 15.

Передача данных между activity

Итак, в нашем приложении действуют activity `MainActivity` и `CheatActivity`, и мы можем задуматься о передаче данных между ними. На рис. 6.8 показано, какие данные будут передаваться между двумя activity.

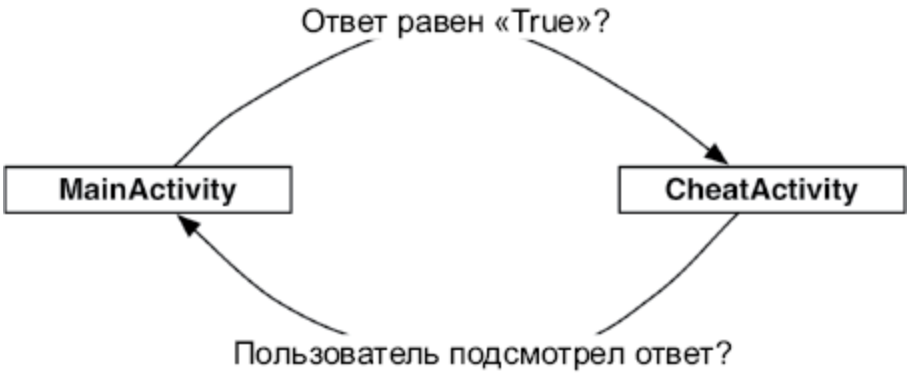


Рис. 6.8. Обмен данными между MainActivity и CheatActivity

`MainActivity` передает `CheatActivity` ответ на текущий вопрос при запуске `CheatActivity`.

Когда пользователь нажимает кнопку «Назад», чтобы вернуться к `MainActivity`, экземпляр `CheatActivity` уничтожается. В последний момент он передает `MainActivity` информацию о том, подсмотрел ли пользователь правильный ответ.

Начнем с передачи данных от `MainActivity` к `CheatActivity`.

Дополнения интентов

Чтобы сообщить `CheatActivity` ответ на текущий вопрос, мы будем передавать значение `questionBank[currentIndex].answer`.

Значение будет отправлено в виде *дополнения* (*extra*) объекта `Intent`, передаваемого `startActivity(Intent)`.

Дополнения представляют собой произвольные данные, которые вызывающая activity может передать вместе с интентом. Их можно рассматривать как аналоги аргументов конструктора, несмотря на то что вы не можете использовать произвольный конструктор с подклассом activity (Android создает экземпляры activity и несет ответственность за их жизненный цикл). ОС направляет интент activity получателю, которая обращается к дополнению и извлекает данные (рис. 6.9).

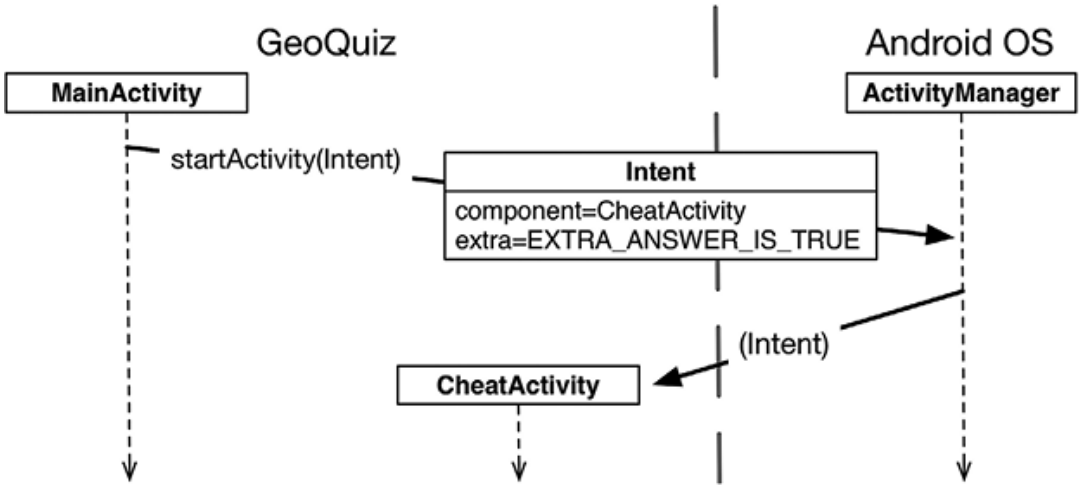


Рис. 6.9. Дополнения интентов: взаимодействие с другими activity

Дополнение представляет собой пару «ключ — значение» наподобие той, которая использовалась для сохранения значения `currentIndex` в `MainActivity.onSaveInstanceState(Bundle)`.

Для включения дополнений в интент используется функция `Intent.putExtra(...)`, а точнее

putExtra(name:String,value:Boolean).

Функция Intent.putExtra(...) существует в нескольких разновидностях, но всегда получает два аргумента. В первом аргументе передается ключ String, а во втором — значение того или иного типа. Функция всегда возвращает сам объект Intent, так что при необходимости можно использовать цепочки из сцепленных вызовов.

Добавьте в CheatActivity.kt ключ для дополнения.

Листинг 6.8. Добавление константы (CheatActivity.kt)

```
private const val EXTRA_ANSWER_IS_TRUE =
        "com.bignerdranch.android.geoquiz.answe
r_is_true"

class CheatActivity : AppCompatActivity() {
    ...
}
```

Activity может запускаться из нескольких разных мест, поэтому ключи для дополнений должны определяться в activity, которые читают и используют их. Как видно из листинга 6.8, уточнение дополнения именем пакета предотвращает конфликты имен с дополнениями других приложений.

Теперь можно вернуться к MainActivity и включить дополнение в интент, но это не лучший вариант. Нет никаких причин, по которым MainActivity или любому другому коду приложения было бы нужно знать подробности реализации тех данных, которые CheatActivity ожидает получить в дополнении интента. Эту работу правильнее инкапсулировать в функцию newIntent(...).

Создайте эту функцию в CheatActivity.

Листинг 6.9. Функция newIntent(...) для CheatActivity (CheatActivity.kt)

```
class CheatActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState:
Bundle?) {
        ...
    }

    companion object {
        fun newIntent(packageContext: Context,
answerIsTrue: Boolean): Intent {
            return Intent(packageContext,
CheatActivity::class.java).apply {
                putExtra(EXTRA_ANSWER_IS_TRUE,
answerIsTrue)
            }
        }
    }
}
```

Эта функция позволяет создать объект `Intent`, настроенный дополнениями, необходимыми для `CheatActivity`. Логический аргумент `answerIsTrue` помещается в интент под приватным именем с использованием константы `EXTRA_ANSWER_IS_TRUE`. Вскоре мы прочитаем это значение. Подобное использование функции `newIntent(...)` для подклассов activity упрощает настройку интентов в других местах кода.

Раз уж речь зашла о других местах, используем эту функцию в слушателе кнопки **CHEAT!**.

Листинг 6.10. Запуск CheatActivity.kt с дополнением (MainActivity.kt)

```
cheatButton.setOnClickListener {
    // Начало CheatActivity
                ~~val intent = Intent(this, CheatActivity::class.java)~~
                    val answerIsTrue = quizViewModel.currentQuestionAnswer
                        val intent = CheatActivity.newIntent(this@MainActivity, answerIsTrue)
    startActivity(intent)
}
```

В нашей ситуации достаточно одного дополнения, но при необходимости можно добавить в `Intent` несколько дополнений. В этом случае добавьте в `newIntent(...)` дополнительные аргументы в соответствии с используемой схемой.

Для чтения значения из дополнения используется функция `Intent.getBooleanExtra(String,Boolean)`.

Первый аргумент `getBooleanExtra(...)` содержит имя дополнения, а второй — ответ по умолчанию, если ключ не найден.

В `CheatActivity` прочитайте значение из дополнения в `onCreate(Bundle?)` и сохраните его в переменной.

Листинг 6.11. Использование дополнения (CheatActivity.kt)

```
class CheatActivity : AppCompatActivity() {

    private var answerIsTrue = false
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_cheat)

                                    answerIsTrue =
intent.getBooleanExtra(EXTRA_ANSWER_IS_TRUE,
false)
    }
    ...
}
```

Обратите внимание: `Activity.getIntent()` всегда возвращает объект `Intent`, который был передан в `startActivity(Intent)`.

Наконец, добавьте в `CheatActivity` код, обеспечивающий использование прочитанного значения в виджете `TextView` ответа и кнопке **SHOWANSWER**.

Листинг 6.12. Добавление функциональности подсматривания ответов (CheatActivity.kt)

```
class CheatActivity : AppCompatActivity() {

        private lateinit var answerTextView:
TextView
       private lateinit var showAnswerButton:
Button

    private var answerIsTrue = false
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        ...

                                    answerIsTrue    =
intent.getBooleanExtra(EXTRA_ANSWER_IS_TRUE,
false)
                                answerTextView    =
findViewById(R.id.answer_text_view)
                               showAnswerButton    =
findViewById(R.id.show_answer_button)
        showAnswerButton.setOnClickListener {
            val answerText = when {
                                    answerIsTrue  ->
R.string.true_button
                else -> R.string.false_button
            }
            answerTextView.setText(answerText)
        }
    }
    ...
}
```

Код достаточно тривиален: мы задаем текст `TextView` при помощи функции `TextView.setText(int)`. Функция `TextView.setText(...)` существует в нескольких вариантах; здесь используется вариант, получающий идентификатор строкового ресурса.

Запустите приложение GeoQuiz. Нажмите кнопку **CHEAT!**, чтобы перейти к `CheatActivity`. Затем нажмите кнопку **SHOWANSWER**, чтобы открыть ответ на текущий вопрос.

Получение результата от дочерней activity

В текущей версии приложения пользователь может беспрепятственно жульничать. Сейчас мы исправим этот недостаток; для этого `CheatActivity` будет сообщать `MainActivity`, подсмотрел ли пользователь ответ.

Чтобы получить информацию от дочерней activity, вызовите функцию `Activity.startActivityForResult(Intent,Int)`.

Первый параметр содержит тот же интент, что и прежде. Во втором параметре передается *код запроса* — определяемое пользователем целое число, которое передается дочерней activity, а затем принимается обратно родителем. Оно используется тогда, когда activity запускает сразу несколько типов дочерних activity и ей необходимо определить, кто из потомков возвращает данные. `MainActivity` запускает только один тип дочерней activity, но передача константы в коде запроса — полезная практика, с которой вы будете готовы к возможным будущим изменениям.

В классе `MainActivity` измените слушателя `cheatButton` и включите в него вызов `startActivityForResult(Intent,int)`.

Листинг 6.13. Вызов startActivityForResult(...) (MainActivity.kt)

```
private const val TAG = "MainActivity"
private const val KEY_INDEX = "index"
private const val REQUEST_CODE_CHEAT = 0

class MainActivity : AppCompatActivity() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
```

```
            ...
            cheatButton.setOnClickListener {
                ...
                    ~~startActivity(intent)~~
                      startActivityForResult(intent,
REQUEST_CODE_CHEAT)
            }

            updateQuestion()
        }
        ...
}
```

Передача результата

Существует две функции, которые могут вызываться в дочерней activity для возвращения данных родителю:

```
setResult(resultCode: Int)
setResult(resultCode: Int, data: Intent)
```

Как правило, *код результата* содержит одну из двух предопределенных констант: `Activity.RESULT_OK` или `Activity.RESULT_CANCELED`. (Также можно использовать другую константу, `RESULT_FIRST_USER`, как смещение при определении собственных кодов результатов.)

Назначение кода результата полезно в том случае, когда родитель должен выполнить разные действия в зависимости от того, как завершилась дочерняя activity.

Например, если в дочерней activity присутствуют кнопки **OK** и **Cancel**, то дочерняя activity может назначать разные коды результата в зависимости от того, какая кнопка была нажата.

Родительская activity будет выбирать разные варианты действий в зависимости от полученного кода.

Вызов `setResult(...)` необязателен для дочерней activity. Если вам не нужно различать результаты или получать произвольные данные по интенту, просто разрешите ОС отправить код результата по умолчанию. Код результата всегда возвращается родителю, если дочерняя activity была запущена функцией `startActivityForResult(...)`. Если функция `setResult(...)` не вызывалась, то при нажатии пользователем кнопки «Назад» родитель получит код `Activity.RESULT_CANCELED`.

Возвращение интента

В нашей реализации дочерняя activity должна вернуть `MainActivity` некоторые данные. Соответственно, мы создадим объект `Intent`, поместим в него дополнение, а затем вызовем `Activity.setResult(int,Intent)` для передачи этих данных `MainActivity`.

Добавьте в `CheatActivity` константу для ключа дополнения и приватную функцию, которая выполняет необходимую работу. Затем включите вызов этой функции в слушателя кнопки `SHOWANSWER`.

Листинг 6.14. Назначение результата (CheatActivity.kt)

```
const val EXTRA_ANSWER_SHOWN = 
"com.bignerdranch.android.geoquiz.answer_shown"
private const val EXTRA_ANSWER_IS_TRUE =
        "com.bignerdranch.android.geoquiz.answe
r_is_true"
```

```
class CheatActivity : AppCompatActivity() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        showAnswerButton.setOnClickListener {
            ...
            answerTextView.setText(answerText)
            setAnswerShownResult(true)
        }
    }

                                private        fun
setAnswerShownResult(isAnswerShown: Boolean) {
        val data = Intent().apply {
                    putExtra(EXTRA_ANSWER_SHOWN,
isAnswerShown)
        }
        setResult(Activity.RESULT_OK, data)
    }
    ...
}
```

Когда пользователь нажимает кнопку **SHOWANSWER**, `CheatActivity` упаковывает код результата и интент в вызове `setResult(int,Intent)`.

Затем, когда пользователь нажимает кнопку «Назад» для возвращения к `MainActivity`, `ActivityManager` вызывает следующую функцию родительской activity:

```
onActivityResult(requestCode: Int, resultCode:
Int, data: Intent)
```

В параметрах передается исходный код запроса от `MainActivity`, код результата и интент, переданный `setResult(int,Intent)`.

Последовательность взаимодействий изображена на рис. 6.10.

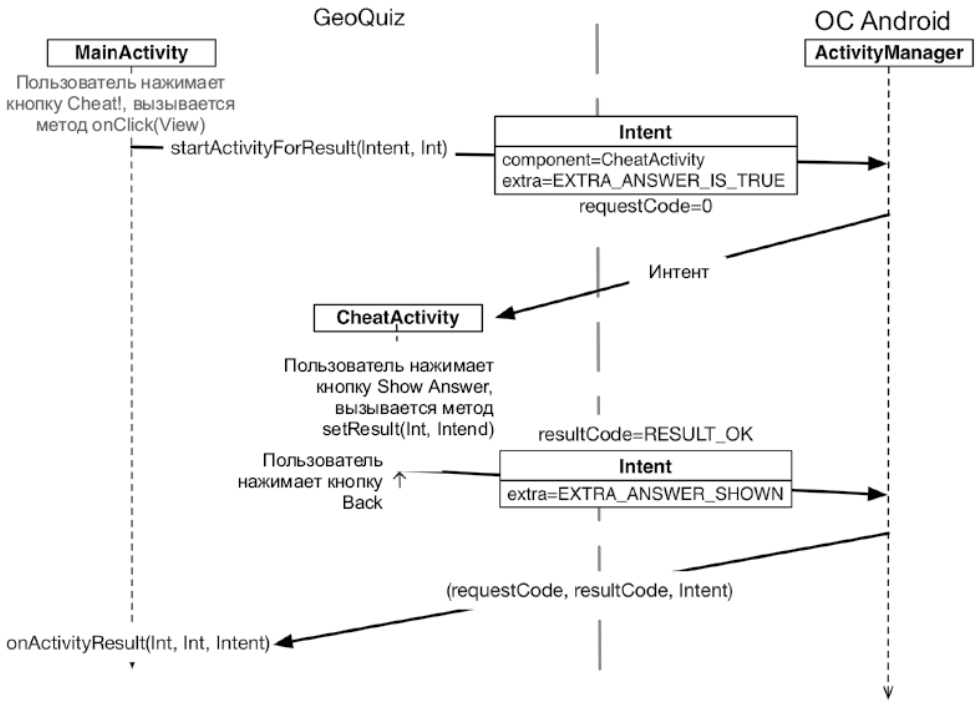


Рис. 6.10. Диаграмма последовательности для GeoQuiz

Остается последний шаг: переопределение `onActivityResult(int,int,Intent)` в `MainActivity` для обработки результата.

Обработка результата

В `QuizViewModel.kt` добавьте новое свойство для хранения значения, которое `CheatActivity` передает обратно. Чит-статус пользователя является частью состояния пользовательского интерфейса. Сохранение значения в `QuizViewModel` вместо `MainActivity` означает, что значение

будет сохраняться при изменении конфигурации, а не уничтожаться с действием, как обсуждалось в главе 4.

Листинг 6.15. Отслеживание читерства в QuizViewModel (QuizViewModel.kt)

```
class QuizViewModel : ViewModel() {

    var currentIndex = 0
    var isCheater = false
    ...
}
```

Добавьте в MainActivity.kt новую переменную для хранения значения, возвращаемого CheatActivity. Затем включите в переопределение onActivityResult(...) код его получения, проверки кода запроса и кода результата, чтобы быть уверенным в том, что они соответствуют ожиданиям. Как и в предыдущем случае, это полезная практика, которая упростит возможные будущие изменения.

Листинг 6.16. Реализация onactivityResult(...) (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
    }

     override fun onActivityResult(requestCode:
Int,
```

```
                                        resultCode:
Int,
                                    data: Intent?)
{
            super.onActivityResult(requestCode,
resultCode, data)

        if (resultCode != Activity.RESULT_OK) {
            return
        }

         if (requestCode == REQUEST_CODE_CHEAT)
{
            quizViewModel.isCheater =
                data?.getBooleanExtra(EXTRA_ANS
WER_SHOWN, false) ?: false
        }
    }
    ...
}
```

Измените функцию checkAnswer(Boolean) в MainActivity. Она должна проверять, подсмотрел ли пользователь ответ, и реагировать соответствующим образом.

Листинг 6.17. Изменение уведомления в зависимости от значения isCheater (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
          private fun checkAnswer(userAnswer:
Boolean) {
```

```
                val correctAnswer: Boolean =
quizViewModel.currentQuestionAnswer

            val messageResId = if (userAnswer
== correctAnswer) {
                R.string.correct_toast
            } else {
                R.string.incorrect_toast
            }
        val messageResId = when {
                    quizViewModel.isCheater ->
R.string.judgment_toast
                userAnswer == correctAnswer ->
R.string.correct_toast
            else -> R.string.incorrect_toast
        }
             Toast.makeText(this, messageResId,
Toast.LENGTH_SHORT)
                .show()
    }
}
```

Запустите приложение GeoQuiz. Нажмите кнопку **CHEAT!**, затем нажмите кнопку **SHOWANSWER**. После этого нажмите кнопку «Назад». Попробуйте ответить на текущий вопрос. На экране должно появиться осуждение.

Что произойдет, если вы перейдете к следующему вопросу? Вы все равно читер. Если вы хотите ослабить правила в отношении жульничества, выполните соответствующее упражнение в конце главы.

На данный момент функция GeoQuiz завершена. В следующей главе мы наведем небольшой лоск путем включения

анимации перехода к activity при отображении CheatActivity. При этом вы узнаете, как включить новейшие функции Android, сохранив поддержку старых версий Android.

Ваши activity с точки зрения Android

Давайте посмотрим, что происходит при переключении между activity с точки зрения ОС. Прежде всего, когда вы щелкаете на приложении GeoQuiz в лаунчере, ОС запускает не приложение, а activity в приложении. А если говорить точнее, запускается *activityлаунчера* приложения. Для GeoQuiz activity лаунчера является MainActivity.

Когда мастер **NewProject** создавал приложение GeoQuiz, класс MainActivity был назначен activity лаунчера по умолчанию. Статус activity лаунчера задается в манифесте элементом intent-filter в объявлении MainActivity (листинг 6.18).

Листинг 6.18. В MainActivity объявляется activity лаунчера (AndroidManifest.xml)

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    ... >

    <application
        ... >
                                        <activity
android:name=".CheatActivity">
        </activity>
        <activity android:name=".MainActivity">
            <intent-filter>
```

```
                <action
android:name="android.intent.action.MAIN"/>

                <category
android:name="android.intent.category.LAUNCHER"
/>
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Когда экземпляр `MainActivity` окажется на экране, пользователь может нажать кнопку **CHEAT!**. При этом экземпляр `CheatActivity` запустится поверх `MainActivity`. Эти activity образуют стек (рис. 6.11).

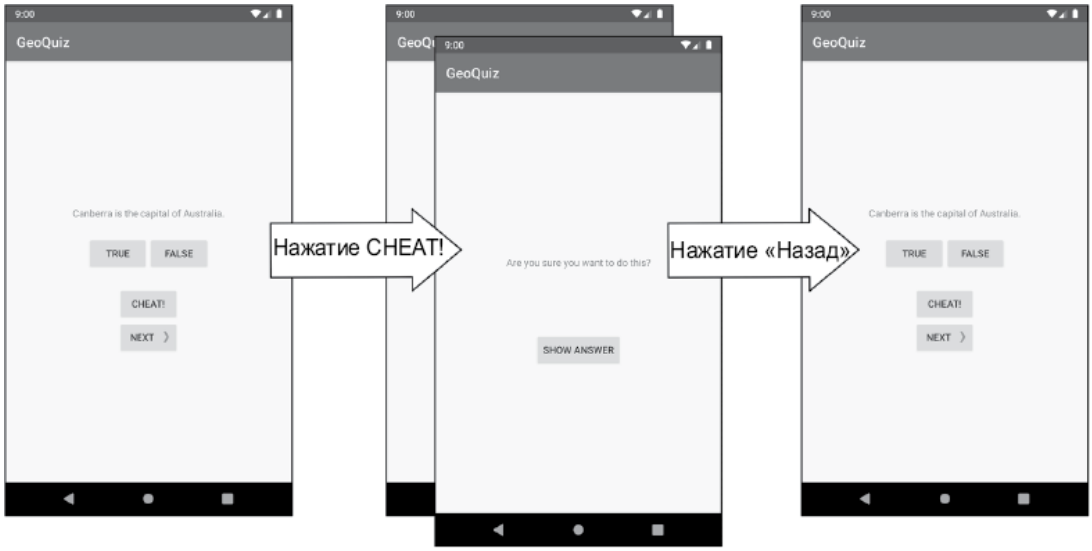


Рис. 6.11. Стек возврата GeoQuiz

При нажатии кнопки «Назад» в `CheatActivity` этот экземпляр выводится из стека, а `MainActivity` занимает свою позицию на вершине, как показано на рис. 6.11.

Вызов `Activity.finish()` в `CheatActivity` также выводит `CheatActivity` из стека.

Если запустить GeoQuiz и нажать кнопку «Назад» в `MainActivity`, то `MainActivity` будет извлечена из стека и вы вернетесь к последнему экрану, который просматривался перед запуском GeoQuiz (рис. 6.12).

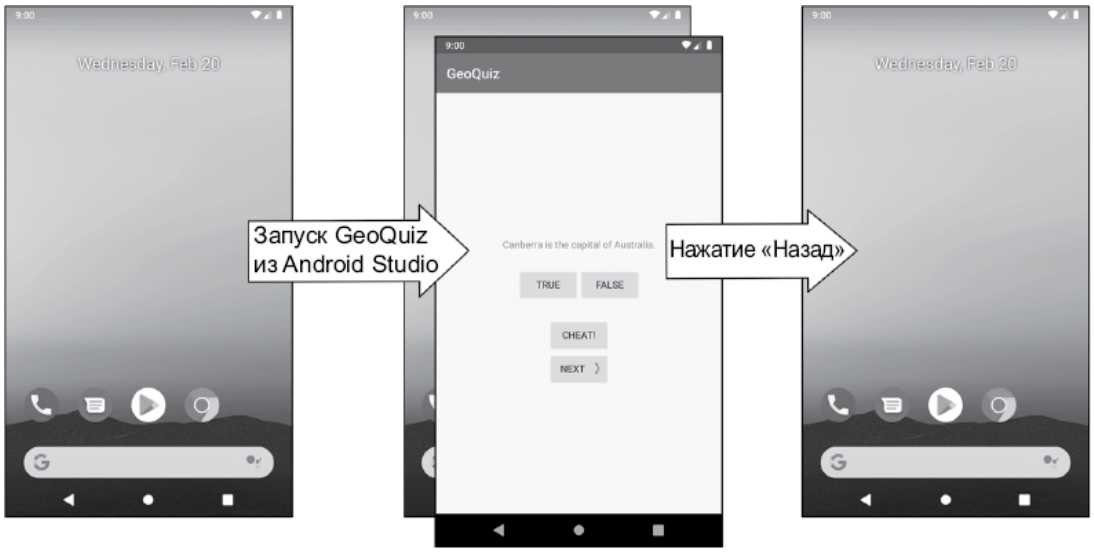


Рис. 6.12. Переход к главному экрану

Если вы запустили GeoQuiz из лаунчера, то нажатие кнопки «Назад» из `MainActivity` вернет вас обратно (рис. 6.13).

Нажатие кнопки «Назад» в запущенном лаунчере вернет вас к экрану, который был открыт перед его запуском.

Мы видим, что `ActivityManager` поддерживает *стек возврата* (*back stack*) и что этот стек не ограничивается activity вашего приложения. Он совместно использует activity всех приложений; это одна из причин, по которым `ActivityManager` участвует в запуске activity и находится под управлением ОС, а не вашего приложения. Стек представляет использование ОС и устройства в целом, а не использование одного приложения.

(Как насчет кнопки «Вверх»? О том, как реализовать и настроить эту кнопку, расскажем в главе 14.)

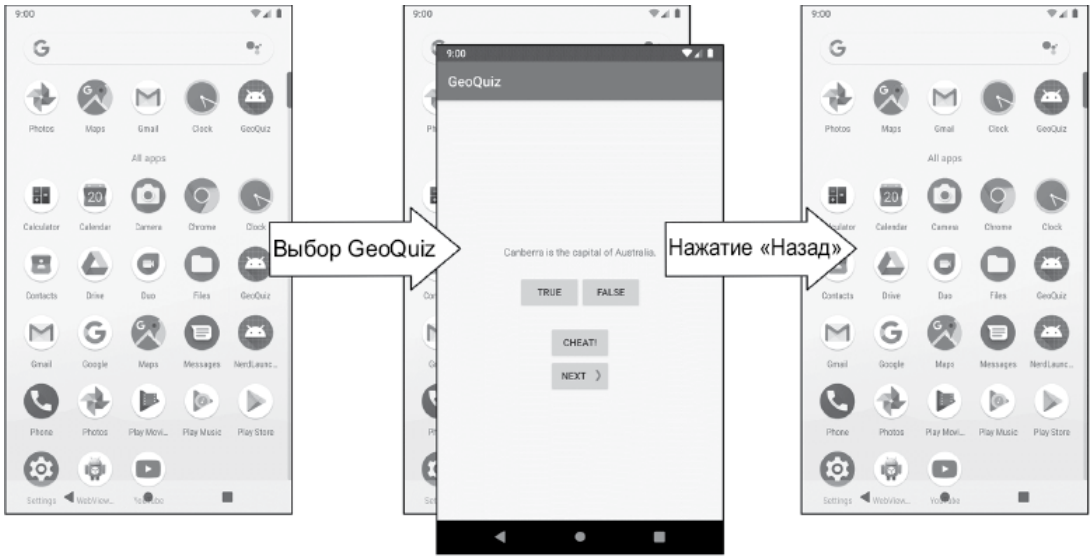


Рис. 6.13. Запуск GeoQuiz из лаунчера

Упражнение. Лазейка для читера

Мошенники никогда не выигрывают... Если, конечно, им не удастся обойти вашу защиту от мошенничества. А скорее всего, они так и сделают — именно потому, что они мошенники.

У GeoQuiz есть кое-какая лазейка. Пользователи могут вращать `CheatActivity` после чита, чтобы удалить следы обмана. После возврата к `MainActivity` их жульничество будет забыто.

Исправьте эту ошибку, сохраняя состояние пользовательского интерфейса `CheatActivity` во время вращения и после уничтожения процесса, воспользовавшись методами из главы 4.

Упражнение. Отслеживание читов по вопросу

В настоящее время, когда пользователь читерит на одном вопросе, он считается читером по всем вопросам. Обновите GeoQuiz, чтобы отслеживать, сколько раз пользователь нарушал закон. Когда пользователь использует чит для ответа на заданный вопрос, осуждайте его всякий раз, когда он пытается ответить на этот вопрос. Когда пользователь отвечает на вопрос, с которым он не жульничал, покажите правильный или неправильный ответ.

7. Версии Android SDK и совместимость

Теперь, после «боевого крещения» с приложением GeoQuiz, пора поближе познакомиться с разными версиями Android. Информация этой главы пригодится вам в следующих главах книги, когда мы займемся разработкой более сложных и реалистичных приложений.

Версии Android SDK

В табл. 7.1 перечислены версии SDK, соответствующие версии прошивки Android, и процент устройств, использующих их, по состоянию на май 2019 года.

(Версии Android, используемые менее чем на 0,1 % устройств, в таблице не приводятся.)

За каждым выпуском с кодовым названием следуют инкрементные выпуски. Например, платформа Android 4.0 (API уровня 14) Ice Cream Sandwich была почти немедленно заменена инкрементными выпусками, которые в конечном итоге привели к появлению Android 4.0.3 и 4.0.4 (API уровня 15).

Конечно, проценты из табл. 7.1 постоянно изменяются, но в них проявляется важная закономерность: устройства Android со старыми версиями не подвергаются немедленному обновлению или замене при появлении новой версии. По состоянию на май 2019 года почти 11 % устройств все еще работали под управлением KitKat или более ранней версии. Версия Android 4.4 (последнее обновление KitKat) была выпущена в октябре 2013 года.

Таблица 7.1. Уровни API Android, версии прошивки и процент устройств

Уровень API	Кодовое название	Версия прошивки устройства	% используемых устройств
28	Pie	9.0	10,4
27	Oreo	8.1	15,4
26		8.0	12,9
25	Nougat	7.1	7,8
24		7.0	11,4
23	Marshmallow	6.0	16,9
22	Lollipop	5.1	11,5
21		5.0	3,0
19	KitKat	4.4	6,9
18	Jelly Bean	4.3	0,5
17		4.2	1,5
16		4.1	1,2
15	Ice Cream Sandwich (ICS)	4.0.3, 4.0.4	0,3
10	Gingerbread	2.3.3–2.3.7	0,3

(Обновленные данные из таблицы доступны по адресу **developer.android.com/about/dashboards/index.html**.)

Почему на многих устройствах продолжают работать старые версии Android? В основном из-за острой конкуренции между производителями устройств Android и операторами сотовой связи. Операторы стремятся иметь возможности и телефоны, которых нет у других сетей. Производители устройств тоже испытывают давление — все их телефоны базируются на одной ОС, но им нужно выделяться на фоне конкурентов. Сочетание этого давления со стороны рынка и операторов сотовой связи привело к появлению многочисленных устройств со специализированными модификациями Android.

Устройство со специализированной версией Android не сможет перейти на новую версию, выпущенную Google. Вместо этого ему придется ждать совместимого «фирменного» обновления, которое может выйти через несколько месяцев после выпуска версии Google… А может вообще не выйти — производители часто предпочитают расходовать ресурсы на новые устройства, а не на обновление старых.

Совместимость и программирование Android

Из-за задержек обновления в сочетании с регулярным выпуском новых версий совместимость становится важной проблемой в программировании Android. Чтобы привлечь широкую аудиторию, разработчики Android должны создавать приложения, которые хорошо работают на устройствах с тремя-четырьмя версиями Android, а также на устройствах различных формфакторов.

Поддержка разных размеров не столь сложна, как может показаться. Смартфоны выпускаются с экранами различных размеров, но система макетов Android хорошо приспосабливается к ним. С планшетами дело обстоит сложнее, но в этом случае на помощь приходят квалификаторы конфигураций (как будет показано в главе 17). Впрочем, для устройств Android TV и Android Wear (также работающих под управлением Android) различия в пользовательском интерфейсе настолько серьезны, что разработчику приходится переосмысливать организацию взаимодействия с пользователем и дизайн приложения.

Разумный минимум

Самой старой версией Android, поддерживаемой в примерах книги, является API уровня 21 (Lollipop). Также встречаются

упоминания более старых версий, но мы сосредоточимся на тех версиях, которые считаем современными (API уровня 21+). Доля устройств с версиями Gingerbread, Ice Cream Sandwich, Jelly Bean и KitKat падает от месяца к месяцу, и объем работы, необходимой для поддержки старых версий, перевешивает пользу от них.

Инкрементные выпуски не создают особых проблем с обратной совместимостью. Изменение основной версии — совсем другое дело. Работа, необходимая для поддержки только устройств 5.x, не так уж велика, но если вам также необходимо поддерживать устройства 4.x, придется потратить время на анализ различий между версиями. К счастью, компания Google предоставила специальные библиотеки, которые упростят задачу. Вы узнаете о них в следующих главах.

При создании проекта GeoQuiz в мастере нового приложения вы выбирали минимальную версию SDK (рис. 7.1). (Учтите, что в Android термины «версия SDK» и «уровень API» являются синонимами.)

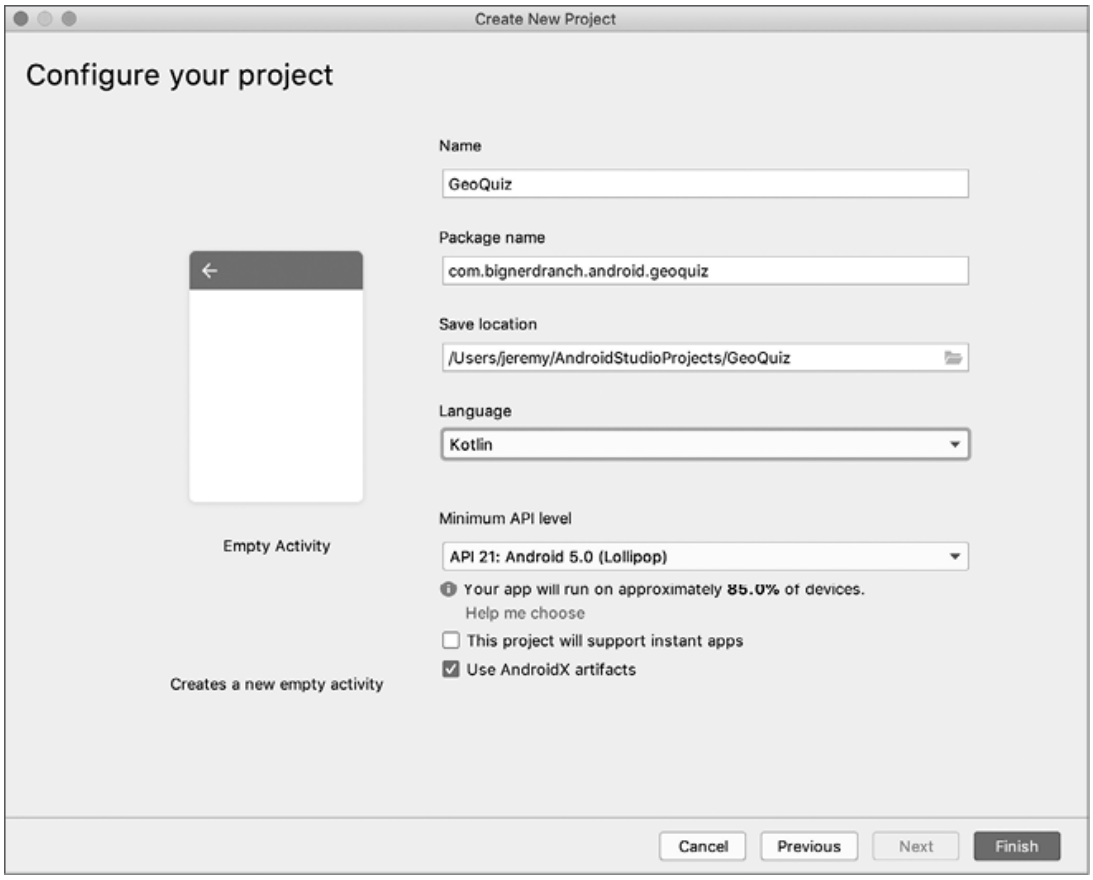


Рис. 7.1. Еще не забыли?

Кроме минимальной поддерживаемой версии также можно задать *целевую* версию (*target* version) и версию *сборки* (*build* version). Давайте разберемся, чем различаются эти версии и как изменить их.

Все эти свойства задаются в файле `build.gradle` в модуле `app`. Версия сборки задается исключительно в этом файле. Минимальная и целевая версии SDK задаются в файле `build.gradle`, но используются для замены или инициализации значений из файла `AndroidManifest.xml`.

Откройте файл `build.gradle`, находящийся в модуле `app`. Обратите внимание на значения `compileSdkVersion`, `minSdkVersion` и `targetSdkVersion`.

```
compileSdkVersion 28
defaultConfig {
                                        applicationId
"com.bignerdranch.android.geoquiz"
    minSdkVersion 21
    targetSdkVersion 28
```

Минимальная версия SDK

Значение `minSdkVersion` определяет нижнюю границу, за которой ОС отказывается устанавливать приложение.

Выбирая API уровня 21 (Lollipop), вы разрешаете Android устанавливать GeoQuiz на устройствах с версией Lollipop и выше. Android откажется устанавливать GeoQuiz на устройствах с версией ниже, чем Lollipop.

Снова обращаясь к табл. 7.1, мы видим, почему API 21 — подходящий вариант в качестве минимальной версии SDK: в этом случае приложение можно будет устанавливать более чем на 90 % используемых устройств.

Целевая версия SDK

Значение `targetSdkVersion` сообщает Android, для какого уровня API проектировалось ваше приложение. Чаще всего в этом поле указывается новейшая версия Android.

Когда следует понижать целевую версию SDK? Новые выпуски SDK могут изменять внешний вид вашего приложения на устройстве и даже поведение ОС «за кулисами». Если ваше

приложение уже спроектировано, убедитесь в том, что в новых версиях оно работает так, как должно. За информацией о возможных проблемах обращайтесь к документации по адресу **developer.android.com/reference/android/os/Build.VERSION_CODES.html**. Далее либо измените свое приложение, чтобы оно работало с новым поведением, либо понизьте целевую версию SDK.

Понижение целевой версии SDK гарантирует, что приложение будет работать с внешним видом и поведением целевой версии, в которой оно хорошо работало. Этот параметр существует для обеспечения совместимости с новыми версиями Android, так как все изменения последующих версий игнорируются до увеличения `targetSdkVersion`.

Важно отметить, что Google ограничивает нижнюю планку целевой SDK, если вы хотите отправить ваше приложение в Play Store. На момент написания этой статьи любые новые приложения или обновления должны иметь целевой SDK не ниже 26 уровня API (Oreo), иначе Play Store отклонит их. Это гарантирует, что пользователи смогут получить максимум от улучшений производительности и безопасности последних версий Android. Требования к минимальным версиям будут возрастать по мере выхода новых версий Android, поэтому обязательно следите за документацией, чтобы знать, когда нужно обновлять целевую версию.

Версия SDK для компиляции

Последний параметр SDK обозначен в листинге именем `compileSdkVersion`. В файле `AndroidManifest.xml` этот параметр отсутствует. Если минимальная и целевая версии SDK помещаются в манифест и сообщаются ОС, версия SDK для компиляции относится к закрытой информации, известной только вам и компилятору.

Функциональность Android используется через классы и функции SDK. Версия SDK, используемая для компиляции, указывает, какая версия должна использоваться при сборке вашего кода. Когда Android Studio ищет классы и функции, на которые вы ссылаетесь в директивах импорта, версия SDK для компиляции определяет, в какой версии SDK будет осуществляться поиск.

В качестве версии SDK для компиляции лучше всего выбрать новейший уровень API. Тем не менее при необходимости версию SDK для компиляции существующего приложения можно изменить, например при выпуске очередной версии Android, чтобы вы могли использовать новые функции и классы, появившиеся в этой версии.

Все эти параметры — минимальную версию SDK, целевую версию SDK, версию SDK для компиляции — можно изменить в файле `build.gradle`, однако следует помнить, что изменения проявятся только после синхронизации проекта с измененными файлами.

Безопасное добавление кода для более поздних версий API

Различия между минимальной версией SDK и версией SDK для компиляции создают проблемы совместимости, которые необходимо учитывать. Например, что произойдет в GeoQuiz при выполнении кода, рассчитанного на версию SDK после минимальной версии Lollipop (API уровня 21)? Если установить такое приложение и запустить его на устройстве с Lollipop, произойдет сбой.

Раньше тестирование в подобных ситуациях было сущим кошмаром. Однако благодаря усовершенствованию Android Lint проблемы, порожденные вызовом нового кода на старых устройствах, теперь успешно обнаруживаются. Если вы

используете код версии, превышающей минимальную версию SDK, Android Lint сообщит об ошибке сборки.

Весь код текущей версии GeoQuiz совместим с API уровня 21 и ниже. Добавим код API уровня выше 21 (Lollipop) и посмотрим, что произойдет.

Откройте файл `MainActivity.kt`. Включите в функции `OnClickListener` кнопки CHEAT следующий фрагмент кода для создания круговой анимации на время появления `CheatActivity`.

Листинг 7.1. Добавление кода анимации (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        cheatButton.setOnClickListener { view ->
            // Начало CheatActivity
                        val answerIsTrue =
quizViewModel.currentQuestionAnswer
                            val intent =
CheatActivity.newIntent(this@MainActivity,
answerIsTrue)
            val options = ActivityOptions
                    .makeClipRevealAnimation(vi
ew, 0, 0, view.width, view.height)

                startActivityForResult(intent,
REQUEST_CODE_CHEAT, options.toBundle())
        }
```

```
            updateQuestion()
        }
        ...
    }
```

Вы используете класс `ActivityOptions`, чтобы настроить запуск activity. В приведенном выше коде вы вызываете функцию `makeClipRevealAnimation(...)`, чтобы указать, что `CheatActivity` должен использовать анимацию отображения. Значения, которые вы передаете в `makeClipRevealAnimation(...)`, указывают на объект, который следует использовать как источник анимации (в данном случае кнопка **CHEAT!**), положение *x* и *y* (относительно источника) для начала отображения новой activity, а также начальную ширину и высоту новой activity.

Обратите внимание, что вы назвали лямбда-аргумент `view`, а не использовали имя по умолчанию. В контексте настройки слушателя щелчка мыши лямбда-аргумент представляет собой кликнутый виджет. Присваивать имя аргументу необязательно, но это может улучшить читабельность вашего кода. Мы рекомендуем называть аргумент, когда он используется где-то еще.

Наконец, вы вызвали функцию `options.toBundle()`, чтобы упаковать `ActivityOptions` в объект `Bundle`, а затем передали их в функцию `startActivityForResult(...)`. `ActivityManager` использует пакет опций, чтобы определить, как вывести вашу activity на экран.

Обратите внимание, что в строке, где вы вызываете `ActivityOptions.makeClipRevealAnimation(...)`, появляется ошибка Lint в виде красной загогулины под названием функции, а при нажатии на функцию появляется

красная лампочка. Функция `makeClipRevealAnimation(...)` была добавлена в Android SDK на уровне API 23, поэтому данный код будет аварийно завершаться на устройстве под управлением API 22 или ниже.

Поскольку заданная вами версия SDK для компиляции равна API 28-го уровня, то сам компилятор проблем с этим кодом испытывать не будет. А вот Android Lint знает минимальную версию SDK, и поэтому жалуется.

Сообщения об ошибках будут выглядеть примерно так: «Call requires API level 23 (current min is 21)». Запуск кода с этим предупреждением возможен, но Lint знает, что это небезопасно.

Как избавиться от ошибок? Первый способ — поднять минимальную версию SDK до 23. Однако тем самым вы не столько решаете проблему совместимости, сколько обходите ее. Если ваше приложение не может устанавливаться на устройствах с API уровня 23 и более старых устройствах, то проблема совместимости исчезает.

Другое, более правильное решение — заключить код более высокого уровня API в условную конструкцию, которая проверяет версию Android на устройстве.

Листинг 7.2. Предварительная проверка версии Android на устройстве (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
    @SuppressLint("RestrictedApi")
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
```

```
        cheatButton.setOnClickListener { view ->
            ...
                if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
                val options = ActivityOptions
                        .makeClipRevealAnimation(view, 0, 0, view.width, view.height)

                startActivityForResult(intent, REQUEST_CODE_CHEAT, options.toBundle())
            } else {
                startActivityForResult(intent, REQUEST_CODE_CHEAT)
            }
        }
        updateQuestion()
    }
```

Константа `Build.VERSION.SDK_INT` определяет версию Android на устройстве. Она сравнивается с константой, соответствующей M (Marshmallow). (Коды версий доступны по адресу **developer.android.com/reference/android/os/Build.VERSION_CODES.html**.)

Теперь код анимации будет выполняться только в том случае, если приложение работает на устройстве с API уровня 23 и выше. Код стал безопасным для API уровня 21, Android Lint не на что жаловаться.

Запустите GeoQuiz на устройстве с версией Marshmallow или выше и проверьте, как работает новая анимация.

Переходная анимация довольно быстрая, и вы не сможете отличить новую анимацию от оригинальной. Для того чтобы увидеть разницу, вы можете настроить ваше устройство на замедление переходов. Откройте экран настроек и зайдите в опции разработчика (**System⇒Advanced⇒Developeroptions**). Найдите опцию **Transitionanimationscale** (в разделе **Drawing**), выберите ее и присвойте значение **AnimationScale10x** (рис. 7.2).

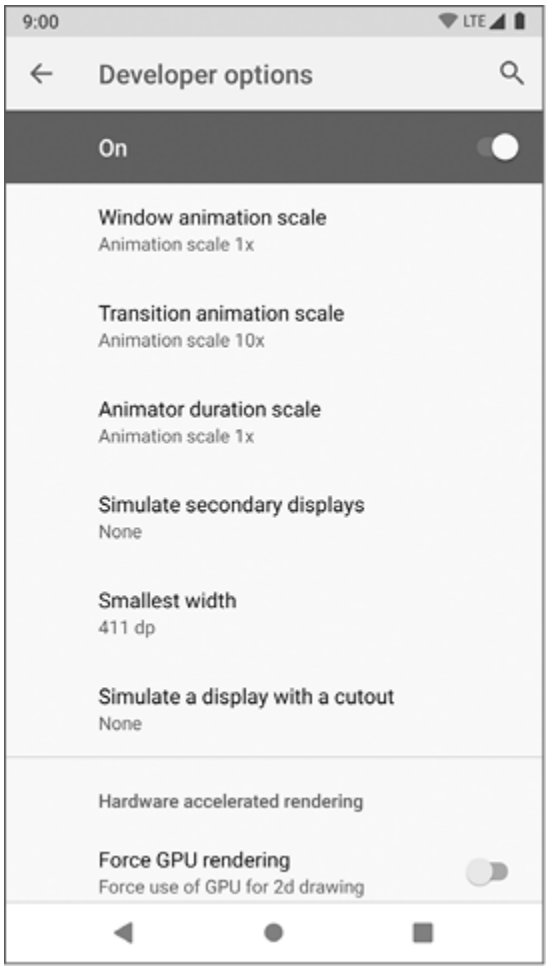


Рис. 7.2. Замедление переходов

Эта настройка замедлит переходы в 10 раз, что поможет увидеть, насколько новая анимация отличается от старой. Вернитесь в GeoQuiz и запустите приложение заново, чтобы увидеть ваш новый переход в замедленном режиме. Отмените изменения в `MainActivity`, если вы хотите увидеть, как выглядит первоначальный переход. Перед тем как двигаться дальше, вернитесь к опциям разработчика и настройте анимацию перехода на скорость 1x.

Вы также можете запустить GeoQuiz на устройстве Lollipop (виртуальном или другом). Анимация будет не такой, но вы можете проверить, что приложение работает нормально.

Вы увидите еще один пример защиты более новых API в главе 27.

Jetpack-библиотеки

Несмотря на то что ограничение уровней API работает, по нескольким причинам оно не оптимально. Во-первых, это лишняя работа для вас как для разработчика, так как для разных версий нужно иметь различные пути в вашем приложении. Во-вторых, это означает, что пользователи вашего приложения будут видеть разные вещи в зависимости от версии их устройства.

В главе 4 вы узнали о библиотеках Jetpack и AndroidX. В дополнение к новым функциям (таким как `ViewModel`) Jetpack-библиотеки также позволяют получить обратную совместимость для новых функций на старых моделях устройств и правильное поведение во всех версиях Android, если это возможно. По крайней мере, использование библиотек позволяет вводить как можно меньше условных проверок API.

Многие из библиотек AndroidX в Jetpack являются модификациями библиотек поддержки, которые им

предшествовали. Лучше всего использовать именно эти библиотеки. Это облегчит жизнь вам как разработчику, так как вам больше не придется думать о различных версиях API и обрабатывать каждый случай в отдельности. Ваши пользователи тоже будут довольны, потому что приложение будет выглядеть одинаково во всех версиях.

К сожалению, Jetpack-библиотеки не являются панацеей для проблем совместимости, потому что в них доступны не все функции, которые могут потребоваться. Команда Android работает над добавлением в Jetpack-библиотеки новых API, но все равно не все API будут доступны. В этом случае вам нужно будет использовать явные проверки версий, пока не появится соответствующая версия Jetpack.

Документация разработчика Android

Ошибки Android Lint сообщают, к какому уровню API относится несовместимый код. Однако вы также можете узнать, к какому уровню API относятся конкретные классы и функции, — эта информация содержится в документации разработчика Android.

Постарайтесь поскорее научиться работать с документацией. Информация Android SDK слишком обширна, чтобы держать ее в голове, а с учетом регулярного появления новых версий разработчик должен знать, что нового появилось, и уметь пользоваться этими новшествами.

Документация для разработчиков Android — отличный и объемный источник информации. Главная страница документации — **developer.android.com**. Она разделена на шесть частей: Platform, Android Studio, Google Play, Android Jetpack, Docs и News. При случае стоит все это внимательно изучить. В каждом разделе описаны различные аспекты разработки под

Android, начиная с азов и заканчивая развертыванием приложения в Play Store.

Platform

Информация о базовой платформе, с акцентом на поддерживаемые форм-факторы и различные версии Android.

Android Studio

Статьи об IDE, которые помогут вам изучить различные инструменты и рабочие процессы, облегчающие вашу жизнь.

Google Play

Советы и хитрости по развертыванию приложений, а также для успешной работы с пользователями.

Android Jetpack

Информация о библиотеках Jetpack и о том, как команда разработчиков Android стремится улучшить процесс разработки приложений. Мы используем кое-какие Jetpack-библиотеки в этой книге, но в этом разделе приведен их полный список.

Docs

Главная страница документации для разработчиков. Здесь вы найдете информацию об отдельных классах во фреймворке, а также множество учебников и примеров кода, которые вы можете проработать, чтобы улучшить и отточить свои навыки.

News

Статьи о последних новостях в Android.

Вы также можете скачать документацию на ПК, чтобы читать ее при отсутствии интернета. На панели **SDKManager** в Android Studio (**Tools⇒SDKManager**) выберите вкладку **SDKTools**. Выберите пункт **DocumentationforAndroidSDK** и нажмите кнопку

Apply. Появится информация об объеме загрузки и запрос на подтверждение. После завершения загрузки вы сможете получить доступ к документации по вашей файловой системе. В каталоге, куда вы загрузили SDK (который можно проверить в диспетчере SDK, если вы его не знаете), появится новый каталог `docs`, содержащий полную документацию.

На сайте документации определите, к какому уровню API принадлежит файл, к которому относится функция `makeClipRevealAnimation(...)`, найдя его имя с помощью строки поиска в правом верхнем углу. Выберите результат `ActivityOptions` (скорее всего, это первый результат поиска), чтобы перейти на страницу ссылок на класс, показанную на рис. 7.3. С правой стороны этой страницы расположены ссылки на различные разделы.

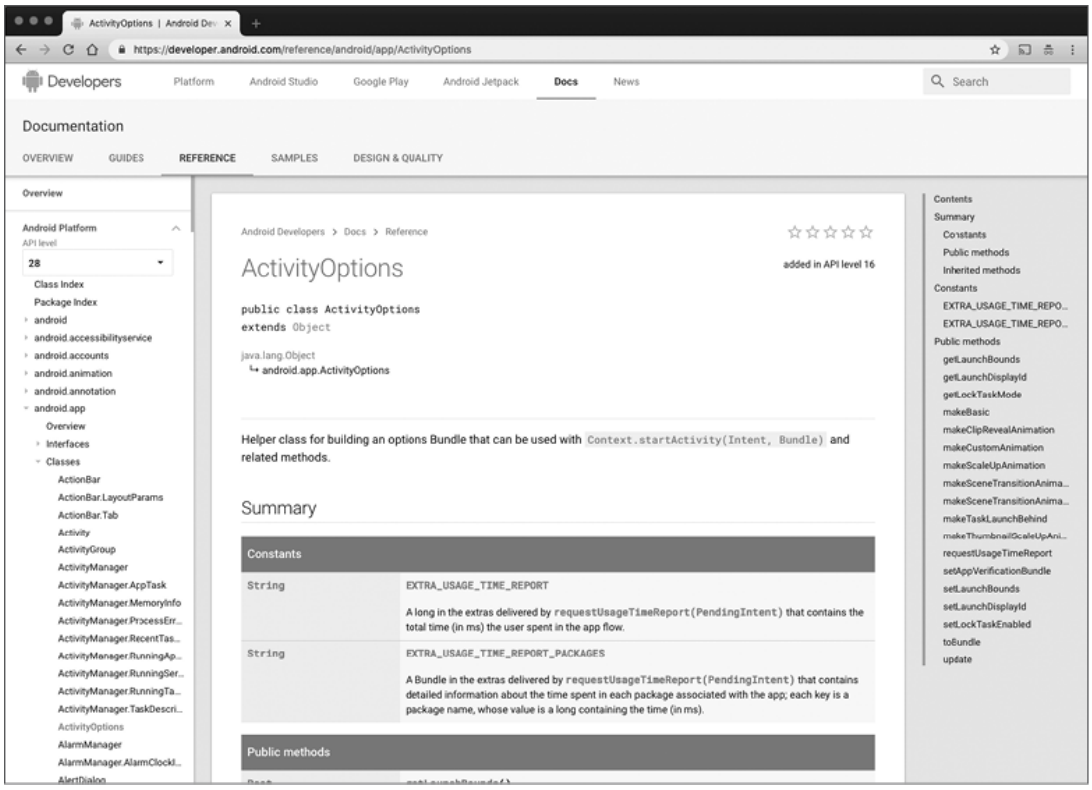


Рис. 7.3. Страница со справочным описанием ActivityOptions

Прокрутите список, найдите функцию `makeClipRevealAnimation(...)` и щелкните мышью по имени функции, чтобы увидеть описание. Справа от сигнатуры функции видно, что функция `makeClipRevealAnimation(...)` появилась в API уровня 23.

Если вы хотите узнать, какие функции `ActivityOptions` доступны, скажем, в API уровня 21, отфильтруйте справочник по уровню API. В левой части страницы, где классы индексируются по пакету, найдите категорию `API level: 28`. Щелкните мышью по близлежащему элементу управления и выберите в списке 21. Все, что появилось в Android после API уровня 21, выделяется в списке серым цветом.

Фильтр уровня API приносит намного больше пользы классам, доступным на используемом вами уровне API. Найдите в документации страницу класса Activity. Верните в фильтре уровня API значение 21; обратите внимание, как много функций появилось с момента выхода API: например, функция `onMultiWindowModeChanged(...)`, включенная в SDK в Nougat, позволяет получать уведомления, когда activity меняет режим с полноэкранного на мультиоконный и наоборот.

Продолжая чтение книги, постарайтесь почаще возвращаться к документации. Она наверняка понадобится вам для выполнения упражнений, однако не пренебрегайте самостоятельными изысканиями каждый раз, когда вам захочется побольше узнать о некотором классе, функции и т.д. Создатели Android постоянно обновляют и совершенствуют свою документацию, и вы всегда узнаете из нее что-то новое.

Упражнение. Вывод версии Android на устройстве

Добавьте в макет GeoQuiz виджет `TextView` для вывода уровня API устройства, на котором работает программа. На рис. 7.4 показано, как должен выглядеть результат.

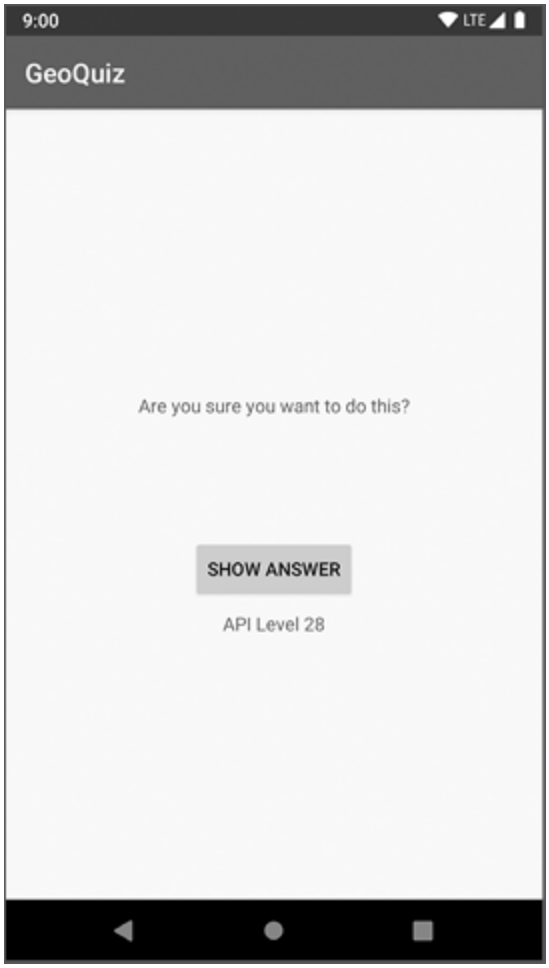


Рис. 7.4. Результат упражнения

Задать текст `TextView` в макете не получится, потому что версия операционной системы устройства не известна до момента выполнения. Найдите функцию `TextView` для задания текста в справочной странице `TextView` в документации Android. Вам нужна функция, получающая один аргумент — строку (или `CharSequence`).

Для настройки размера и гарнитуры текста используйте атрибуты XML, перечисленные в описании `TextView`.

Упражнение. Ограничение подсказок

Ограничьте пользователя тремя подсказками. Храните информацию о том, сколько раз пользователь подсматривал ответ, и выводите количество оставшихся подсказок под кнопкой. Если ни одной подсказки не осталось, то кнопка получения подсказки блокируется.

8. UI-фрагменты и FragmentManager

В этой главе мы начнем строить приложение CriminalIntent. Оно предназначено для хранения информации об «офисных преступлениях»: грязной посуде, оставленной в раковине, или пустом лотке общего принтера после печати документов.

В приложении CriminalIntent пользователь создает запись о преступлении с заголовком, датой и фотографией. Также можно выбрать подозреваемого в адресной книге и отправить жалобу по электронной почте, опубликовать ее в Twitter, Facebook или другом приложении. Сообщив о преступлении, пользователь освобождается от негатива и может сосредоточиться на текущей задаче.

CriminalIntent — сложное приложение, для разработки которого нам понадобится целых 13 глав. В нем используется интерфейс типа «*список/детализация*»: на главном экране выводится список зарегистрированных преступлений. Пользователь может добавить новое или выбрать существующее преступление для просмотра и редактирования информации (рис. 8.1).

Гибкость пользовательского интерфейса

Разумно предположить, что приложение типа «список/детализация» состоит из двух activity: для управления списком и для управления детализированным представлением. Щелчок на преступлении в списке запускает экземпляр детализированной activity. Нажатие кнопки «Назад» уничтожает activity детализации и возвращает на экран список, в котором можно выбрать другое преступление.

Такая архитектура работает, но что делать, если вам потребуется более сложная схема представления информации и навигации между экранами?

Допустим, пользователь запустил приложение CriminalIntent на планшете. Экраны планшетов и некоторых больших телефонов достаточно велики для одновременного отображения списка и детализации, по крайней мере в альбомной ориентации (рис. 8.2).

Пользователь просматривает описание преступления на телефоне и хочет увидеть следующее преступление в списке. Было бы удобно, если бы пользователь мог провести пальцем по экрану, чтобы перейти к следующему преступлению без возвращения к списку.

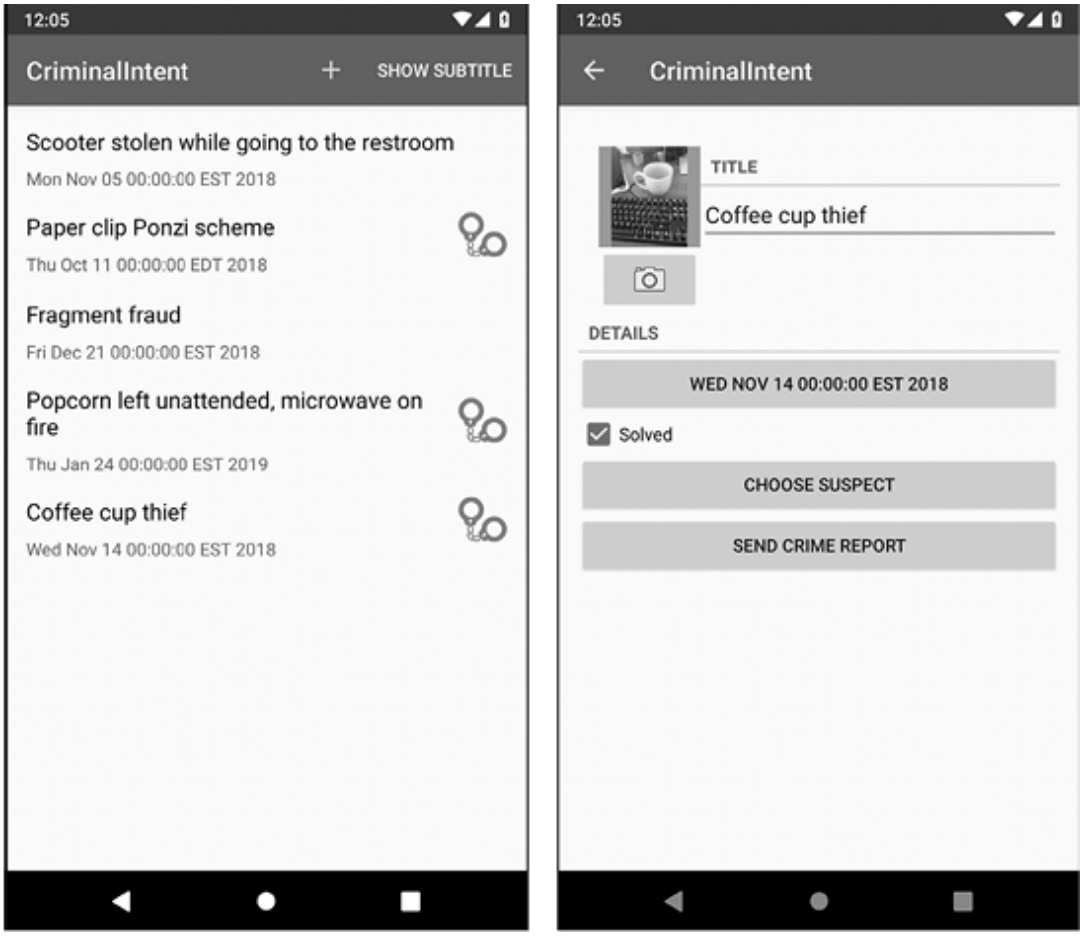


Рис. 8.1. Приложение CriminalIntent

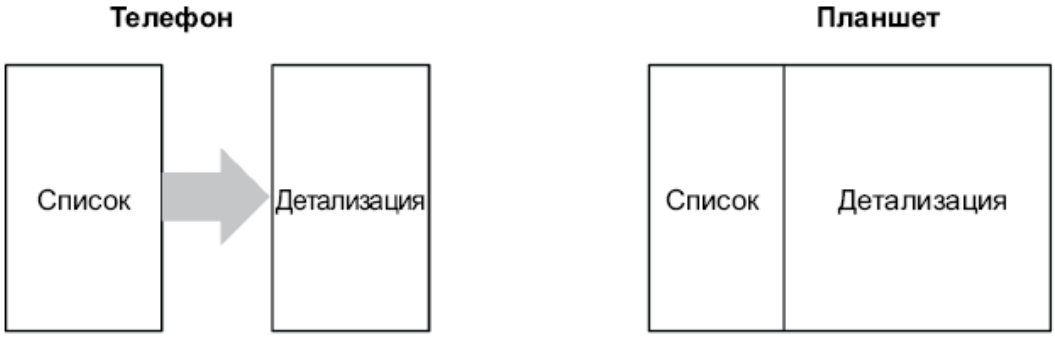


Рис. 8.2. Идеальный интерфейс «список/детализация» для телефонов и планшетов

Эти сценарии объединяет гибкость пользовательского интерфейса: возможность формирования и изменения представления activity во время выполнения в зависимости от того, что нужно пользователю или устройству.

Подобная гибкость в activity не предусмотрена. Представления activity могут изменяться во время выполнения, но код, управляющий этими изменениями, должен находиться в представлении. В результате activity тесно связывается с конкретным экраном, с которым работает пользователь.

Знакомство с фрагментами

Закон Android нельзя нарушить, но можно обойти, передав управление пользовательским интерфейсом приложения от activity одному или нескольким *фрагментам*.

Фрагмент представляет собой объект контроллера, которому activity может доверить выполнение операций. Чаще всего такой операцией является управление пользовательским интерфейсом — целым экраном или его частью.

Фрагмент, управляющий пользовательским интерфейсом, называется UI-фрагментом. UI-фрагмент имеет собственное представление, которое заполняется на основании файла

макета. Представление фрагмента содержит элементы пользовательского интерфейса, с которыми будет взаимодействовать пользователь.

Представление activity содержит место, в которое вставляется представление фрагмента. Более того, хотя в нашем примере activity управляет одним фрагментом, она может содержать несколько мест для представлений нескольких фрагментов.

Фрагменты, связанные с activity, могут использоваться для формирования и изменения экрана в соответствии с потребностями приложения и пользователей. Представление activity формально остается неизменным на протяжении жизненного цикла, и законы Android не нарушаются.

Давайте посмотрим, как эта схема работает в приложении «список/детализация». Представление activity строится из фрагмента списка и фрагмента детализации. Представление детализации содержит подробную информацию о выбранном элементе списка.

При выборе другого элемента на экране появляется новое детализированное представление. С фрагментами эта задача решается легко; activity заменяет фрагмент детализации другим фрагментом детализации (рис. 8.3). Это существенное изменение представления происходит без уничтожения activity.

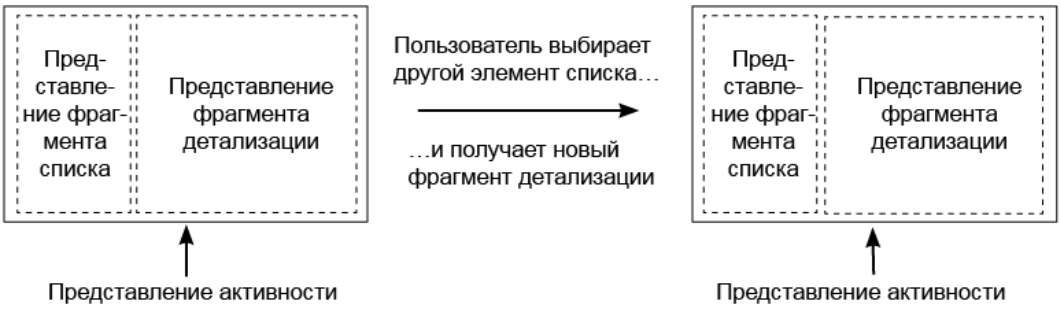


Рис. 8.3. Переключение фрагмента детализации

Применение UI-фрагментов позволяет разделить пользовательский интерфейс вашего приложения на структурные блоки, а это полезно не только для приложений «список/детализация». Работа с отдельными блоками упрощает разработку интерфейсов со вкладками, анимированных боковых панелей и многого другого. Кроме того, некоторые из новых Android Jetpack API, такие как контроллер навигации, лучше всего работают с фрагментами. Таким образом, использование фрагментов позволяет добиться хорошей интеграции с API Jetpack.

Начало работы над CriminalIntent

В этой главе мы возьмемся за представление детализации приложения CriminalIntent. На рис. 8.4 показано, как будет выглядеть приложение CriminalIntent к концу главы.

Экраном, показанным на рис. 8.4, будет управлять UI-фрагмент `CrimeFragment`. *Хостом* (*host*) экземпляра `CrimeFragment` является activity `MainActivity`.

Пока считайте, что хост предоставляет позицию в иерархии представлений, в которой фрагмент может разместить свое представление (рис. 8.5). Фрагмент не может вывести представление на экран сам по себе. Его представление отображается только при размещении в иерархии activity.

Проект CriminalIntent будет большим; диаграмма объектов поможет понять логику его работы. На рис. 8.6 изображена общая структура CriminalIntent. Запоминать все объекты и связи между ними необязательно, но прежде чем трогаться с места, полезно хотя бы в общих чертах понимать, куда вы направляетесь.

Мы видим, что класс `CrimeFragment` делает примерно то же, что в GeoQuiz делали activity: он создает пользовательский

интерфейс и управляет им, а также обеспечивает взаимодействие с объектами модели.

Рис. 8.4. Приложение CriminalIntent к концу главы

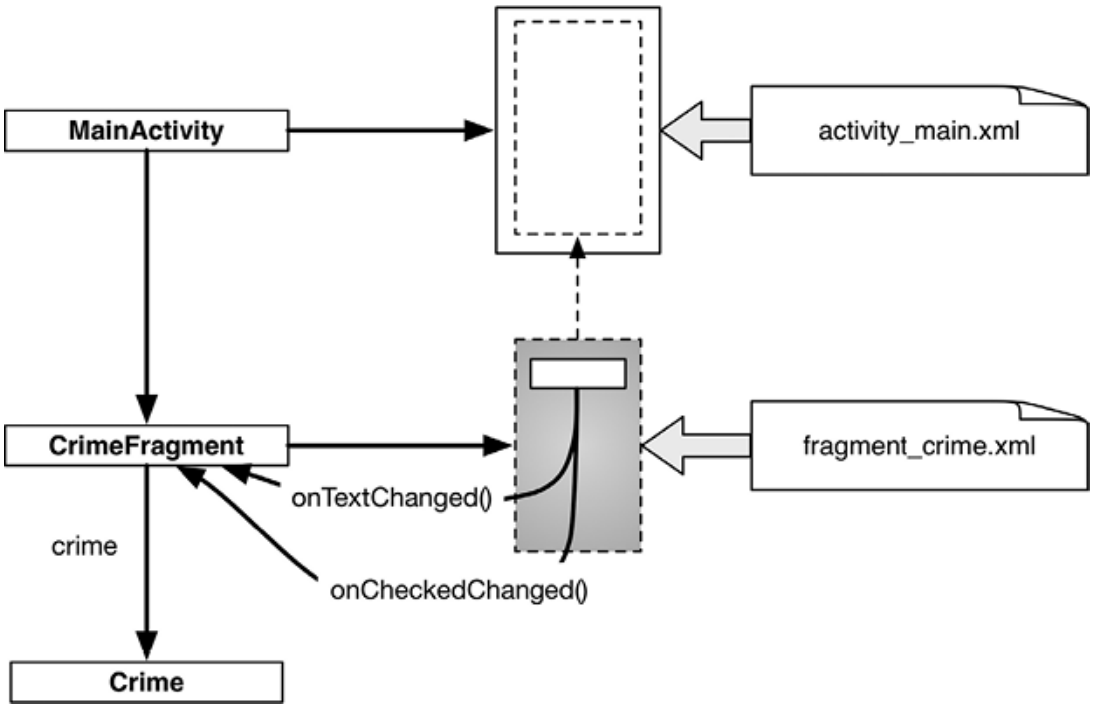


Рис. 8.5. MainActivity как хост CrimeFragment

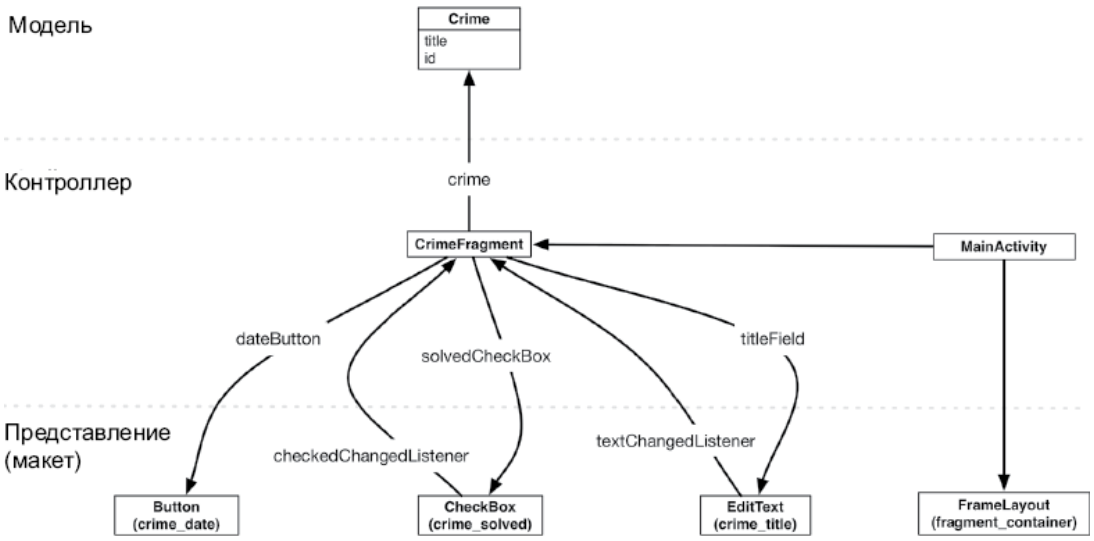


Рис. 8.6. Диаграмма объектов CriminalIntent (для этой главы)

Мы напишем три класса, изображенные на рис. 8.6: `Crime`, `CrimeFragment` и `MainActivity`.

Экземпляр `Crime` представляет одно офисное преступление. В этой главе описание преступления будет состоять только из

заголовка и идентификатора. Заголовок содержит текст (например, «Свалка токсичных отходов в раковине» или «Кто-то украл мой йогурт!»), а идентификатор однозначно определяет экземпляр `Crime`.

В этой главе мы для простоты будем использовать один экземпляр `Crime`. В класс `CrimeFragment` включается свойство (`crime`) для хранения этого отдельного инцидента.

Представление `MainActivity` состоит из элемента `FrameLayout`, определяющего место, в котором будет отображаться представление `CrimeFragment`.

Представление `CrimeFragment` состоит из элемента `LinearLayout` с несколькими дочерними представлениями, включая `EditText`, `Button` и `CheckBox`. `CrimeFragment` определяет поле для каждого из этих представлений и назначает для него слушателя, обновляющего уровень модели при изменении текста.

Создание нового проекта

Но довольно разговоров — пора создать новое приложение. Создайте новое Android-приложение (**File⇒New⇒NewProject**). Выберите шаблон **EmptyActivity** (рис. 8.7). Нажмите кнопку **Next**.

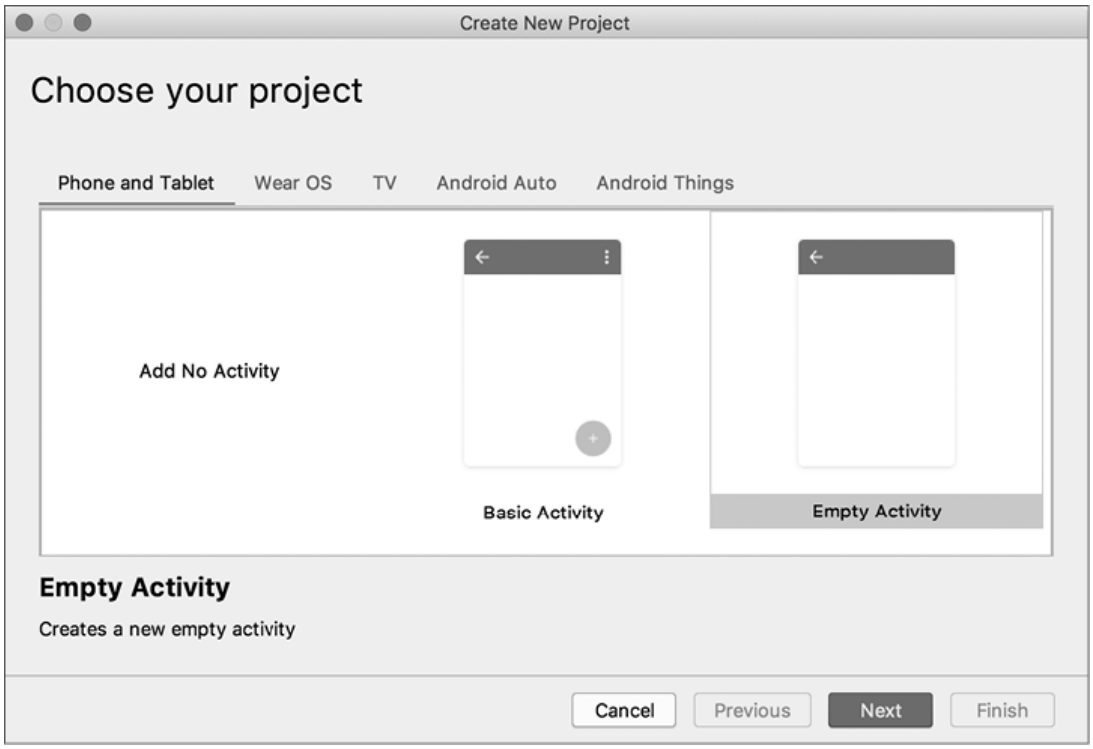


Рис. 8.7. Создание приложения CriminalIntent

Настройте ваш проект, как показано на рис. 8.8. Назовите приложение **CriminalIntent**. Убедитесь, что задано имя пакета **com.bignerdranch.android.criminalintent**, а язык — **Kotlin**. Выберите **API21:Android5.0(Lollipop)** из раскрывающегося списка **MinimumAPIlevel**. Наконец, установите флажок **UseAndroidXartifacts**.

Нажмите кнопку **Finish**, чтобы создать проект.

Прежде чем продолжать работу над `MainActivity`, мы создадим уровень модели CriminalIntent. Для этого мы напишем класс `Crime`.

Создание класса Crime

На панели **Project** щелкните правой кнопкой мыши по пакету `com.bignerdranch.android.criminalintent` и выберите

команду **New⇒KotlinFile/Class** в контекстном меню. Введите имя файла `Crime.kt`.

Добавьте в `Crime.kt` поля для представления идентификатора преступления, названия, даты и статуса и конструктор, инициализирующий поля идентификатора и даты (листинг 8.1). Помимо полей, добавьте в определение класса `Crime` ключевое слово `data`, чтобы он превратился в класс данных.

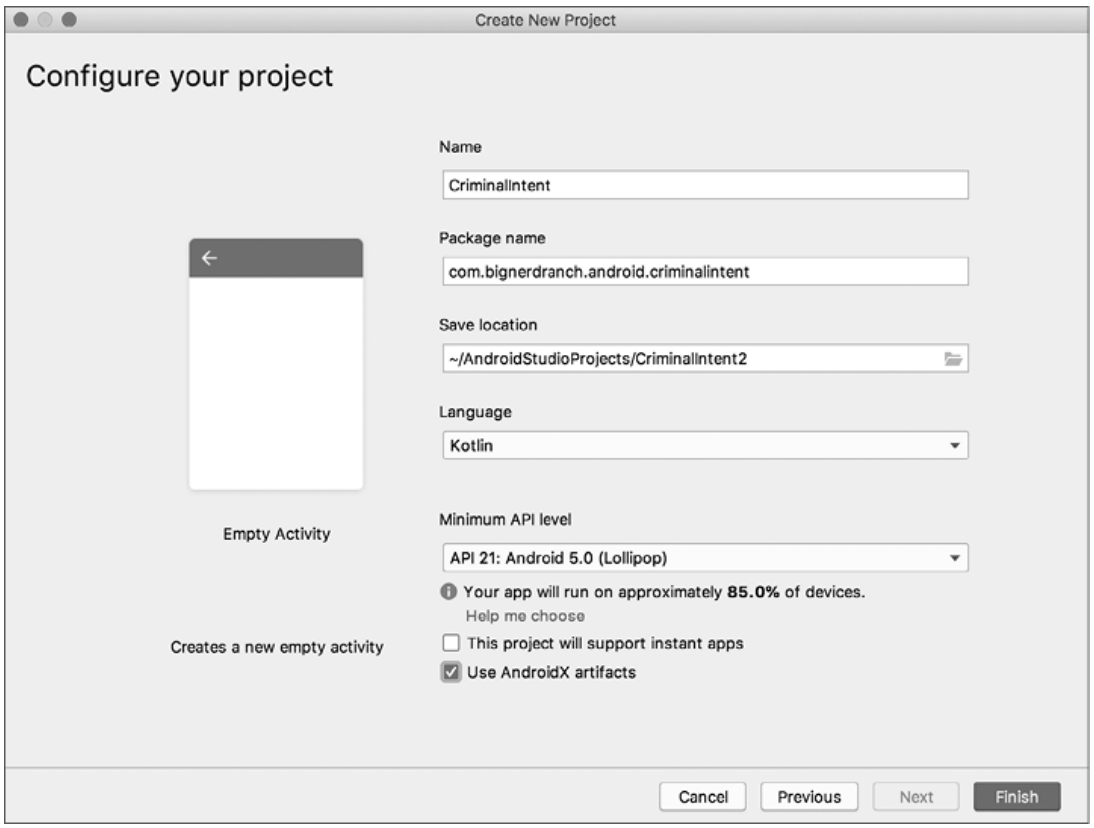


Рис. 8.8. Настройки проекта CriminalIntent

Листинг 8.1. Добавление в класс Crime (Crime.kt)

```
data class Crime(val id: UUID = UUID.randomUUID(),
                 var title: String = "",
```

```
        var date: Date = Date(),
        var isSolved: Boolean = false)
```

При импорте `Date` вам будет предложено несколько вариантов. Обязательно импортируйте `java.util.Date`.

`UUID` — вспомогательный класс Java, входящий в инфраструктуру Android, — предоставляет простой способ генерирования универсально-уникальных идентификаторов. В конструкторе такой идентификатор генерируется вызовом `UUID.randomUUID()`.

Инициализация переменной `Date` с помощью конструктора `Date` по умолчанию задает текущую дату. Это будет дата преступления по умолчанию.

Это все, что потребуется для класса `Crime` и слоя модели CriminalIntent.

На этом этапе вы создали слой модели и activity, включающую в себя фрагмент. Теперь поговорим о том, как эта activity выполняет функцию хоста.

Создание UI-фрагмента

Последовательность действий при создании UI-фрагмента не отличается от последовательности действий при создании activity:

- разработка интерфейса посредством определения виджетов в файле макета;
- создание класса и назначение макета, который был определен ранее, его представлением;
- подключение виджетов, заполненных на основании макета в коде.

Определение макета CrimeFragment

Представление `CrimeFragment` будет отображать информацию, содержащуюся в экземпляре `Crime`.

Начнем с определения строк, которые увидит пользователь, в файле `res/values/strings.xml`.

Листинг 8.2. Добавление строк (res/values/strings.xml)

```
<resources>
    <string name="app_name">CriminalIntent</string>
    <string name="crime_title_hint">Enter a title for the crime.</string>
    <string name="crime_title_label">Title</string>
    <string name="crime_details_label">Details</string>
    <string name="crime_solved_label">Solved</string>
</resources>
```

Затем определяется пользовательский интерфейс. Макет представления `CrimeFragment` состоит из вертикального элемента `LinearLayout`, содержащего два виджета `TextView`, виджеты `EditText`, `Button` и `Checkbox`.

Чтобы создать файл макета, щелкните правой кнопкой мыши по папке `res/layout` на панели **Project** и выберите команду **New⇒Layoutresourcefile** в контекстном меню. Присвойте файлу фрагмента имя **fragment_crime.xml**. Выберите корневым элементом `LinearLayout` и нажмите кнопку **OK**; Android Studio сгенерирует файл самостоятельно.

Когда файл откроется, перейдите к разметке XML. Мастер уже добавил элемент `LinearLayout` за вас. Добавьте виджеты, образующие макет фрагмента, в файл `res/layout/fragment_crime.xml` (рис. 8.3).

Листинг 8.3. Файл макета для представления фрагмента (res/layout/fragment_crime.xml)

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="16dp">

    <TextView
                                        style="?android:listSeparatorTextViewStyle"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/crime_title_label"/>

    <EditText
            android:id="@+id/crime_title"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
```

```
            android:hint="@string/crime_title_h
int"/>

    <TextView
                                            style="?
android:listSeparatorTextViewStyle"
            android:layout_width="match_parent"
            android:layout_height="wrap_content
"
            android:text="@string/crime_details
_label"/>

    <Button
            android:id="@+id/crime_date"
            android:layout_width="match_parent"
            android:layout_height="wrap_content
"
              tools:text="Wed Nov 14 11:56 EST
2018"/>

    <CheckBox
            android:id="@+id/crime_solved"
            android:layout_width="match_parent"
            android:layout_height="wrap_content
"
            android:text="@string/crime_solved_
label"/>

</LinearLayout>
```

(В первом определении TextView введен новый синтаксис, связанный со стилем просмотра: style="?

`android:listSeparatorTextViewStyle"`. Не бойтесь. Смысл этого синтаксиса вы узнаете в разделе «Стили, темы и атрибуты тем» в главе 10.)

Вспомните, что пространство имен `tools` позволяет предоставить информацию для предварительного просмотра. Нужно добавить к кнопке текст, чтобы в окне предварительного просмотра не было пустоты. Перейдите на вкладку **Design**, чтобы включить предварительный просмотр вашего фрагмента (рис. 8.9).

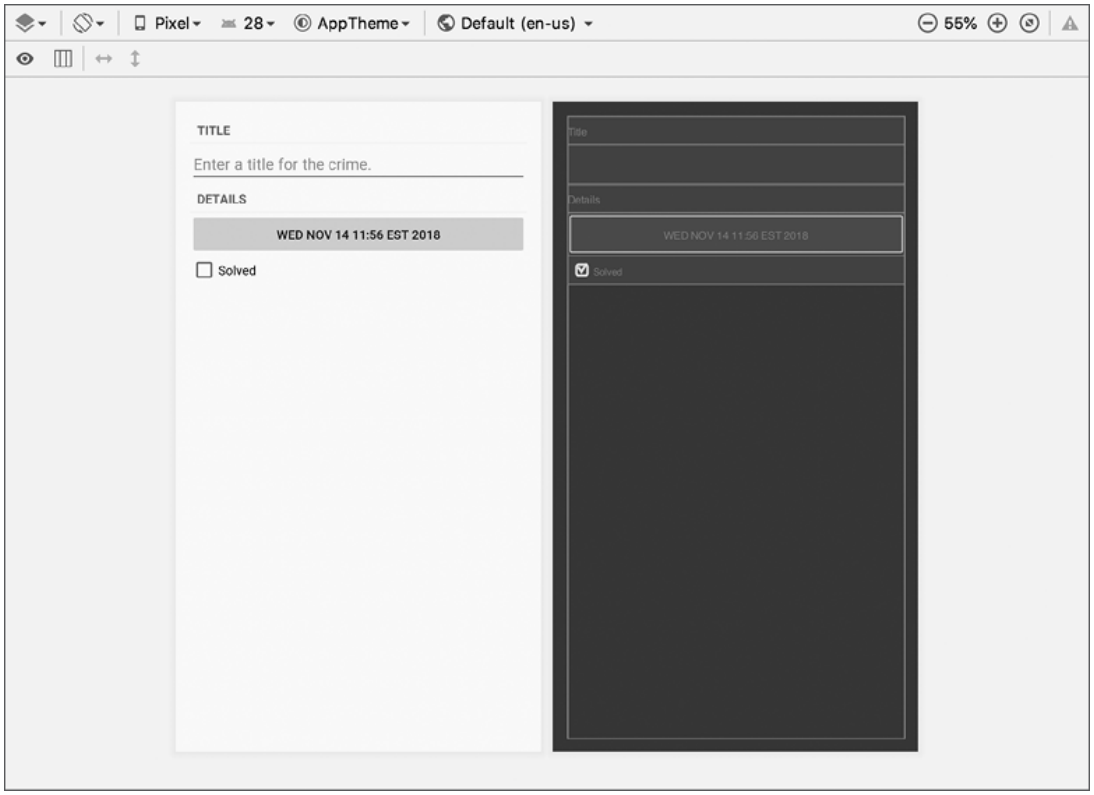


Рис. 8.9. Предварительный просмотр макета фрагмента

Создание класса CrimeFragment

Создадим еще один Kotlin-файл для класса `CrimeFragment`. На этот раз выберите вариант **Class**, после чего Android Studio создаст для вас пустое определение класса. Теперь этот класс

нужно преобразовать во фрагмент. Превратите класс во фрагмент, создав подкласс класса `Fragment`.

Листинг 8.4. Подклассирование класса Fragment (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {
}
```

При подклассировании `Fragment` вы заметите, что Android Studio находит два класса с именем `Fragment`: `android.app.Fragment` и `androidx.fragment.app.Fragment`. Класс `android.app.Fragment` реализует версию фрагментов, встроенную в ОС Android. Мы используем версию библиотеки поддержки, поэтому выберите пункт `androidx.fragment.app.Fragment` (рис. 8.10). (Помните, что Jetpack-библиотеки находятся в пакетах, имена которых начинаются со слова `android`.)

Рис. 8.10. Выбор класса Fragment из Jetpack-библиотеки

Если вы не видите этого диалога, щелкните мышью по имени класса `Fragment`. Если диалог все равно не появляется, можно вручную импортировать нужный класс: добавьте строку `importandroidx.fragment.app.Fragment` в верхнюю часть файла.

Если же у вас импортирована библиотека `android.app.Fragment`, удалите эту строку кода. Затем

импортируйте корректный класс `Fragment` с помощью ⌥+⏎ (Alt+Enter).

Различные типы фрагментов

Новые приложения для Android всегда должны собираться с использованием версии фрагментов Jetpack (`androidx`). Если вы поддерживаете старые приложения, вы увидите две другие версии: версию фреймворка и версию библиотеки поддержки v4. Это старые версии класса `Fragment`, и вы должны рассмотреть возможность перевода приложений, которые их используют, на текущую версию Jetpack.

Фрагменты были введены в API уровня 11 вместе с первыми планшетами Android, когда появилась необходимость обеспечения гибкости пользовательского интерфейса. Реализация фреймворков фрагментов уже была встроена в устройства, работающие на API уровня 11 и выше. Вскоре после этого в библиотеку поддержки v4 была добавлена реализация класса `Fragment` для включения поддержки фрагментов на старых устройствах. С каждой новой версией Android в обе эти версии фрагментов добавлялись новые возможности и исправления безопасности.

Но начиная с Android 9.0 (API 28) фреймворк-версия фрагментов устарела. Никаких дальнейших обновлений этой версии производиться не будет, поэтому не стоит использовать ее для новых проектов. Также фрагменты ранее поддерживаемых библиотек были перенесены в Jetpack-библиотеки. Никаких дальнейших обновлений библиотек поддержки после версии 28 больше не будет. Все последующие обновления будут относиться к версии Jetpack, а не к фреймворку или фрагментам поддержки v4.

Итак: всегда используйте фрагменты Jetpack в новых проектах и переносите существующие проекты, чтобы они оставались актуальными и могли следовать за обновлениями.

Реализация функций жизненного цикла фрагмента

`CrimeFragment` — контроллер, взаимодействующий с объектами модели и представления. Его задача — выдача подробной информации о конкретном преступлении и ее обновление при модификации пользователем.

В приложении GeoQuiz activity выполняли большую часть работы контроллера в функциях жизненного цикла. В приложении CriminalIntent эта работа будет выполняться фрагментами в функциях жизненного цикла фрагментов. Многие из этих функций соответствуют уже известным вам функциям `Activity`, таким как `onCreate(Bundle?)`.

(Подробнее о жизненном цикле фрагмента вы узнаете позже в этой главе.)

В файле `CrimeFragment.kt` добавьте переменную для экземпляра `Crime` и реализацию `Fragment.onCreate(Bundle?)`.

Android Studio может немного помочь с переопределением функций. Когда вы определяете функцию `onCreate(Bundle?)`, введите несколько начальных символов имени функции там, где она должна размещаться. Android Studio выдает список рекомендаций (рис. 8.11).

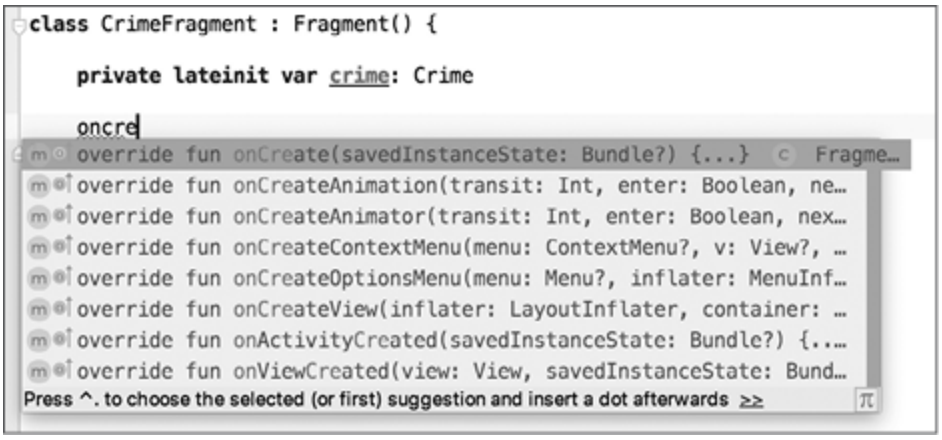


Рис. 8.11. Переопределение функции onCreate(Bundle?)

Нажмите клавишу ⏎ (**Enter**), чтобы выбрать функцию `onCreate(Bundle?)`; Android Studio сгенерирует объявление функции за вас, добавив туда вызов реализации суперкласса. Внесите изменения в свой код, который должен создавать новый объект `Crime` (листинг 8.5).

Листинг 8.5. Переопределение Fragment.onCreate(Bundle?) (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {

    private lateinit var crime: Crime

    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        crime = Crime()
    }
}
```

В этой реализации стоит обратить внимание на пару моментов. Во-первых, функция

`Fragment.onCreate(Bundle?)` объявлена открытой, а функции Kotlin по умолчанию объявляются публичными, если при объявлении не указан модификатор видимости, тогда как функция `Activity.onCreate(Bundle?)` защищена. `Fragment.onCreate(Bundle?)` и другие функции жизненного цикла `Fragment` должны быть открытыми, потому что они будут вызываться произвольной activity, которая станет хостом фрагмента.

Во-вторых, как и в случае с activity, фрагмент использует объект `Bundle` для сохранения и загрузки состояния. Вы можете переопределить `Fragment.onSaveInstanceState(Bundle)` для ваших целей, как и функцию `Activity.onSaveInstanceState(Bundle)`.

Обратите внимание на то, что *не* происходит во `Fragment.onCreate(Bundle?)`: мы не заполняем представление фрагмента. Экземпляр фрагмента настраивается во `Fragment.onCreate(Bundle?)`, но создание и настройка представления фрагмента осуществляются в другой функции жизненного цикла фрагмента: `onCreateView(LayoutInflater,ViewGroup?,Bundle?)`.

Именно в этой функции заполняется макет представления фрагмента, а заполненный объект `View` возвращается хост-activity. Параметры `LayoutInflater` и `ViewGroup` необходимы для заполнения макета. Объект `Bundle` содержит данные, которые используются функцией для воссоздания представления по сохраненному состоянию.

В файле `CrimeFragment.kt` добавьте реализацию `onCreateView(...)`, которая заполняет разметку `fragment_crime.xml`. Для заполнения объявления функции можно воспользоваться приемом, показанным на рис. 8.11.

Листинг 8.6. Переопределение onCreateView(...) (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {

    private lateinit var crime: Crime

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        crime = Crime()
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                                val    view    =
inflater.inflate(R.layout.fragment_crime,
container, false)
        return view
    }
}
```

В функции `onCreateView(...)` мы явно заполняем представление фрагмента, вызывая `LayoutInflater.inflate(...)` с передачей идентификатора ресурса макета. Второй параметр определяет родителя представления, что обычно необходимо для правильной настройки виджета. Третий параметр указывает, нужно ли включать заполненное представление в родителя. Мы передаем

false, потому что представление будет добавлено в контейнере activity. Представление фрагмента не нужно сразу добавлять в родительское представление — activity обработает этот момент позже.

Подключение виджетов во фрагменте

Теперь мы займемся подключением EditText, Checkbox и Button во фрагменте. Это следует делать в функции onCreateView(...).

Начните с EditText. После того как представление будет заполнено, функция получит ссылку на EditText с помощью findViewById.

Листинг 8.7. Настройка виджета EditText (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {
    private lateinit var crime: Crime
    private lateinit var titleField: EditText
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                                val     view     =
inflater.inflate(R.layout.fragment_crime,
container, false)
                                  titleField     =
view.findViewById(R.id.crime_title) as EditText
        return view
    }
```

```
}
```

Получение ссылок на `Fragment.onCreateView(...)` происходит практически так же, как в `Activity.onCreate(...)`. Единственное различие заключается в том, что для представления фрагмента вызывается функция `View.findViewById(int)`. Функция `Activity.findViewById(int)`, которую мы использовали ранее, является вспомогательной функцией, вызывающей `View.findViewById(int)` в своей внутренней реализации. У класса `Fragment` аналогичной вспомогательной функции нет, поэтому приходится вызывать основную функцию.

Когда ссылка установлена, нужно добавить слушателя в обратном вызове `onStart()`.

Листинг 8.8. Добавление слушателя к виджету EditText (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        ...
    }

    override fun onStart() {
        super.onStart()
```

```
        val titleWatcher = object : TextWatcher
{

            override fun beforeTextChanged(
                sequence: CharSequence?,
                start: Int,
                count: Int,
                after: Int
            ) {
                  // Это пространство оставлено
пустым специально
            }

            override fun onTextChanged(
                sequence: CharSequence?,
                start: Int,
                before: Int,
                count: Int
            ) {
                                crime.title  =
sequence.toString()
            }

                                override  fun
afterTextChanged(sequence: Editable?) {
                // И это
            }
        }

        titleField.addTextChangedListener(title
Watcher)
```

```
        }
    }
```

Назначение слушателей во фрагменте работает точно так же, как в activity. В листинге 8.8 мы создаем анонимный класс, который реализует интерфейс слушателя `TextWatcher`. Этот интерфейс содержит три функции, но нас интересует только одна: `onTextChanged(...)`.

В функции `onTextChanged(...)` мы вызываем `toString()` для объекта `CharSequence`, представляющего ввод пользователя. Эта функция возвращает строку, которая затем используется для задания заголовка `Crime`.

Обратите внимание, что слушатель `TextWatcher` настраивается в функции `onStart()`. Некоторые слушатели срабатывают не только при взаимодействии с ними, но и при восстановлении состояния виджета, например при повороте. Слушатели, которые реагируют на ввод данных, такие как `TextWatcher` для `EditText` или `OnCheckChangedListener` для `CheckBox`, тоже так работают.

Слушатели, которые реагируют только на взаимодействие с пользователем, такие как `OnClickListener`, не восприимчивы к такому поведению, поскольку на них не влияет настройка данных в виджете. Именно по этой причине вы не столкнулись с подобной ситуацией в GeoQuiz. В этом приложении слушатели устанавливались только на клики, то есть они не срабатывают при повороте, поэтому вы могли настроить все в `onCreate(...)` — перед любым восстановлением состояния.

Состояние виджета восстанавливается после функции `onCreateView(...)` и перед функцией `onStart()`. При восстановлении состояния содержимое `EditText` будет установлено на любое значение, которое в данный момент находится в заголовке `crime.title`. В этот момент, если вы

уже установили слушателя на EditText (например, в onCreate(...) или onCreateView(...)), будут выполняться функции TextWatcher-а beforeTextChanged(...), onTextChanged(...) и afterTextChanged(...). Установка слушателя в onStart() позволяет избежать такого поведения, так как слушатель подключается после восстановления состояния виджета.

Затем виджет Button настраивается для отображения даты (листинг 8.9).

Листинг 8.9. Настройка текста Button (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {
    private lateinit var crime: Crime
    private lateinit var titleField: EditText
    private lateinit var dateButton: Button
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                                val view =
inflater.inflate(R.layout.fragment_crime,
container, false)

                                titleField =
view.findViewById(R.id.crime_title) as EditText
                                dateButton =
view.findViewById(R.id.crime_date) as Button

        dateButton.apply {
```

```
            text = crime.date.toString()
            isEnabled = false
        }

        return view
    }
}
```

Блокировка кнопки гарантирует, что кнопка никак не среагирует на нажатие. Кроме того, внешний вид кнопки изменяется в заблокированном состоянии. В главе 13 мы снова сделаем кнопку доступной, а пользователь получит возможность выбрать дату преступления.

Переходим к `CheckBox`: получите ссылку `onCreateView(...)` и назначьте слушателя в `onStart()`, который обновит поле `solvedCheckBox` объекта `Crime`, как показано в листинге 8.10. Несмотря на то что функция `OnClickListener` не вызвана восстановлением состояния фрагмента, размещение на `OnStart` помогает удерживать всех слушателей в одном месте и легко находить их.

Листинг 8.10. Прослушивание изменений Checkbox (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {

    private lateinit var crime: Crime
    private lateinit var titleField: EditText
    private lateinit var dateButton: Button
        private lateinit var solvedCheckBox:
CheckBox
    ...
```

```
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                                val    view    =
inflater.inflate(R.layout.fragment_crime,
container, false)

                                titleField    =
view.findViewById(R.id.crime_title) as EditText
                                dateButton    =
view.findViewById(R.id.crime_date) as Button
                            solvedCheckBox    =
view.findViewById(R.id.crime_solved)        as
CheckBox
        ...
    }

    override fun onStart() {
        ...
        titleField.addTextChangedListener(title
Watcher)

        solvedCheckBox.apply {
setOnCheckedChangeListener { _, isChecked ->
                crime.isSolved = isChecked
            }
        }
    }
}
```

Код CrimeFragment готов. Было бы замечательно, если бы вы могли немедленно запустить CriminalIntent и поэкспериментировать с написанным кодом. К сожалению, это невозможно — фрагменты не могут самостоятельно выводить свои представления на экран. Сначала необходимо добавить CrimeFragment в MainActivity.

Хостинг UI-фрагментов

Чтобы стать хостом для UI-фрагмента, activity должна:

- определить место представления фрагмента в своем макете;
- управлять жизненным циклом экземпляра фрагмента.

Вы можете прикрепить фрагменты к своей activity в коде. Вы определяете, когда фрагмент будет добавлен в activity и что с ним произойдет после этого. Можно удалить фрагмент, заменить его другим, а затем снова добавить первый фрагмент.

Подробности кода будут приведены ниже. Сначала вы определите компоновку MainActivity.

Определение контейнерного представления

Мы добавим UI-фрагмент в код хост-activity, но нужно найти место для представления фрагмента в иерархии представлений activity. Найдите макет MainActivity в файле res/layout/activity_main.xml.

Откройте этот файл и замените дефолтный макет кодом FrameLayout. XML-код показан в листинге 8.11.

Листинг 8.11. Создание макета контейнера фрагмента (activity_crime.xml)

```
    <androidx.constraintlayout.widget.ConstraintLayout
            xmlns:android="http://schemas.android.com/apk/res/android"
            xmlns:tools="http://schemas.android.com/tools"
            xmlns:app="http://schemas.android.com/apk/res-auto"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            tools:context=".MainActivity">

        <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Hello World!"
                app:layout_constraintBottom_toBottomOf="parent"
                app:layout_constraintLeft_toLeftOf="parent"
                app:layout_constraintRight_toRightOf="parent"
                app:layout_constraintTop_toTopOf="parent"/>

    </androidx.constraintlayout.widget.ConstraintLayout>
<FrameLayout
```

```
        xmlns:android="http://schemas.android.com
    /apk/res/android"
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
```

Элемент `FrameLayout` станет *контейнерным представлением* для `CrimeFragment`. Обратите внимание: контейнерное представление абсолютно универсально; оно не привязывается к классу `CrimeFragment`. Мы можем (и будем) использовать один макет для хостинга разных фрагментов.

Хотя `activity_crime.xml` состоит исключительно из контейнерного представления одного фрагмента, макет activity может быть более сложным; он может определять несколько контейнерных представлений, а также собственные виджеты.

Запустите CriminalIntent, чтобы проверить ваш код. Под панелью приложения вы увидите пустой `FrameLayout`, содержащий имя приложения (рис. 8.12).

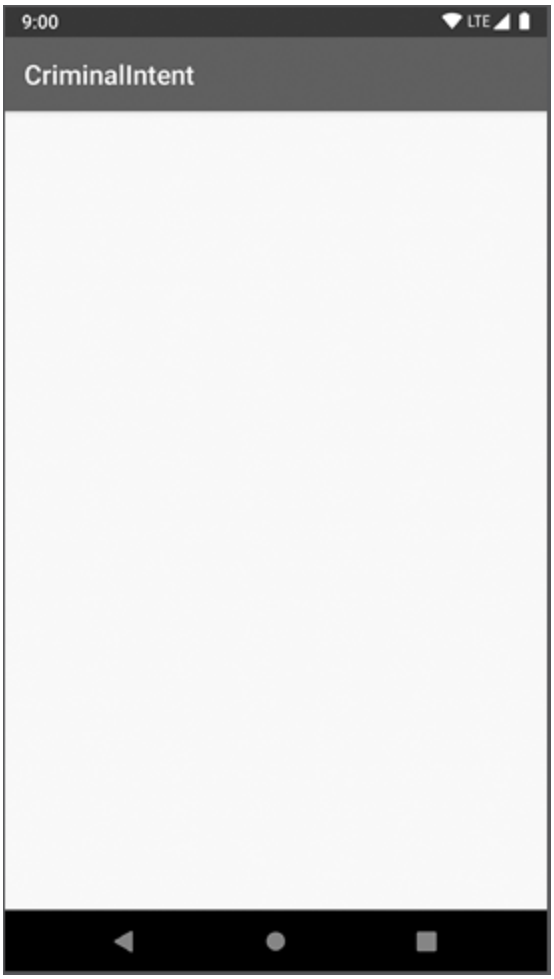


Рис. 8.12. Пустой элемент FrameLayout

Элемент `FrameLayout` пуст, потому что в `MainActivity` еще не отображается фрагмент. Позднее мы напишем код, который помещает представление фрагмента во `FrameLayout`. Но сначала фрагмент необходимо создать.

(Панель приложения в верхней части интерфейса создается автоматически из-за настроек activity. О панели приложения будет более подробно рассказано в главе 14.)

Добавление UI-фрагмента во FragmentManager

Когда в Honeycomb появился класс `Fragment`, в класс `Activity` были внесены изменения: в него был добавлен

компонент, называемый FragmentManager. FragmentManager работает с двумя вещами: списком фрагментов и обратным стеком транзакций (о котором вы скоро узнаете) (рис. 8.13). Он отвечает за добавление представлений фрагментов в иерархию представлений activity и управление жизненными циклами фрагментов.

В приложении CriminalIntent нас интересует только список фрагментов FragmentManager.

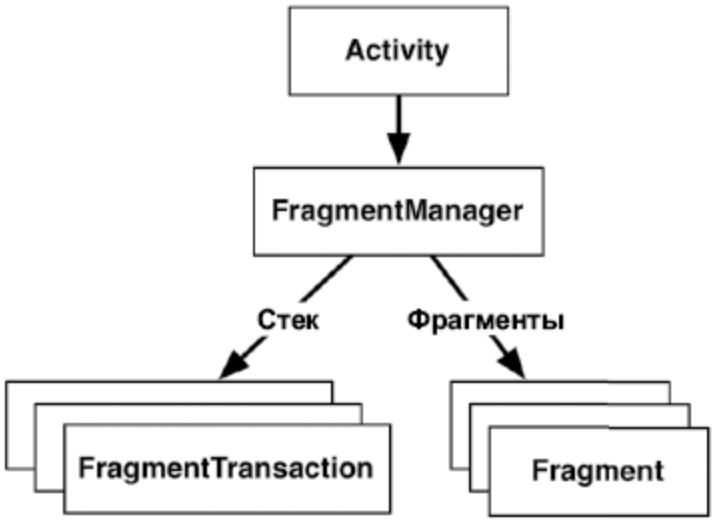


Рис. 8.13. Компонент FragmentManager

Транзакции фрагментов

После получения объекта FragmentManager добавьте следующий код, который передает ему фрагмент для управления. (Позднее мы рассмотрим этот код более подробно, а пока просто включите его в приложение.)

Листинг 8.12. Добавление CrimeFragment (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val currentFragment =
            supportFragmentManager.findFragment
ById(R.id.fragment_container)

        if (currentFragment == null) {
            val fragment = CrimeFragment()
            supportFragmentManager
                .beginTransaction()
                  .add(R.id.fragment_container,
fragment)
                .commit()
        }
    }
}
```

Для добавления фрагмента в код activity выполняются явные вызовы во `FragmentManager` activity. Доступ к менеджеру фрагментов осуществляется с помощью свойства `supportFragmentManager`. Мы будем использовать `supportFragmentManager`, так как используем Jetpack-библиотеку и класс `AppCompatActivity`. У имени есть префикс support, так как свойство возникло в библиотеке поддержки v4, но с тех пор библиотека поддержки была перенесена в Jetpack как библиотека `androidx`.

Разбираться в остальном коде, добавленном в листинг 8.12, лучше всего не с начала. Найдите операцию `add(...)` и

окружающий ее код. Этот код создает и закрепляет *транзакцию фрагмента*.

```
if (currentFragment == null) {
    val fragment = CrimeFragment()
        supportFragmentManager
            .beginTransaction()
                .add(R.id.fragment_container,
fragment)
            .commit()
}
```

Транзакции фрагментов используются для добавления, удаления, присоединения, отсоединения и замены фрагментов в списке фрагментов.

Они позволяют объединять операции в группы, например добавлять несколько фрагментов в разные контейнеры. Они лежат в основе механизма использования фрагментов для формирования и модификации экранов во время выполнения.

`FragmentManager` ведет стек транзакций, по которому вы можете перемещаться. Если в транзакции присутствует несколько операций, то при удалении транзакции из обратного стека их порядок реверсируется. Это позволяет лучше контролировать состояние пользовательского интерфейса при группировании операций с фрагментами в одну транзакцию.

Функция `FragmentManager.beginTransaction()` создает и возвращает экземпляр `FragmentTransaction`. Класс `FragmentTransaction` использует *динамичныйинтерфейс*: функции, настраивающие `FragmentTransaction`, возвращают `FragmentTransaction` вместо `Unit`, что позволяет объединять их вызовы в цепочку. Таким образом, выделенный код в приведенном выше листинге означает: «Создать новую

транзакцию фрагмента, включить в нее одну операцию add, а затем закрепить».

Функция `add(...)` отвечает за основное содержание транзакции. Она получает два параметра: идентификатор контейнерного представления и недавно созданный объект `CrimeFragment`. Идентификатор контейнерного представления должен быть вам знаком: это идентификатор ресурса элемента `FrameLayout`, определенного в файле `activity_crime.xml`.

Идентификатор контейнерного представления выполняет две функции:

- сообщает `FragmentManager`, где в представлении activity должно находиться представление фрагмента;
- обеспечивает однозначную идентификацию фрагмента в списке `FragmentManager`.

Когда вам потребуется получить экземпляр `CrimeFragment` от `FragmentManager`, запросите его по идентификатору контейнерного представления.

```
    val currentFragment =
        supportFragmentManager.findFragmentById
(R.id.fragment_container)

if (currentFragment == null) {
    val fragment = CrimeFragment()
    supportFragmentManager
        .beginTransaction()
        .add(R.id.fragment_container, fragment)
        .commit()
```

```
    }
```

Может показаться странным, что FragmentManager идентифицирует CrimeFragment по идентификатору ресурса FrameLayout. Однако идентификация UI-фрагмента по идентификатору ресурса его контейнерного представления встроена в механизм работы FragmentManager. Если вы добавляете в activity несколько фрагментов, то обычно для каждого фрагмента создается отдельный контейнер со своим идентификатором.

Теперь мы можем кратко описать код, добавленный в листинг 8.12, от начала до конца.

Сначала у FragmentManager запрашивается фрагмент с идентификатором контейнерного представления R.id.fragmentContainer. Если этот фрагмент уже находится в списке, FragmentManager возвращает его.

Почему фрагмент может уже находиться в списке? Вызов MainActivity.onCreate (Bundle?) может быть выполнен в ответ на *воссоздание* объекта MainActivity после его уничтожения из-за поворота устройства или освобождения памяти. При уничтожении activity ее экземпляр FragmentManager сохраняет список фрагментов. При воссоздании activity новый экземпляр FragmentManager загружает список и воссоздает хранящиеся в нем фрагменты, чтобы все работало как прежде.

С другой стороны, если фрагменты с заданным идентификатором контейнерного представления отсутствуют, значение fragment равно null. В этом случае мы создаем новый экземпляр CrimeFragment и новую транзакцию, которая добавляет фрагмент в список.

Теперь MainActivity — это хост для CrimeFragment. Чтобы убедиться в этом, запустите приложение CriminalIntent.

На экране отображается представление, определенное в файле `fragment_crime.xml` (рис. 8.14).

Рис. 8.14. MainActivity — хост представления CrimeFragment

FragmentManager и жизненный цикл фрагмента

На рис. 8.15 показан жизненный цикл фрагмента. Он аналогичен жизненному циклу activity: у него есть состояния остановки, паузы и выполнения, а также функции, которые можно переопределить, чтобы сделать что-то в критических точках, многие из которых соответствуют функциям жизненного цикла activity.

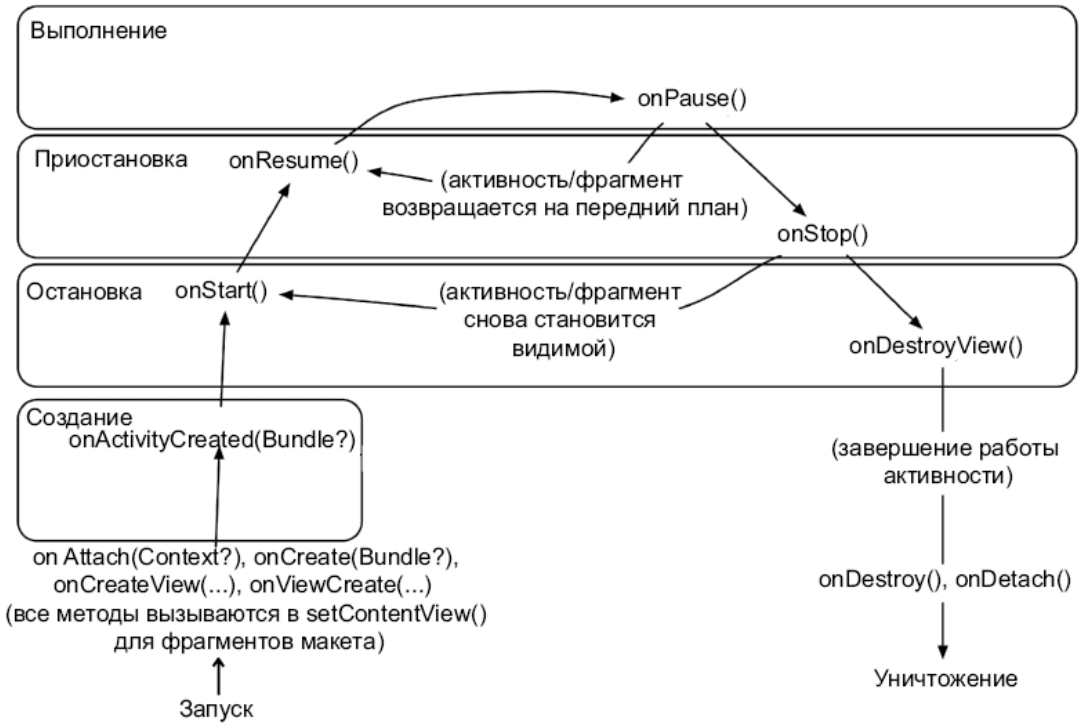


Рис. 8.15. Жизненный цикл фрагмента

Тут важна последовательность. Поскольку фрагмент работает от имени activity, его состояние должно отражать состояние activity. Таким образом, для работы с ним необходимы соответствующие функции жизненного цикла.

Важное различие между жизненным циклом фрагмента и жизненным циклом activity заключается в том, что функции жизненного цикла фрагмента вызываются не операционной системой, а диспетчером хостинга, выполняющим эти функции. Операционная система ничего не знает о фрагментах, которые она использует для управления. Фрагменты — это внутренняя кухня activity. Функции `OnAttach(Context?)`, `onCreate(Bundle?)`, `onCreateView(...)` и

onViewCreated(...) вызываются при добавлении фрагмента во FragmentManager.

Функция onActivityCreated(Bundle) вызывается после выполнения функции onCreate(Bundle?) хост-activity. Мы добавляем CrimeFragment в MainActivity.onCreate(Bundle?), так что эта функция будет вызываться после добавления фрагмента.

Что произойдет, если добавить фрагмент в то время, когда activity уже находится в состоянии остановки, приостановки или выполнения? В этом случае FragmentManager немедленно проводит фрагмент через все действия, необходимые для его согласования с состоянием activity. Например, при добавлении фрагмента в activity, уже находящуюся в состоянии выполнения, фрагмент получит вызовы onAttach(Context), onCreate(Bundle?), onCreateView(...), onViewCreated(...), onActivityCreated(Bundle), onStart() и затем onResume().

После того как состояние фрагмента будет согласовано с состоянием activity, объект FragmentManager хост-activity будет вызывать дальнейшие функции жизненного цикла приблизительно одновременно с получением соответствующих вызовов от ОС для синхронизации состояния фрагмента с состоянием activity.

Архитектура приложений с фрагментами

При разработке приложений с фрагментами очень важно правильно подойти к проектированию. Многие разработчики после первого знакомства с фрагментами пытаются применять их ко всем компонентам приложения, подходящим для повторного использования. Такой способ использования фрагментов ошибочен.

Фрагменты предназначены для инкапсуляции основных компонентов для повторного использования. В данном случае основные компоненты находятся на уровне всего экрана приложения. Если на экран одновременно выводится большое количество фрагментов, ваш код замусоривается транзакциями фрагментов и неочевидными обязанностями. С точки зрения архитектуры существует другой, более правильный способ организации повторного использования вторичных компонентов — выделение их в специализированное представление (класс, являющийся подклассом `View` или один из его подклассов).

Пользуйтесь фрагментами осмотрительно. На практике не рекомендуется отображать более двух или трех фрагментов одновременно (рис. 8.16).

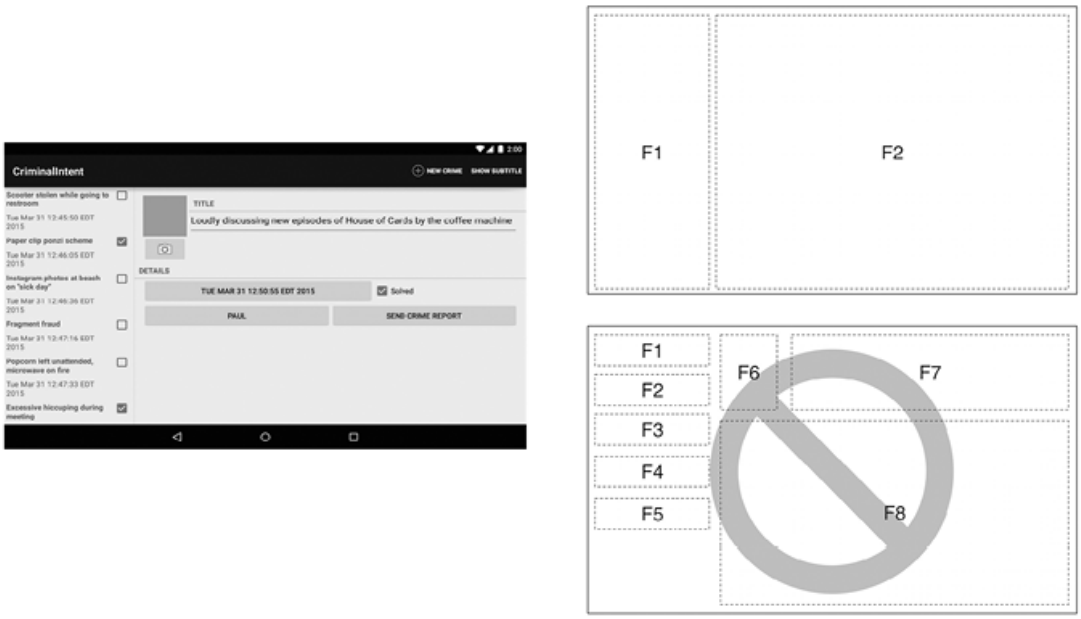


Рис. 8.16. Не гонитесь за количеством

Нужно ли использовать фрагменты

Фрагменты являются предметом множества дискуссий в сообществе Android. Некоторые люди считают, что использование фрагментов и их жизненный цикл создают лишние сложности, и поэтому не используют их в своих проектах. Проблема тут в том, что в Android есть несколько API, использующих фрагменты, например `ViewPager` и Jetpack-библиотека Navigation. Если вам нужно использовать эти опции в вашем приложении, то используйте фрагменты, чтобы извлечь максимум выгоды.

Если вам не нужны фрагментозависимые API, то фрагменты будут полезны в больших приложениях с большими требованиями. Для простых одноэкранных приложений это будет излишняя сложность.

При запуске нового приложения необходимо учитывать, что добавление фрагментов сейчас может создать проблемы в будущем. Заменить activity на activity с фрагментом несложно, но возникает куча проблем. Если одни интерфейсы управляются activity, а другие — фрагментами, это только усугубляет ситуацию, так как приходится держать это в уме. Гораздо проще писать код, сразу используя фрагменты, и не думать о сложностях будущих переделок, не вспоминать, какой стиль контроллера вы используете в каждой части вашего приложения.

Поэтому когда речь идет о фрагментах, мы придерживаемся другого принципа: почти всегда используем фрагменты. Если вы знаете, что ваше будущее приложение будет небольшим, то сложности от использования фрагментов могут оказаться нецелесообразными, поэтому их можно избежать. В более крупных приложениях сложность внедрения компенсируется гибкостью, которую позволяют получить фрагменты, и их использование обоснованно.

Поэтому в некоторых приложениях в этой книге мы будем использовать фрагменты, а в некоторых — нет. В одной главе мы сделаем несколько небольших приложений, и без фрагментов они реализуются проще. Однако большие приложения в этой книге лучше работают с фрагментами, и мы покажем их механизм, чтобы вы умели работать с ними в будущем.

9. Вывод списков и RecyclerView

Уровень модели CriminalIntent в настоящее время состоит из единственного экземпляра `Crime`. В этой главе мы обновим приложение CriminalIntent, чтобы оно поддерживало списки. В списке для каждого преступления будут отображаться краткое описание и дата, а также признак его раскрытия (рис. 9.1).

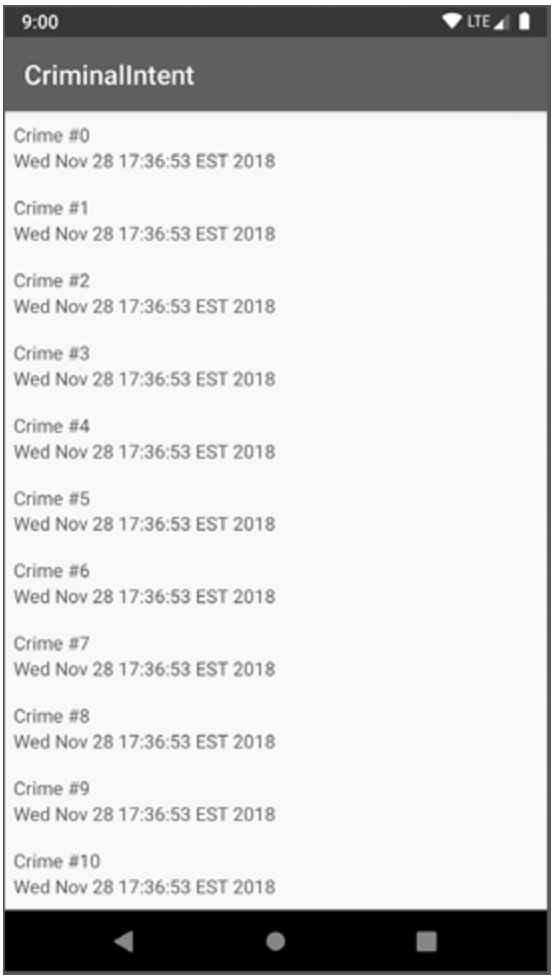


Рис. 9.1. Список преступлений

На рис. 9.2 показана общая структура приложения CriminalIntent для этой главы.

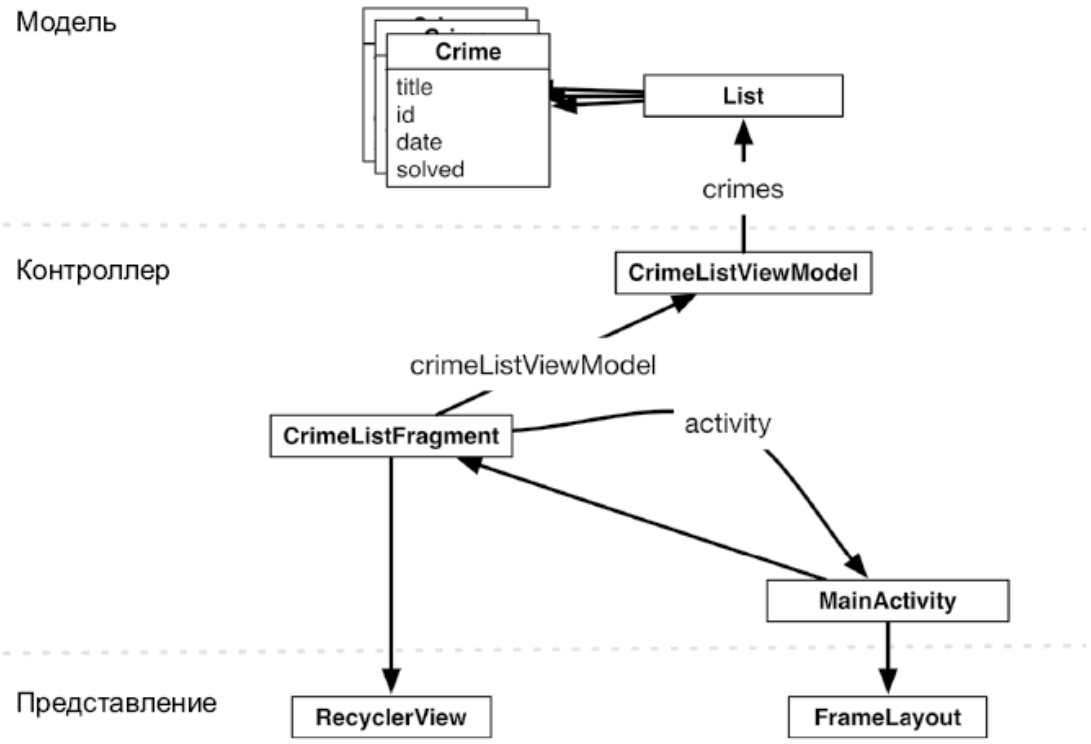


Рис. 9.2. Приложение CriminalIntent со списком

На уровне модели появляется новый объект `ViewModel`, который инкапсулирует данные для нового экрана. `CrimeListViewModel` представляет собой централизованное хранилище для объектов `Crime`.

Для отображения списка необходим новый фрагмент на уровне контроллера CriminalIntent — `CrimeListFragment`. `MainActivity` содержит экземпляр `CrimeListFragment`, который отображает список преступлений на экране.

(Где находится класс `CrimeFragment` на рис. 9.2? Он является частью представления детализации, поэтому на рисунке его нет. В главе 12 мы свяжем части списка и детализации CriminalIntent.)

На рис. 9.2 также видны объекты представлений, связанные с `MainActivity` и `CrimeListFragment`. Представление activity состоит из объекта `FrameLayout`, содержащего фрагмент. Представление фрагмента состоит из

RecyclerView. Класс RecyclerView более подробно рассматривается позднее в этой главе.

Добавление нового фрагмента и ViewModel

В первую очередь добавим новый ViewModel для хранения списка (List) объектов Crime, который будет отображаться на экране. Как вы узнали из главы 4, класс ViewModel входит в состав библиотеки lifecycle-extensions. Поэтому начнем с добавления зависимости lifecycle-extensions в файл app/build.gradle.

Листинг 9.1. Добавление зависимости lifecycle-extensions (app/build.gradle)

```
dependencies {
    ...
                                    implementation
'androidx.appcompat:appcompat:1.1.0-alpha02'
          implementation 'androidx.core:core-
ktx:1.1.0-alpha04'
                                    implementation
'androidx.lifecycle:lifecycle-extensions:2.0.0'
    ...
}
```

Не забудьте синхронизировать файлы Gradle после внесения изменений.

Теперь создадим новый класс Kotlin под названием CrimeListViewModel. Обновите новый класс CrimeListViewModel, чтобы он расширял ViewModel.

Добавьте свойство для хранения списка преступлений. В блоке `init` заполните список фиктивными данными.

Листинг 9.2. Генерирование преступлений (CrimeListViewModel.kt)

```
class CrimeListViewModel : ViewModel() {

    val crimes = mutableListOf<Crime>()

    init {
        for (i in 0 until 100) {
            val crime = Crime()
            crime.title = "Crime #$i"
            crime.isSolved = i % 2 == 0
            crimes += crime
        }
    }
}
```

В результате объект `List` будет содержать созданные пользователем преступления, которые можно будет сохранить и загрузить заново. На данный момент мы заполнили список из 100 объектов.

Объект `CrimeListViewModel` не подходит для долговременного хранения данных, но он не инкапсулирует все данные, необходимые для заполнения представления `CrimeListFragment`. В главе 11 вы узнаете больше о долгосрочном хранении при обновлении CriminalIntent для хранения списка преступлений в базе данных.

Следующим шагом будет добавление нового класса `CrimeListFragment` и его привязка к `CrimeListViewModel`,

Создайте класс `CrimeListFragment` и сделайте его подклассом `androidx.fragment.app.Fragment`.

Листинг 9.3. Реализация CrimeListFragment (CrimeListFragment.kt)

```
private const val TAG = "CrimeListFragment"

class CrimeListFragment : Fragment() {

            private val crimeListViewModel:
CrimeListViewModel by lazy {
        ViewModelProviders.of(this).get(CrimeLi
stViewModel::class.java)
    }

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
                    Log.d(TAG, "Total crimes:
${crimeListViewModel.crimes.size}")
    }

    companion object {
        fun newInstance(): CrimeListFragment {
            return CrimeListFragment()
        }
    }
}
```

Теперь класс `CrimeListFragment` представляет собой пустую оболочку фрагмента. Он регистрирует количество

преступлений, обнаруженных в классе `CrimeListViewModel`. Разбираться с фрагментом мы будем позже в этой главе.

Хороший метод — создать функцию `newInstance(...)`, которую будут вызывать ваши activity, чтобы получить экземпляр вашего фрагмента. Это похоже на функцию `newIntent()`, которую вы использовали в приложении GeoQuiz. Из главы 12 вы узнаете, как передавать данные фрагментам.

Жизненный цикл ViewModel с фрагментами

В главе 4 мы изучили жизненный цикл `ViewModel` при его использовании с activity. Жизненный цикл будет немного другим, когда `ViewModel` используется с фрагментом. У него по-прежнему есть два состояния, то есть существующее и несуществующее, но теперь он связан с жизненным циклом фрагмента вместо activity.

`ViewModel` будет оставаться активным, пока виджет фрагмента находится на экране. `ViewModel` сохраняется при повороте (даже если экземпляр фрагмента не сохраняется) и будет доступен для нового экземпляра объекта.

`ViewModel` уничтожается после уничтожения фрагмента. Это может произойти, когда пользователь нажимает кнопку «Назад», закрывая экран. Это также может произойти, если хост-activity заменяет фрагмент на другой. Хотя на экране отображается та же activity, и фрагмент, и связанный с ним `ViewModel` будут уничтожены, так как они больше не нужны.

Есть один частный случай — это когда вы добавляете транзакцию проекта обратно в стек. Когда activity заменяет текущий фрагмент другим, а транзакция возвращается в стек, фрагмент и его `ViewModel` уничтожены не будут. Если пользователь нажимает кнопку «Назад», транзакция фрагмента

восстанавливается. Оригинальный экземпляр фрагмента помещается обратно на экран, и все данные ViewModel сохраняются.

Теперь нужно обновить класс MainActivity, чтобы он размещал экземпляр CrimeListFragment вместо CrimeFragment.

Листинг 9.4. Добавление транзакции фрагмента в CrimeListFragment (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        if (currentFragment == null) {
            val fragment =  ~~CrimeFragment()~~
CrimeListFragment.newInstance()
            supportFragmentManager
                .beginTransaction()
                .add(R.id.fragment_container,
fragment)
                .commit()
        }
    }
}
```

Теперь мы жестко закодировали класс MainActivity на отображение CrimeListFragment. В главе 12 мы научим MainActivity выгружать CrimeListFragment и CrimeFragment по требованию, когда пользователь перемещается по приложению.

Запустите CriminalIntent, и вы увидите, что `FrameLayout` из `MainActivity` хостит пустой `CrimeListFragment`, как показано на рис. 9.3.

Найдите вывод **LogCat** для класса `CrimeListFragment`. Вы увидите запись с доступным количеством преступлений:

```
2019-02-25 15:19:39.950 26140-
26140/com.bignerdranch.android.criminalintent
    D/CrimeListFragment: Total crimes: 100
```

Добавление RecyclerView

Мы хотим, чтобы класс `CrimeListFragment` отображал список преступлений пользователя. Для этого мы будем использовать класс `RecyclerView`.

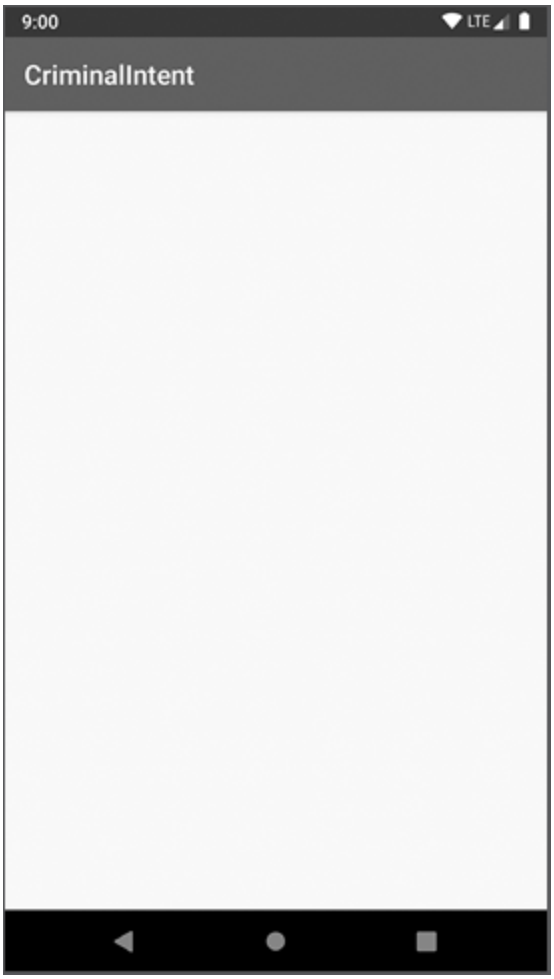


Рис. 9.3. Пустой экран MainActivity

Класс `RecyclerView` находится в другой Jetpack-библиотеке. Начнем с того, что добавим библиотеку `RecyclerView` в зависимости.

Листинг 9.5. Добавление зависимости RecyclerView (app/build.gradle)

```
dependencies {
    ...
                                    implementation
'androidx.lifecycle:lifecycle-extensions:2.0.0'
```

```
                                        implementation
    'androidx.recyclerview:recyclerview:1.0.0'
        ...
    }
```

Опять-таки нужно синхронизировать ваши файлы Gradle перед переходом.

Класс `RecyclerView` будет жить в файле макета `CrimeListFragment`. В первую очередь вы должны создать файл макета. Создайте новый файл макета под именем `fragment_crime_list`, для свойства `Rootelement` задайте значение `androidx.recyclerview.widget.RecyclerView` (рис. 9.4).

В новом файле `res/layout/fragment_crime_list.xml` добавьте идентификатор `RecyclerView`. Сверните закрывающий тег в открывающий тег, так как дочерних классов у `RecyclerView` не будет.

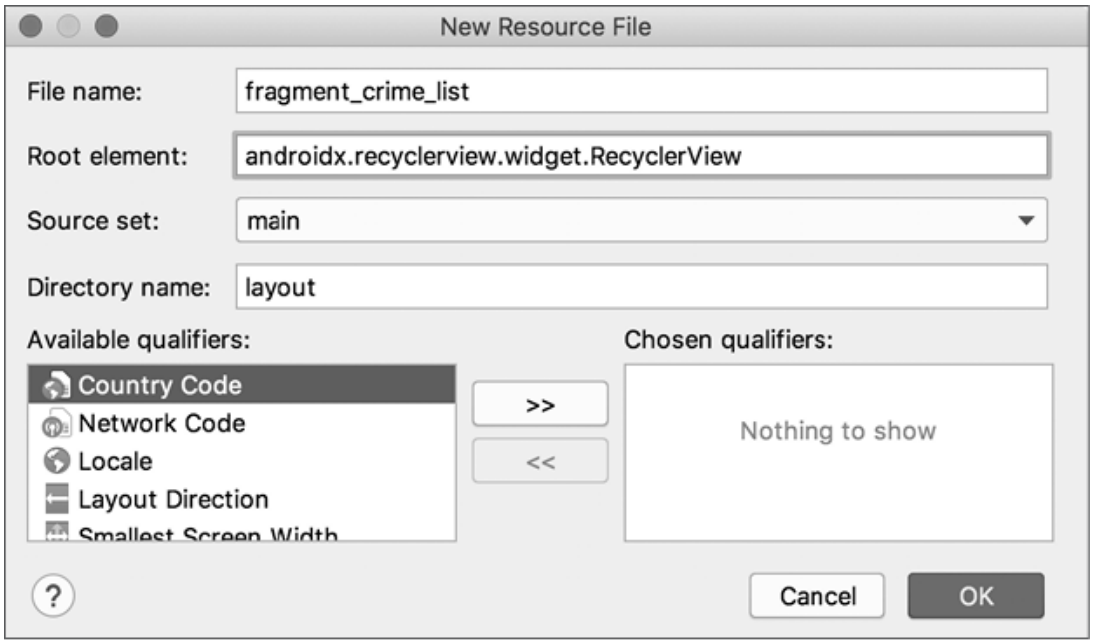


Рис. 9.4. Добавление файла макета CrimeListFragment

Листинг 9.6. Добавление файла макета RecyclerView (res/layout/fragment_crime_list.xml)

```
<androidx.recyclerview.widget.RecyclerView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/crime_recycler_view"
        android:layout_width="match_parent"
            android:layout_height="match_parent">
        android:layout_height="match_parent"/>

    </androidx.recyclerview.widget.RecyclerView>
```

Сейчас, когда виджет `CrimeListFragment` настроен, его нужно подключить к фрагменту. Научите класс `CrimeListFragment` использовать этот файл макета и находить `RecyclerView`, как показано в листинге 9.7.

Листинг 9.7. Настройки виджета CrimeListFragment

```
class CrimeListFragment : Fragment() {

      private lateinit var crimeRecyclerView: RecyclerView

            private val crimeListViewModel: CrimeListViewModel by lazy {
        ViewModelProviders.of(this).get(CrimeListViewModel::class.java)
    }
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
                    Log.d(TAG, "Total crimes:
${crimeListViewModel.crimes.size}")
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                                val view =
inflater.inflate(R.layout.fragment_crime_list,
container, false)

        crimeRecyclerView =
            view.findViewById(R.id.crime_recycle
r_view) as RecyclerView
              crimeRecyclerView.layoutManager =
LinearLayoutManager(context)

        return view
    }
}
```

Обратите внимание, что, как только вы создаете свой RecyclerView, тут же назначаете ему другой объект под названием LayoutManager, который необходим ему для работы.

RecyclerView не отображает элементы на самом экране. Он передает эту задачу объекту LayoutManager.

LayoutManager располагает каждый элемент, а также определяет, как работает прокрутка. Поэтому если RecyclerView пытается сделать что-то подобное при наличии LayoutManager, он сломается.

Доступно несколько встроенных объектов LayoutManager, и вы найдете их еще больше среди сторонних библиотек. Вы используете LinearLayoutManager, которая будет позиционировать элементы в списке по вертикали. Позже в этой книге вы будете использовать GridLayoutManager, чтобы расположить элементы в сетке вместо этого.

Запустите приложение. Вы должны снова увидеть пустой экран, но теперь вы смотрите на пустой RecyclerView.

Создание макета представления элемента

RecyclerView является подклассом ViewGroup. Он отображает список дочерних объектов View, называемых *представлениями элементов*. Это один объект из списка данных представления — утилизатор (в нашем случае одно преступление из списка преступлений). В зависимости от сложности эти дочерние представления могут отображаться сложнее или проще.

В изначальной реализации каждый элемент в списке будет отображать название и дату преступления, как показано на рис. 9.5.

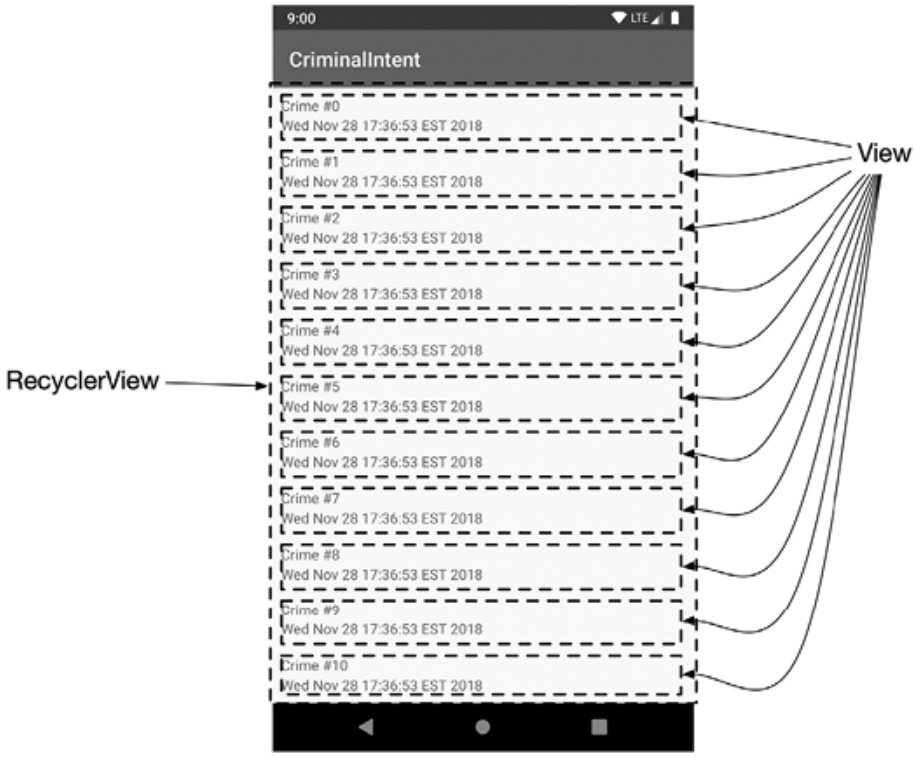


Рис. 9.5. RecyclerView с дочерними View

Каждый элемент, отображаемый в `RecyclerView`, будет иметь свою собственную иерархию, как и `CrimeFragment` имеет иерархию для всего экрана. В частности, объект `View` на каждой строке будет представлять собой `LinearLayout` с двумя `TextView`.

Новый макет для представления списка мы создали так же, как и для представления activity или фрагмента. На панели **Project** щелкните правой кнопкой мыши по папке `res/layout` и выберите команду **New⇒Layoutresourcefile** в контекстном меню. Назовите файл **list_item_crime**, установите корневой элемент `LinearLayout` и нажмите **OK**.

Обновите файл макета, добавив отступы в `LinearLayout` и два `TextView`, как показано в листинге 9.8.

Листинг 9.8. Обновления файла макета для списков (res/layout/list_item_crime.xml)

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
              android:orientation="vertical"
              android:layout_width="match_parent"
              android:layout_height="match_parent">
              android:layout_height="wrap_content"
              android:padding="8dp">

    <TextView
            android:id="@+id/crime_title"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Crime Title"/>

    <TextView
            android:id="@+id/crime_date"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Crime Date"/>

</LinearLayout>
```

Посмотрите на предварительный просмотр в режиме **Design**, и увидите, что вы создали ровно одну строку готового продукта.

Теперь RecyclerView будет создавать новые строки сам.

Реализация ViewHolder

RecyclerView ожидает, что элемент представления будет обернут в экземпляр ViewHolder. ViewHolder хранит ссылку на представление элемента (а иногда и ссылки на конкретные виджеты внутри этого представления).

Определите контейнер для представления, добавив в CrimeListFragment внутренний класс, который расширяется от RecyclerView.ViewHolder.

Листинг 9.9. Начало ViewHolder (CrimeListFragment.kt)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        ...
    }
    private inner class CrimeHolder(view: View)
        : RecyclerView.ViewHolder(view) {
    }
}
```

В конструкторе CrimeHolder мы берем представление для закрепления. Вы сразу же передаете его в качестве аргумента в конструктор классов RecyclerView.ViewHolder. Базовый

класс `ViewHolder` будет закрепляться на свойство под названием `itemView` (рис. 9.6).

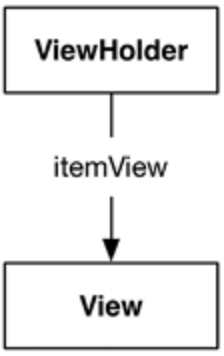


Рис. 9.6. ViewHolder и его itemView

`RecyclerView` никогда не создает объекты `View` сам по себе. Он всегда создает `ViewHolder`, которые выводят свои `itemView` (рис. 9.7).

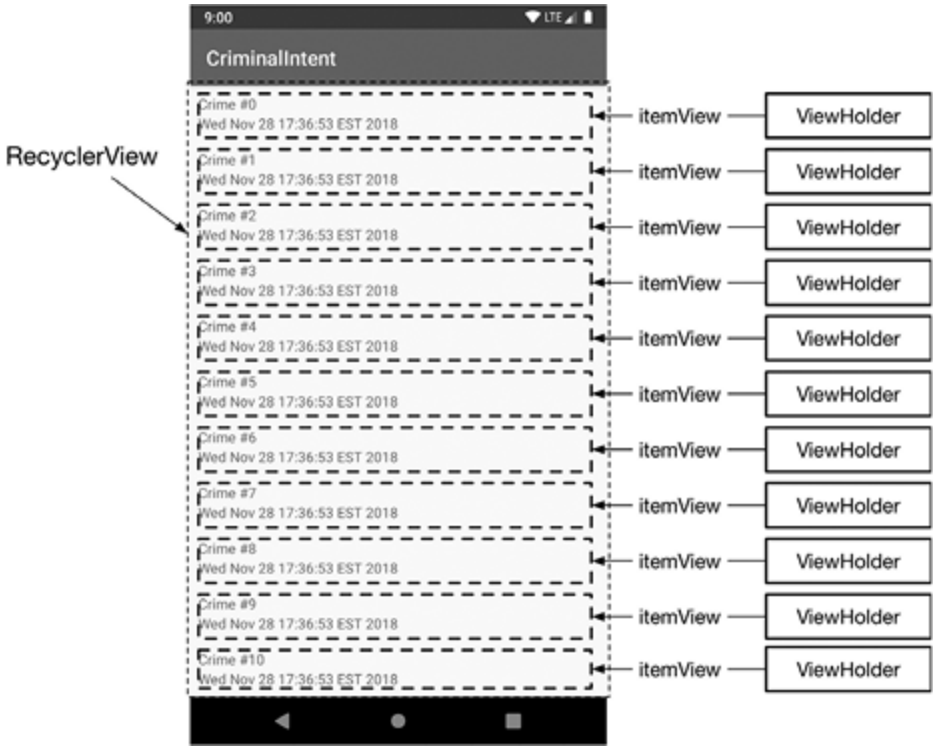


Рис. 9.7. Визуализация ViewHolder

С элементами View все просто, но у ViewHolder есть несколько функций. Для более сложных представлений ViewHolder прикрепляет различные части itemView к Crime более простым и эффективным образом. (К примеру, вам не нужно самому выполнять поиск по иерархии представления элементов, чтобы получать представление текста всякий раз, когда вам нужно настроить заголовок.)

Обновим CrimeHolder, чтобы найти дату и название в иерархии itemView, когда экземпляр впервые создается. Ссылки на текстовые представления хранятся в свойствах.

Листинг 9.10. Извлечение представления в конструктор (CrimeListFragment.kt)

```
private inner class CrimeHolder(view: View)
    : RecyclerView.ViewHolder(view) {

            val titleTextView: TextView =
itemView.findViewById(R.id.crime_title)
            val dateTextView: TextView =
itemView.findViewById(R.id.crime_date)
}
```

Обновленный холдер представления в настоящее время хранит ссылки на названия и даты, так что вы можете легко изменить отображаемое значение через иерархию элемента (рис. 9.8).

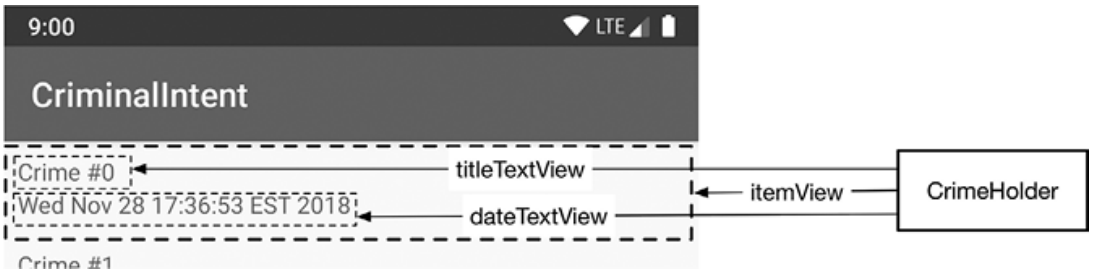


Рис. 9.8. Повторная визуализация ViewHolder

Обратите внимание, что `CrimeHolder` предполагает, что представление, передаваемое в конструктор, имеет дочерние текстовые представления с идентификаторами `R.id.crime_title` и `R.id.crime_date`. Вы можете задать вопрос, кто (или что) создает экземпляры холдеров и можно ли с уверенностью предположить, что иерархия, передаваемая в конструктор, содержит дочерние виджеты. Мы ответим вам через мгновение.

Реализация адаптера для заполнения RecyclerView

На рис. 9.7 все несколько упрощено. Класс `RecyclerView` не создает `ViewHolder` сам по себе. Вместо этого используется *адаптер*. Адаптер представляет собой объект контроллера, который находится между `RecyclerView` и наборами данных, которые отображает `RecyclerView`.

Адаптер выполняет следующие функции:

- создание необходимых `ViewHolder` по запросу;
- связывание `ViewHolder` с данными из модельного слоя.

Утилизатор выполняет следующие функции:

- запрашивает адаптер на создание нового `ViewHolder`;

- запрашивает адаптер привязать `ViewHolder` к элементу данных на этой позиции.

Пришло время создать свой адаптер. Добавим новый внутренний класс `CrimeAdapter` в `CrimeListFragment`. Добавьте первичный конструктор, который получает на вход список преступлений и хранит список прошлых преступлений, как показано в листинге 9.11.

В своем новом `CrimeAdapter` вы также собираетесь переопределить три функции: `onCreateViewHolder(...)`, `onBindViewHolder(...)` и `getItemCount()`. Чтобы пожалеть ваши пальцы, Android Studio сгенерирует эти переопределения сама. Когда вы наберете начальную строку нового кода, поместите курсор на `CrimeAdapter` и нажмите кнопку **⌥+⏎** (**Alt+Enter**). Выберите пункт **Implementmembers** из всплывающего меню.

В диалоговом окне **Implementmembers** выберите все три функции и нажмите кнопку **OK**. После этого вам нужно заполнить все так, как показано ниже.

Листинг 9.11. Создание CrimeAdapter (CrimeListFragment.kt)

```
class CrimeListFragment : Fragment() {
    ...
    private inner class CrimeHolder(view: View)
        : RecyclerView.ViewHolder(view) {
        ...
    }

        private inner class CrimeAdapter(var
crimes: List<Crime>)
```

```
        : RecyclerView.Adapter<CrimeHolder>() {

        override fun onCreateViewHolder(parent:
ViewGroup, viewType: Int)
                : CrimeHolder {
                                val   view   =
layoutInflater.inflate(R.layout.list_item_crime
, parent, false)
            return CrimeHolder(view)
        }

                override  fun  getItemCount()  =
crimes.size

          override fun onBindViewHolder(holder:
CrimeHolder, position: Int) {
            val crime = crimes[position]
            holder.apply {
                            titleTextView.text  =
crime.title
                             dateTextView.text  =
crime.date.toString()
            }
        }
    }
}
```

Функция `Adapter.onCreateViewHolder(...)` отвечает за создание представления на дисплее, оборачивает его в холдер и возвращает результат. В этом случае вы наполняете `list_item_view.xml` и передаете полученное представление в новый экземпляр `CrimeHolder`. (На данный момент вы

можете игнорировать параметры `onCreateViewHolder(...)`. Если вы делаете что-то интересное вроде отображения различных типов представлений в одно представление утилизатора, вам нужны только эти значения. См. упражнение в конце этой главы для получения более подробной информации.)

Функция `Adapter.onBindViewHolder(holder:CrimeHolder,position:Int)` отвечает за заполнение данного холдера `holder` преступлением из данной позиции `position`. В этом случае преступления из списка преступлений окажутся в нужной позиции. Затем вы можете использовать заголовок и данные от этого преступления, чтобы установить текст в соответствующих текстовых представлениях.

Когда утилизатору нужно знать, сколько элементов в наборе данных поддерживают его (например, когда он впервые создается), он будет просить свой адаптер вызвать `Adapter.getItemCount()`. Функция `getItemCount()`возвращает количество элементов в списке преступлений, отвечая на запрос утилизатора.

Сам `RecyclerView` ничего не знает об объекте преступления или перечне преступлений, которые будут отображаться. Зато `CrimeAdapter` знает все данные преступления. Адаптер также знает, какие преступления входят в утилизатор (рис. 9.9).

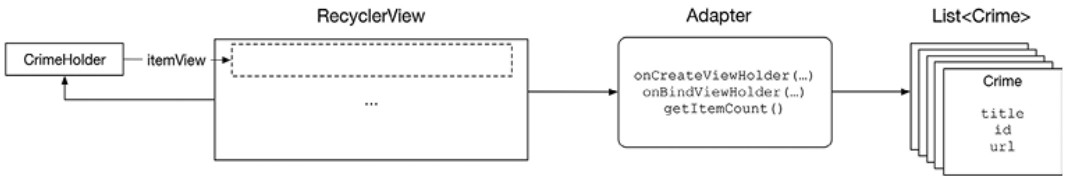


Рис. 9.9. Adapter находится между утилизатором и набором данных

Когда утилизатору требуется объект для отображения, он запрашивает адаптер. На рис. 9.10 показывается пример диалога, который может инициировать `RecyclerView`.

`RecyclerView` вызывает функцию `onCreateViewHolder(ViewGroup,Int)` адаптера для создания нового `ViewHolder` наряду с полезной нагрузкой: `ViewHolder` (и его `itemView`), созданный адаптером и передаваемый в `RecyclerView`, до сих пор не заполнен данными из конкретного элемента в наборе данных.

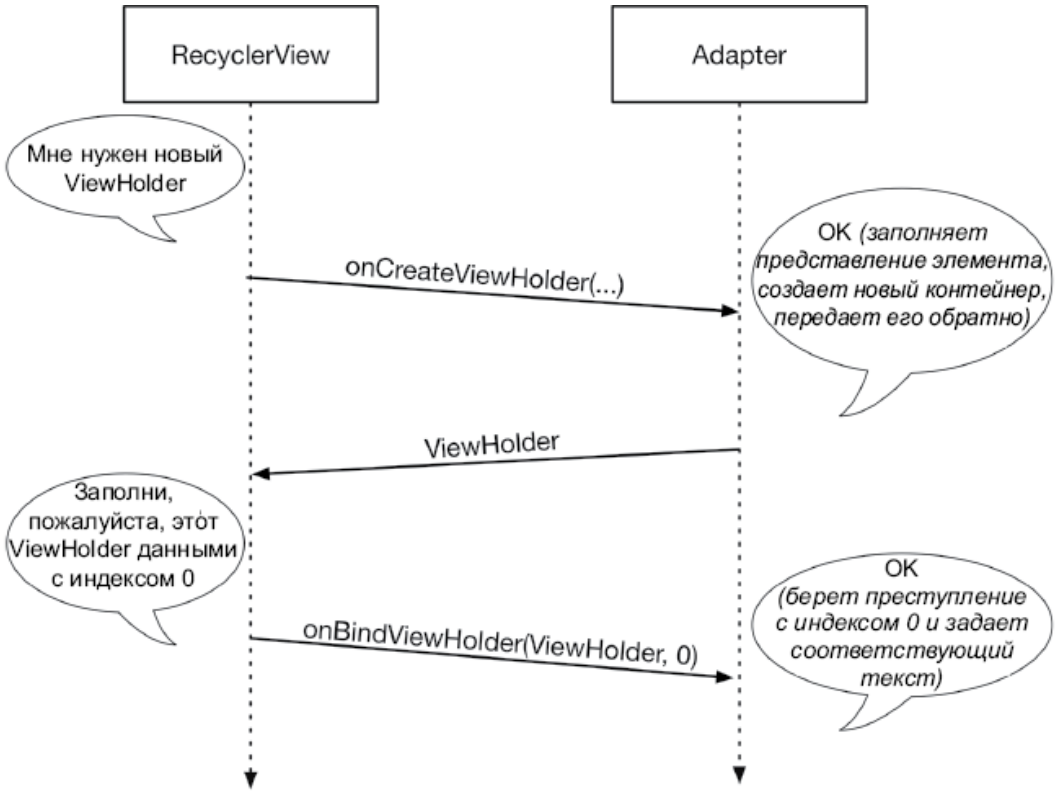


Рис. 9.10. Взаимодействие RecyclerView и Adapter

Наконец, `RecyclerView` вызывает функцию `onBindViewHolder(ViewHolder,Int)`, передает `ViewHolder` в эту функцию вместе с позицией. Адаптер будет искать данные модели для этой позиции и привяжет их к представлению `ViewHolder`. Для привязки адаптер заполняет отображающие

данные в объектной модели. После завершения этого процесса RecyclerView разместит элемент списка на экране.

Настройка адаптера RecyclerView

Когда у вас наконец есть Adapter, подключите его к RecyclerView. Реализуйте функцию под названием updateUI, которая настраивает интерфейс CrimeListFragment. На данный момент она создает CrimeAdapter и устанавливает его на RecyclerView.

Листинг 9.12. Установка объекта Adapter (CrimeListFragment.kt)

```
class CrimeListFragment : Fragment() {

      private lateinit var crimeRecyclerView:
RecyclerView
    private var adapter: CrimeAdapter? = null
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                                val    view    =
inflater.inflate(R.layout.fragment_crime_list,
container, false)

        crimeRecyclerView =
                view.findViewById(R.id.crime_re
cycler_view) as RecyclerView
```

```
                crimeRecyclerView.layoutManager =
LinearLayoutManager(context)

        updateUI()

        return view
    }

    private fun updateUI() {
        val crimes = crimeListViewModel.crimes
        adapter = CrimeAdapter(crimes)
        crimeRecyclerView.adapter = adapter
    }
    ...
}
```

В последующих главах мы модернизируем функцию `updateUI()`, применяя более глубокие настройки интерфейса.

Запустите приложение CriminalIntent и прокрутите новый `RecyclerView`, который должен выглядеть так, как показано на рисунке 9.11.

Потяните экран, и вы увидите еще больше элементов. Каждый видимый `CrimeHolder` выводит отдельно взятое преступление (если строки с преступлениями у вас значительно выше, чем на рисунке, проверьте, что свойство `layout_height` в `LinearLayout` строки установлено на `wrap_content`).

При резкой прокрутке анимация должна идти как по маслу. Этот эффект — прямое следствие того, что событие `onBindViewHolder(...)` выполняет лишь минимальный объем работы. Отсюда вывод: событие `onBindViewHolder(...)` всегда должно быть эффективно. В

противном случае анимация будет выглядеть не как по маслу, а как по засушенному твердому пармезану.

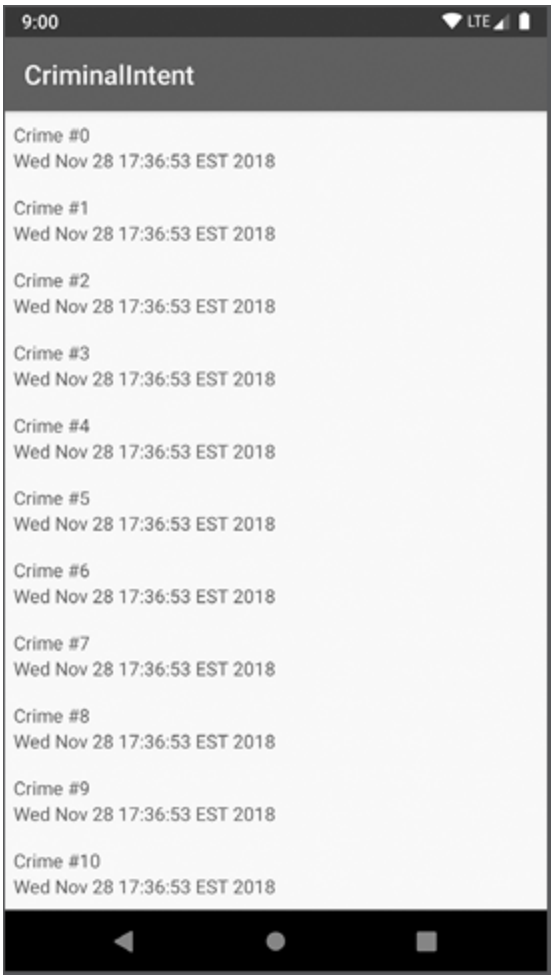


Рис. 9.11. RecyclerView, заполненный преступлениями

Переработка представлений

На рис. 9.11 видно 11 строк. Вы можете просмотреть все 100 преступлений. Значит ли это, что у вас в памяти одновременно находится 100 объектов `View`? Нет, благодаря вашему `RecyclerView`.

Создание представления для каждого элемента в списке — плохая стратегия. Вы наверняка догадываетесь, что в списке

может быть гораздо больше 100 элементов и список элементов может быть сложнее. Кроме того, преступлению нужен объект View только в момент нахождения на экране, поэтому нет необходимости заводить 100 таких. Было бы гораздо больше смысла создавать объекты View только по мере необходимости.

RecyclerView делает именно это. Вместо того чтобы создать 100 объектов View, он создает их столько, сколько нужно для заполнения экрана. Когда представление пропадает с экрана, RecyclerView использует его заново, а не выбрасывает. Тем самым он оправдывает свое название — перерабатывает объекты.

В связи с этим функция onCreateViewHolder(ViewGroup,Int) будет вызываться намного реже, чем onBindViewHolder(ViewHolder,Int). Когда создано достаточно объектов ViewHolder, RecyclerView перестает вызывать onCreateViewHolder(...). Вместо этого он экономит время и память путем утилизации старых объектов ViewHolder и передает их в onBindViewHolder(ViewHolder,Int).

Очистка элементов связанного списка

В данный момент Adapter привязывает данные преступления непосредственно к текстовым виджетам в функции Adapter.onBindViewHolder(...). Это работает отлично, но лучше более четко разделить задачи между холдером и адаптером. Адаптер должен знать как можно больше о внутренней кухне и данных холдера.

Мы рекомендуем поместить весь код, который будет выполнять привязку, внутрь CrimeHolder. Во-первых, нужно добавить свойство для сохранения преступления. И пока вы тут,

сделайте имеющиеся свойства текстового виджета приватными. Добавьте функцию `bind(Crime)` в `CrimeHolder`. В этой новой функции нужно кэшировать привязываемые преступления в свойства и присвоить текстовые значения свойствам `titleTextView` и `dateTextView`.

Листинг 9.13. Функция bind(Crime) (CrimeListFragment.kt)

```
private inner class CrimeHolder(view: View)
    : RecyclerView.ViewHolder(view) {

    private lateinit var crime: Crime

       private val titleTextView: TextView =
itemView.findViewById(R.id.crime_title)
        private val dateTextView: TextView =
itemView.findViewById(R.id.crime_date)

    fun bind(crime: Crime) {
        this.crime = crime
        titleTextView.text = this.crime.title
                            dateTextView.text =
this.crime.date.toString()
    }
}
```

Получив объект `Crime` для привязки, `CrimeHolder` будет обновлять название и дату соответствующего преступления.

Затем мы вызываем новоиспеченную функцию `bind(Crime)` всякий раз, когда `RecyclerView` запрашивает привязку `CrimeHolder` к конкретному преступлению.

Листинг 9.14. Вызов функции bind(Crime) (CrimeListFragment.kt)

```
private inner class CrimeAdapter(var crimes: List<Crime>)
    : RecyclerView.Adapter<CrimeHolder>() {

       override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CrimeHolder {
        ...
    }

        override fun onBindViewHolder(holder: CrimeHolder, position: Int) {
        val crime = crimes[position]
           holder.apply {
                           titleTextView.text = crime.title
                            dateTextView.text = crime.date.toString()
           }
        holder.bind(crime)
    }

    override fun getItemCount() = crimes.size
}
```

Снова запустите CriminalIntent. Результат должен выглядеть так же, как было показано на рис. 9.11.

Отклик на нажатия

В качестве вишенки на торте RecyclerView, приложение CriminalIntent должно реагировать на нажатие на элементы списка. В главе 12 мы создадим подробный вид преступления, который будет появляться, когда пользователь будет нажимать на него в списке. В настоящее время будем просто выводить объект Toast после нажатия.

Как вы, возможно, заметили, RecyclerView, будучи мощным инструментом, выполняет не так-то много работы (пусть это будет примером для всех нас). То же самое здесь: обработка событий касания в основном зависит от вас. Если вам они нужны, RecyclerView может обрабатывать эти события, но это нужно редко.

Вместо этого можно воспользоваться стандартным OnClickListener. Так как каждый View имеет связанный с ним ViewHolder, вы можете создать OnClickListener для всех View. Добавьте в CrimeHolder следующий код.

Листинг 9.15. Отклик на нажатия в CrimeHolder (CrimeListFragment.kt)

```
private inner class CrimeHolder(view: View)
            : RecyclerView.ViewHolder(view),
View.OnClickListener {

    private lateinit var crime: Crime

       private val titleTextView: TextView =
itemView.findViewById(R.id.crime_title)
        private val dateTextView: TextView =
itemView.findViewById(R.id.crime_date)

    init {
```

```
            itemView.setOnClickListener(this)
        }

        fun bind(crime: Crime) {
            this.crime = crime
            titleTextView.text = this.crime.title
                                dateTextView.text   =
    this.crime.date.toString()
        }

        override fun onClick(v: View) {
            Toast.makeText(context, "${crime.title}
    pressed!", Toast.LENGTH_SHORT)
                 .show()
        }
    }
```

В листинге 9.15 `CrimeHolder` сам по себе реализует интерфейс `OnClickListener`. В окне `itemView`, которое представляет собой `View` для всей строки, `CrimeHolder` устанавливается в качестве приемника событий нажатия.

Запустите CriminalIntent и нажмите на пункт в списке. Вы должны увидеть сообщение о нажатии.

Для любознательных: ListView и GridView

Ядро Android OS включает классы `ListView`, `GridView` и `Adapter`. До выхода Android 5.0 это были предпочтительные способы создания списков или сеток элементов.

API для этих компонентов очень похожи на `RecyclerView`. Класс `ListView` или `GridView` отвечает за прокрутку набора

элементов, но не располагает информацией об этих элементах. Адаптер отвечает за создание каждого из изображений в списке. Однако `ListView` и `GridView` не заставляют вас использовать шаблон `ViewHolder` (хотя вы можете и должны делать это).

Эти старые реализации заменены реализацией `RecyclerView` из-за сложности изменения поведения `ListView` или `GridView`.

Создание горизонтальной прокрутки `ListView`, например, не входит в `ListView` API и требует много работы. Создание пользовательского макета и прокрутки с помощью `RecyclerView` — тоже непростая задача, но `RecyclerView` лучше для нее предназначен.

Еще одна ключевая особенность `RecyclerView` — анимация элементов в списке. Анимированное добавление или удаление элементов — это сложная задача, в которой легко допустить ошибку. `RecyclerView` делает процесс намного проще, имеет несколько встроенных анимаций и позволяет легко настраивать их.

Например, если вы обнаружили, что преступление в положении 0 перемещается в положение 5, можно было бы анимировать это вот так:

```
recyclerView.adapter.notifyItemMoved(0, 5)
```

Упражнение. Типы View в RecyclerView

Для этого сложного упражнения вам нужно будет создать два типа строк в вашем `RecyclerView`: для обычных и для более серьезных преступлений. Чтобы это реализовать, вы будете работать с функцией в `RecyclerView.Adapter`. Присвойте новое свойство `requiresPolice` объекту `Crime` и используйте

его, чтобы определить, какой тип `View` загружен в `CrimeAdapter`, путем реализации функции `getItemViewType(Int)` (**developer.android.com/reference/android/support/v7/widget/**`RecyclerView`**.Adapter.html#get**`ItemView`**Type**).

В функции `onCreateViewHolder(ViewGroup,Int)` вам также необходимо добавить логику, которая возвращает различные `ViewHolder` в зависимости от `ViewType`, возвращаемого функцией `getItemViewType(Int)`. Используйте оригинальный макет для преступлений, которые не требуют вмешательства полиции, и новый макет с усовершенствованным интерфейсом, содержащий кнопку с надписью «Связаться с полицией» для серьезных преступлений.

10. Создание интерфейсов с использованием макетов и виджетов

В этой главе мы поближе познакомимся с макетами и виджетами, а также немного украсим список элементов в `RecyclerView`. Вы также узнаете о `ConstraintLayout` — новом механизме управления макетами. На рис. 10.1 показано, как выглядит представление `CrimeListFragment` после того, как текущая версия приложения сделает следующий шаг на пути к совершенству.

В предыдущих главах мы использовали вложенные иерархии расположения виджетов. Например, в файле `res/layout/activity_main.xml`, который вы создали для GeoQuiz в главе 1, один `LinearLayout` был вложен в другой `LinearLayout`. Это вложение трудно найти и трудно редактировать. Хуже того, такое вложение может ухудшить производительность вашего приложения. Вложенные макеты заставляют Android OS тратить много времени на расчеты и размещение, и у пользователей может возникнуть задержка в отображении элементов на экране.

Плоские невложенные макеты быстрее измеряются и выводятся операционной системой. И это одна из областей, где `ConstraintLayout` работает действительно хорошо. Вы можете создавать красивые сложные макеты без использования вложенности.

Прежде чем серьезно браться за `ConstraintLayout`, необходимо провести небольшую подготовку. В проект нужно добавить изображение наручников с рис. 10.1. Откройте файл с примерами ((**www.bignerdranch.com/solutions/AndroidProgramming4e.zip**) и найдите каталог

09_LayoutsAndWidgets/CriminalIntent/app/src/main/res. Скопируйте версии ic_solved.png для разных плотностей пикселов в соответствующие папки drawable в вашем проекте.

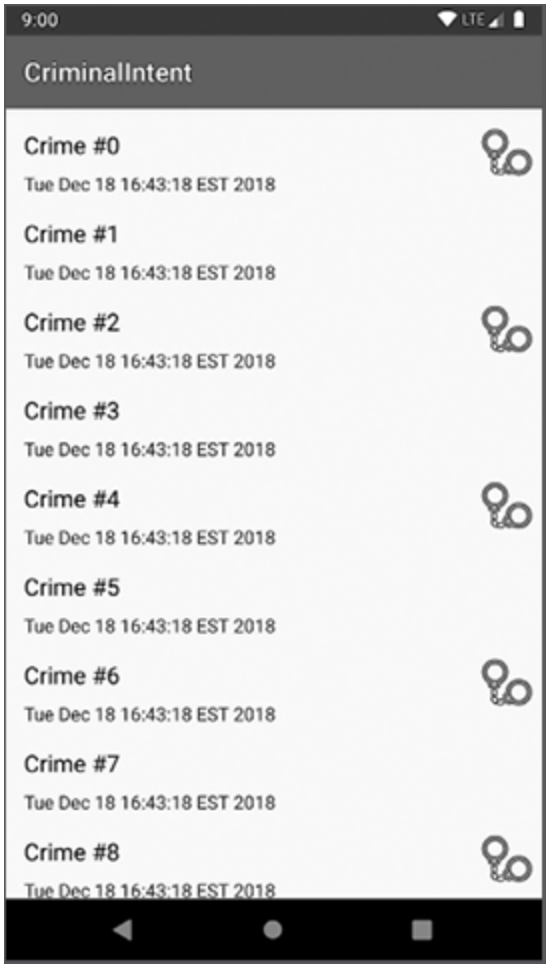


Рис. 10.1. CriminalIntent — теперь с красивыми картинками

Знакомство с ConstraintLayout

При использовании механизма ограничений макета ConstraintLayout вместо применения вложенных макетов в макет включается набор *ограничений*. Ограничение, словно резиновая лента, «притягивает» два визуальных объекта друг к другу. Например, ограничение может связать правый край

виджета ImageView с правым краем родителя (ConstraintLayout), как показано на рис. 10.2. При таком ограничении ImageView будет прилегать к правому краю.

Рис. 10.2. Ограничение связывает ImageView с правым краем

Ограничения можно задать для всех четырех сторон ImageView (левой, верхней, правой и нижней). Противодействующие ограничения компенсируются, и ImageView будет располагаться прямо в центре (рис. 10.3).

Рис. 10.3. ImageView с противоположными ограничениями

Общая картина выглядит так: чтобы разместить представления в нужном месте ConstraintLayout, вы не перетаскиваете их мышью по экрану, а назначаете соответствующие ограничения.

Как насчет определения размеров виджетов? Есть три варианта: предоставить выбор самому виджету (вашему старому знакомому wrap_content), решить самостоятельно или позволить виджету изменять свои размеры в соответствии с ограничениями.

Со всеми этими средствами один контейнер ConstraintLayout может вмещать много разных макетов без применения вложения. В этой главе вы узнаете, как использовать ограничения в файле list_item_crime.xml.

Использование графического конструктора макетов

До настоящего момента мы создавали макеты, вводя разметку XML. В этом разделе мы воспользуемся графическим конструктором.

Откройте файл `list_item_crime.xml` и выберите вкладку **Design** в нижней части окна файла (рис. 10.4).

В центре графического конструктора располагается уже знакомая панель предварительного просмотра. Справа от нее находится область раскладки (*blueprint*) — она выглядит почти так же, как область предварительного просмотра, но в ней отображаются контуры всех представлений (мы уже упоминали об этом в главе 1). Это может быть полезно тогда, когда вы хотите увидеть размеры всех представлений, а не только отображаемую в них информацию.

Слева располагается панель. На ней размещаются практически все виджеты, которые только можно себе представить, упорядоченные по категориям. Слева внизу находится дерево компонентов, которое показывает, как организованы виджеты в макете. Если оно не отображается, понажимайте на вкладки слева от области предварительного просмотра.

Под деревом компонентов располагается панель свойств. На ней можно просматривать и редактировать атрибуты виджета, выделенного в дереве компонентов.

В первую очередь нам надо преобразовать `list_item_crime.xml` для использования `ConstraintLayout`. Щелкните правой кнопкой мыши по корневому узлу `LinearLayout` в дереве компонентов и выберите команду **ConvertLinearLayouttoConstraintLayout** в контекстном меню (рис. 10.5).

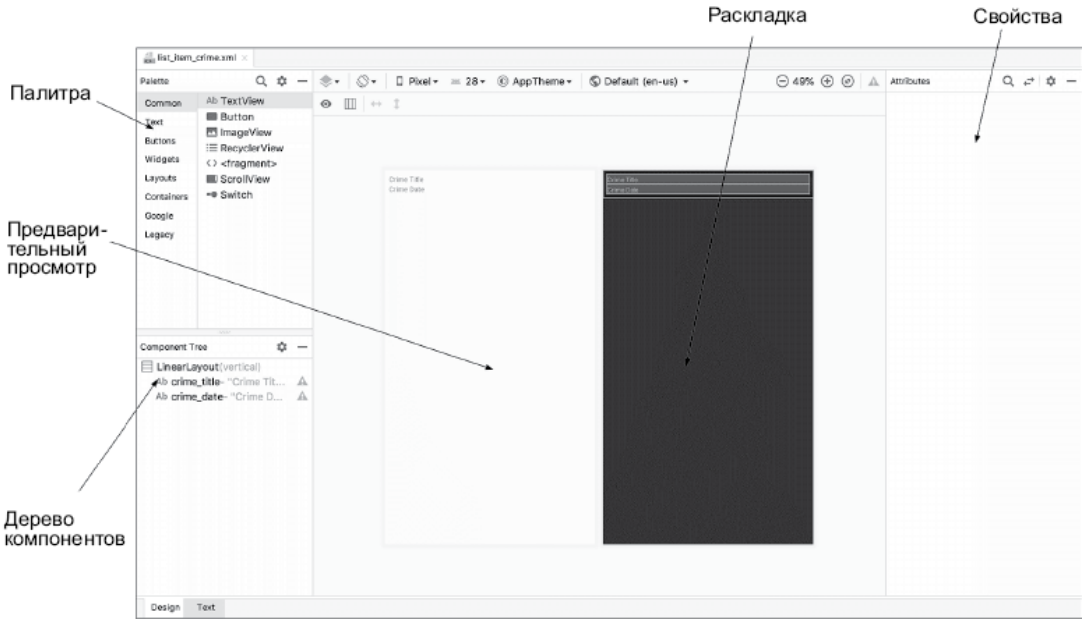


Рис. 10.4. Представления в графическом конструкторе

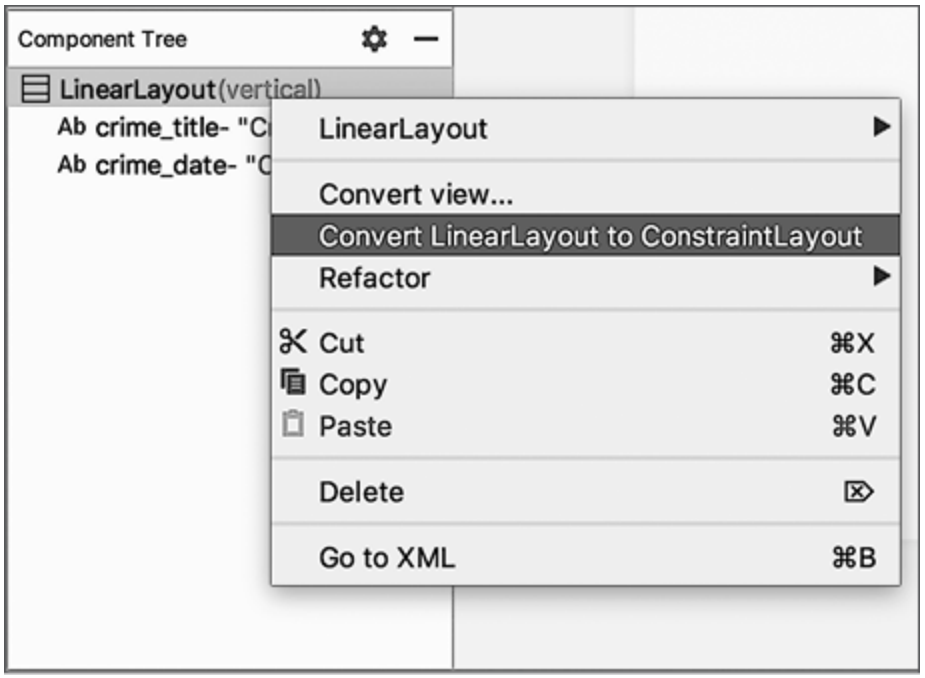


Рис. 10.5. Преобразование корневого представления в ConstraintLayout

Android Studio отобразит диалоговое окно с предложением задать параметры оптимизации преобразования (рис. 10.6). Файл макета `list_item_crime.xml` очень прост, так что у Android Studio немного возможностей для оптимизации. Оставьте значения по умолчанию и нажмите кнопку **OK**.

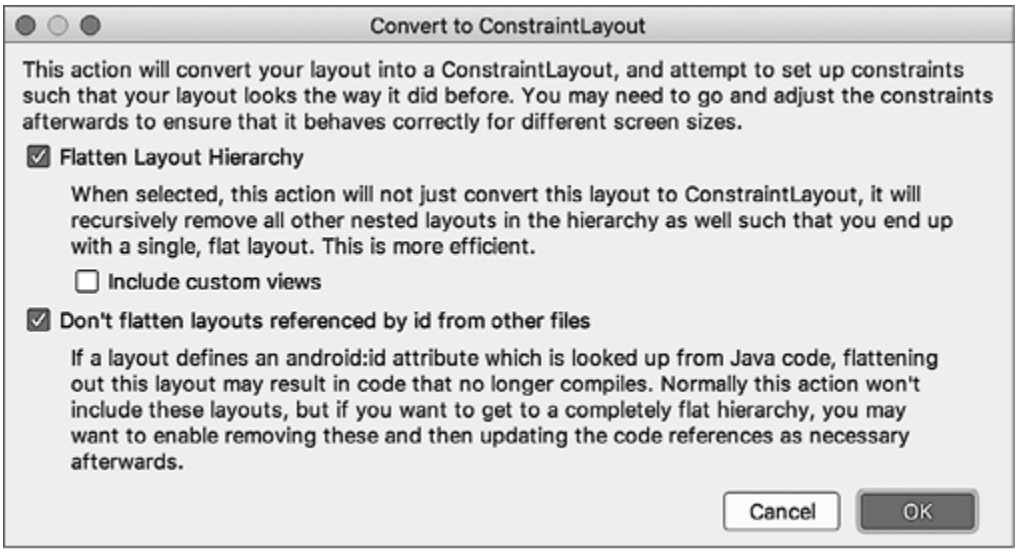


Рис. 10.6. Преобразование с конфигурацией по умолчанию

Наберитесь терпения. Процесс преобразования может занять некоторое время. Но когда он будет завершен, у вас появится новенький, с иголочки `ConstraintLayout` (рис. 10.7).

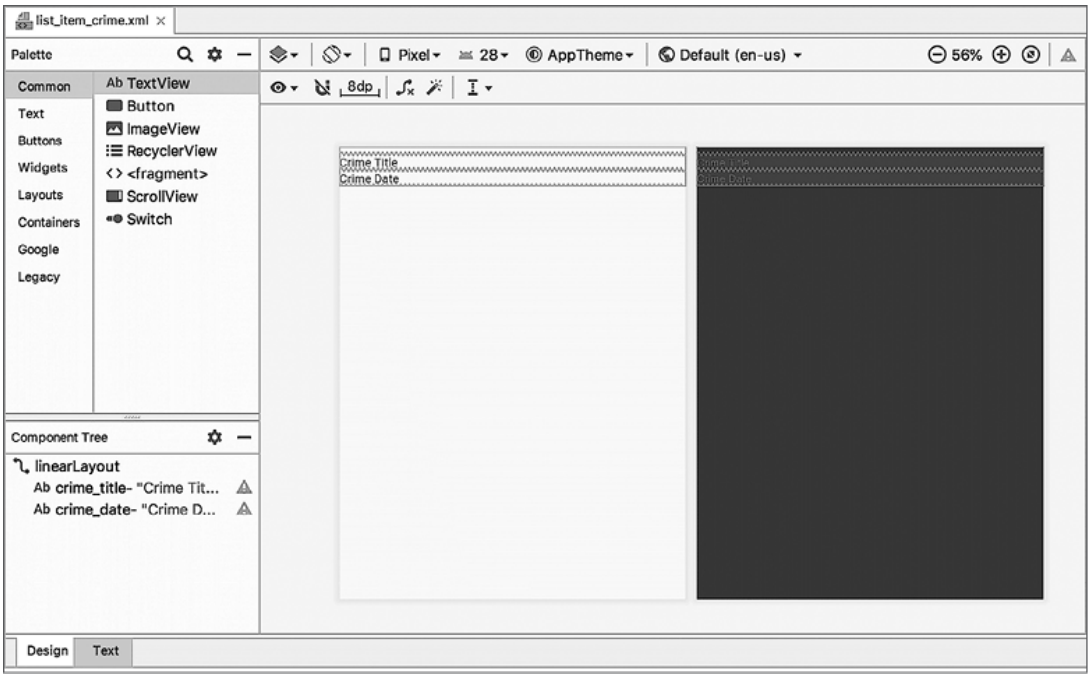


Рис. 10.7. Новый ConstraintLayout

(Не понимаете, откуда в дереве компонентов `linearLayout`? Объясним через мгновение.)

В верхней части области предварительного просмотра находится панель инструментов с несколькими кнопками (рис. 10.8). Щелкните мышью в области предварительного просмотра, чтобы отобразить все кнопки.

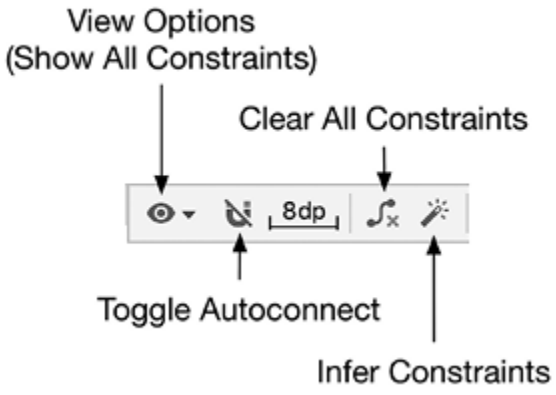


Рис. 10.8. Элементы управления ограничениями

View Options

Команда **ViewOptions⇒ShowAllConstraints** выводит заданные ограничения в режимах предварительного просмотра и раскладки. В одних случаях этот режим просмотра полезен, в других нет. При большом количестве ограничений эта кнопка выводит слишком много информации.

В раскрывающемся списке **View Options** много полезных опций, таких как **Show Layout Decorations**. При выборе опции **Show Layout Decorations** (Показывать декоративные элементы макета) отображается панель приложения, а также некоторые системные интерфейсы (например, строка состояния), которые пользователь видит во время выполнения. Подробнее об этом в главе 14.

Autoconnect

При включении режима Autoconnect ограничения будут автоматически настраиваться при перетаскивании представлений в область предварительного просмотра. Android Studio старается угадать, какие ограничения должно иметь представление, и создавать их по мере необходимости.

Clear All Constraints

Кнопка **Clear All Constraints** удаляет все ограничения, существующие в файле макета. Вскоре мы воспользуемся этой кнопкой.

Infer Constraints

Как и кнопка **Autoconnect**, эта кнопка автоматически создает ограничения, но данная функциональность срабатывает только при нажатии кнопки. Функциональность **Autoconnect** срабатывает каждый раз, когда в файл макета добавляется новое представление.

Использование ConstraintLayout

Когда вы преобразуете файл `list_item_crime.xml` для использования `ConstraintLayout`, среда Android Studio автоматически добавляет ограничения, которые, на ее взгляд, повторяют поведение старого макета. Но чтобы вы лучше поняли, как работают ограничения, мы сделаем все вручную.

Выберите элемент верхнего уровня в дереве компонентов под названием `linearLayout`. Почему на нем написано `linearLayout`, если вы преобразовали его в `ConstraintLayout`? Это идентификатор, который задается конвертером `ConstraintLayout`. `linearLayout` — это, по сути, ваш `ConstraintLayout`. Вы можете проверить XML-версию макета, чтобы убедиться в этом.

Выберите представление `linearLayout` в дереве компонентов, после чего нажмите кнопку **ClearAllConstraints** (рис. 10.8). В правой верхней части экрана немедленно появляется красный флажок. Щелкните мышью по нему, чтобы понять, о чем вас предупреждают (рис. 10.9).

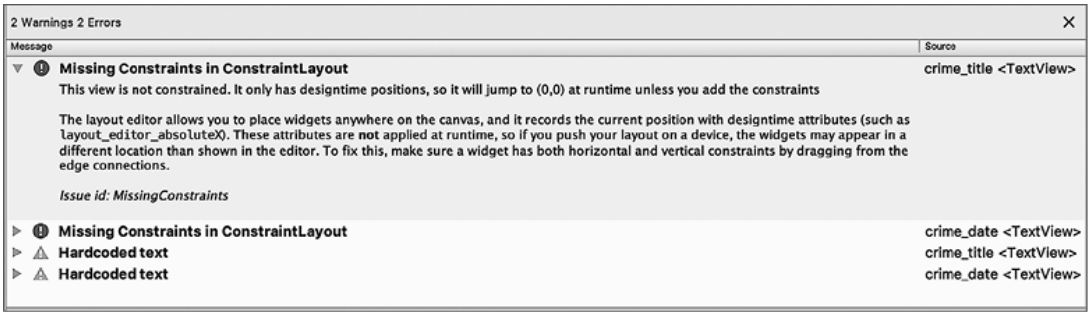


Рис. 10.9. Предупреждения ConstraintLayout

Если заданных ограничений недостаточно, `ConstraintLayout` не сможет точно определить, где разместить представления. Ваши представления `TextView` не имеют ограничений, поэтому для каждого из них выводится предупреждение о том, что оно не будет отображаться в правильном месте во время выполнения.

В этой главе мы добавим необходимые ограничения и избавимся от предупреждений. А во время своей работы следите за предупреждающим индикатором, чтобы избежать непредвиденного поведения во время выполнения.

Освобождение пространства

Сначала необходимо освободить область для размещения — сейчас два представления `TextView` занимают все доступное место, не позволяя разместить что-либо еще. Эти два виджета нужно уменьшить.

Выберите виджет `crime_title` в дереве компонентов и просмотрите его свойства на панели справа (рис. 10.10). Если эта панель скрыта, щелкните мышью по вкладке **Attributes**, чтобы ее открыть.

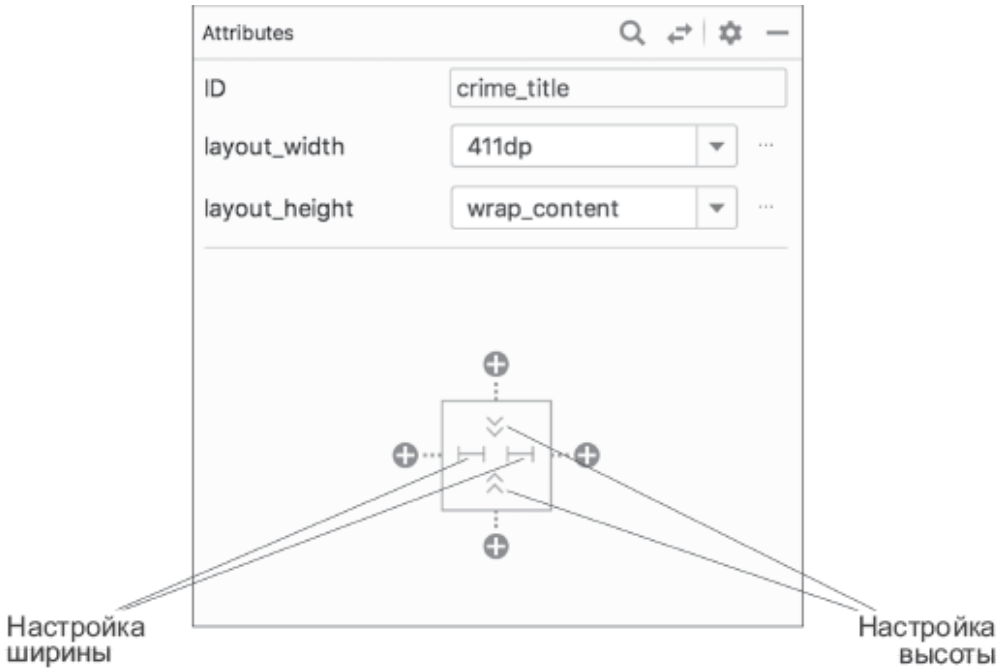


Рис. 10.10. Атрибуты TextView

Вертикальный и горизонтальный размер `TextView` определяются значениями `layout_height` и `layout_width`

соответственно. Им можно присвоить одно из трех значений (рис. 10.11), каждое из которых соответствует значению `layout_width` или `layout_height`.

Представлениям **TextView** даты и заголовка назначается большая фиксированная ширина, поэтому они занимают весь экран. Отрегулируйте высоту и ширину этих виджетов. Пока `crime_title` остается выбранным в дереве компонентов, щелкайте на значении ширины, пока в поле не появится значение `wrap_content`. При необходимости изменяйте высоту, пока в поле высоты также не появится значение `wrap_content` (рис. 10.12).

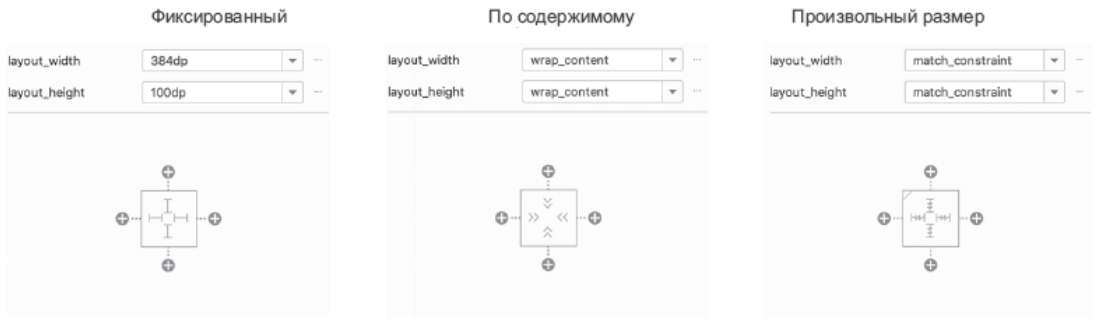


Рис. 10.11. Три варианта определения размера представлений

Таблица 10.1. Типы значений, определяющих размеры представлений

Тип	Значение	Применение
Фиксированный	Ídp	Явно задает размер представления (который не может изменяться) в единицах dp. (Подробнее об этих единицах мы говорили в главе 2.)
По содержимому	wrap_content	Представление получает «желаемый» размер. Для TextView это означает, что представлению будет назначен минимальный размер, необходимый для вывода содержимого
Произвольный размер	match_constraint	Разрешает изменение размеров представления для соблюдения заданных ограничений

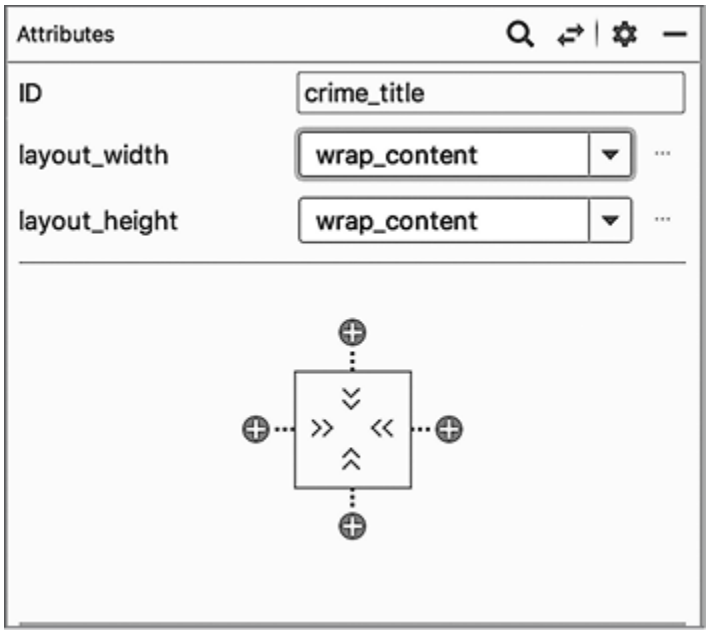


Рис. 10.12. Настройка ширины и высоты

Повторите этот процесс с виджетом `crime_date`, чтобы задать его ширину и высоту.

Теперь оба виджета имеют правильный размер (рис. 10.13), но они все равно перекрываются при запуске приложения, так как для них не задано никаких ограничений. Обратите внимание, что расположение, которое вы видите в предварительном просмотре, отличается от того, что вы видите при запуске приложения. Предварительный просмотр позволяет вам расположить элемент так, чтобы было легче добавлять ваши ограничения, но во время выполнения программы это расположение не сохраняется.

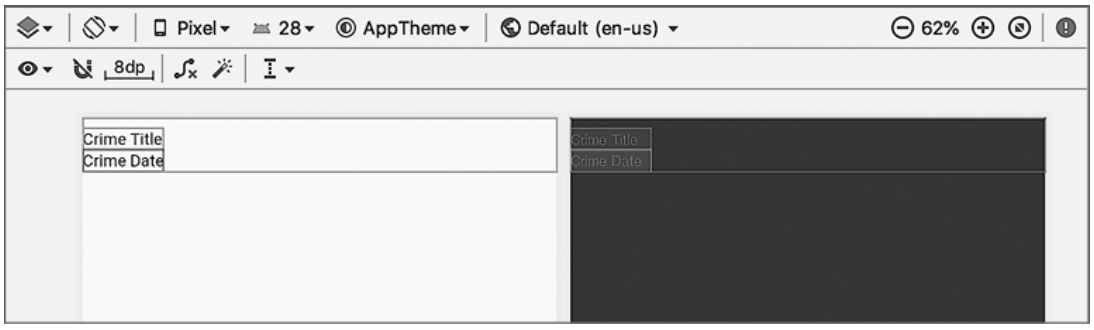


Рис. 10.13. Правильное расположение TextView

Вам будет необходимо добавить ограничения для правильного расположения `TextView`. Но это позже, а пока нужно добавить на макет третий виджет.

Добавление виджетов

В макете освободилось место, и в него можно добавить изображение наручников. Добавьте в файл макета виджет `ImageView`. Найдите его на панели в разделе **Common** (рис. 10.14) и перетащите в дерево компонентов как потомка `ConstraintLayout`, прямо под `crime_date`.

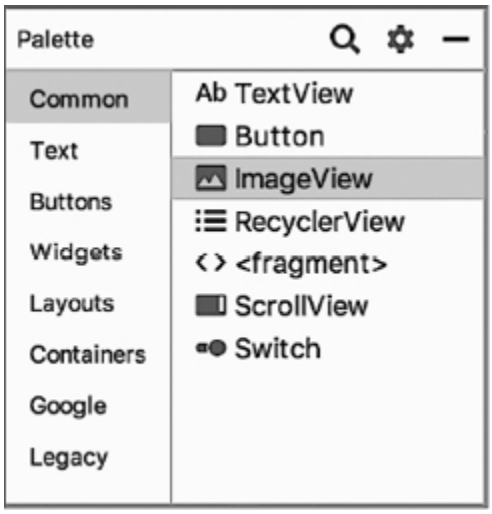


Рис. 10.14. Поиск ImageView

Выберите в разделе **Project** диалогового окна пункт `ic_solved` в качестве ресурса для `ImageView` (рис. 10.15). Изображение будет использоваться для пометки раскрытых преступлений.

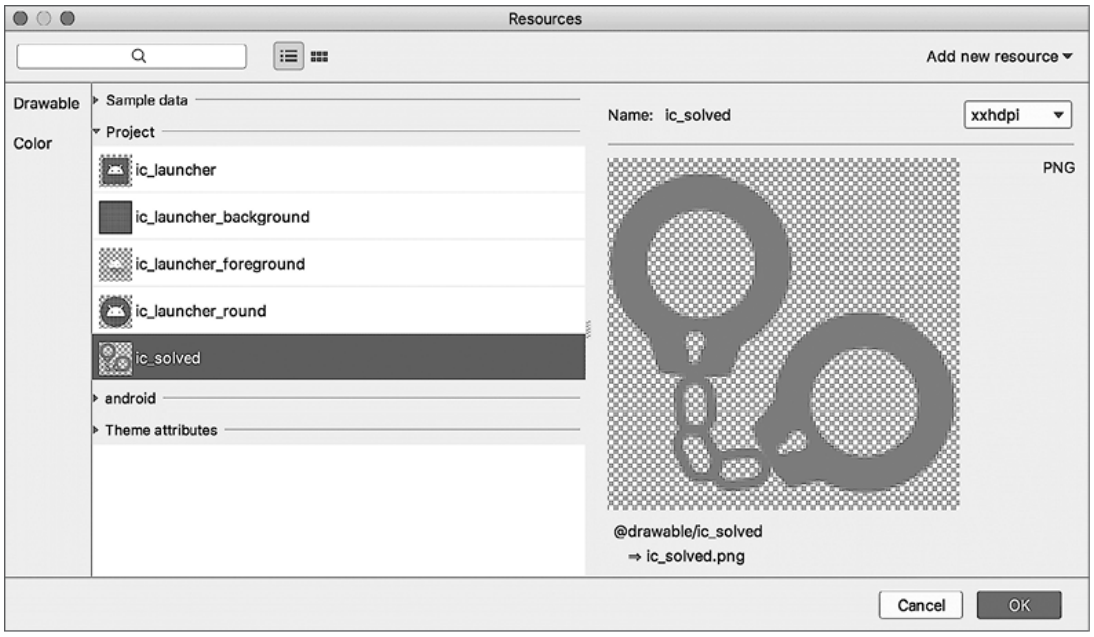


Рис. 10.15. Выбор ресурса ImageView

Виджет `ImageView` теперь является частью макета, но не имеет ограничений. Хоть в графическом редакторе ему назначена позиция, в действительности это ничего не значит.

Пришло время добавить ограничения. Щелкните мышью по `ImageView` в области предварительного просмотра или в дереве компонентов (можно приблизить масштаб, чтобы разглядеть ситуацию получше, инструменты для этого расположены выше инструментов для ограничений). Рядом с `ImageView` появятся *маркеры ограничений* (рис. 10.16).

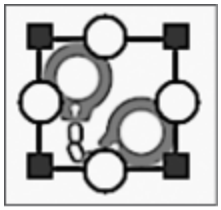


Рис. 10.16. Маркеры ограничений для ImageView

Виджет `ImageView` должен прикрепляться к правому краю представления. Для этого нужно определить ограничения для

верхней, правой и нижней стороны ImageView.

Перед добавлением ограничений перетащите ImageView вправо и вниз, чтобы убрать его от представлений TextView (рис. 10.17). Сейчас неважно, куда мы поместим ImageView. Это место будет проигнорировано, как только вы установите ограничения.

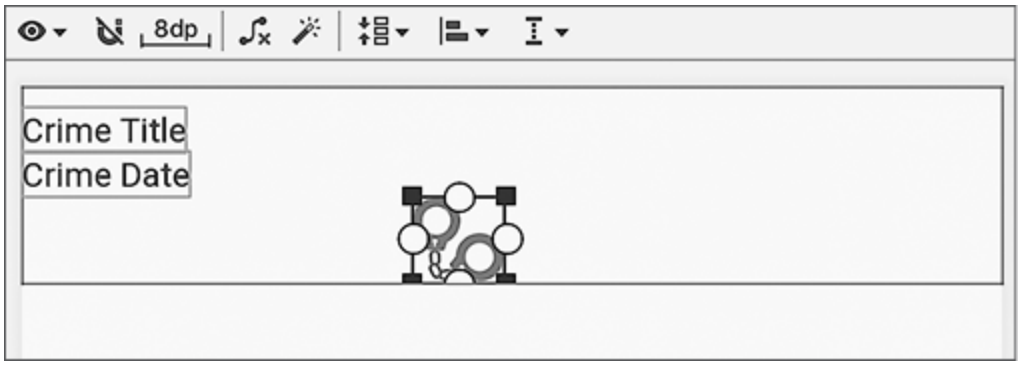


Рис. 10.17. Временное перемещение виджета

Сначала мы создадим ограничение между верхней стороной ImageView и верхней стороной ConstraintLayout. В режиме предварительного просмотра перетащите верхний маркер ограничения с ImageView на верхнюю сторону ConstraintLayout. Маркер ограничения окрасится в зеленый цвет (рис. 10.18).

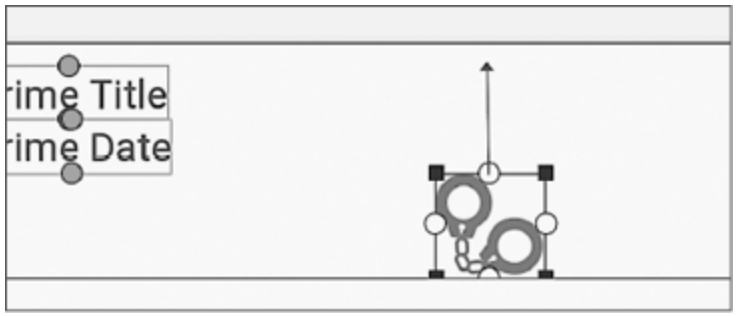


Рис. 10.18. На пути к созданию верхнего ограничения

Тащите дальше. Когда маркер ограничения окрасится в синий цвет, отпустите кнопку мыши, чтобы создать

ограничение (рис. 10.19).

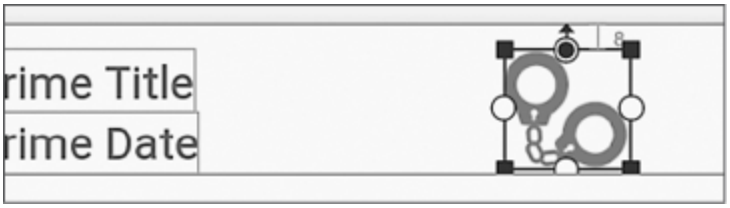


Рис. 10.19. Создание ограничения для верхней стороны ImageView

Будьте внимательны и не щелкайте, когда указатель мыши имеет форму угла — это приведет к изменению размеров `ImageView`. Также следите за тем, чтобы ограничение не соединилось случайно с одним из виджетов `TextView`. Если это произойдет, щелкните мышью по маркеру ограничения, чтобы удалить ошибочное ограничение, и повторите попытку.

Когда вы отпускаете кнопку мыши и задаете ограничение, представление фиксируется в позиции, соответствующей новому ограничению. Так происходит расстановка представлений в `ConstraintLayout` — вы создаете и уничтожаете ограничения.

Чтобы убедиться в том, что у `ImageView` существует нужное ограничение (верхняя сторона `ImageView` связана с верхней стороной `ConstraintLayout`), наведите на `ImageView` указатель мыши. Результат должен выглядеть так, как показано на рис. 10.20.

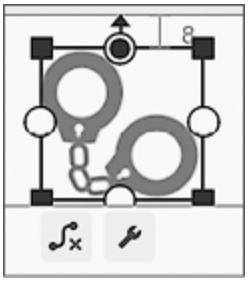


Рис. 10.20. ImageView с ограничением для верхней стороны

Проделайте то же самое с маркером ограничения для нижней стороны: перетащите его с `ImageView` на нижнюю сторону корневого представления (следите за тем, чтобы случайно не присоединить его к `TextView`).

Рис. 10.21. ImageView с ограничением для верхней и нижней стороны

Наконец, перетащите правый маркер ограничения с `ImageView` на правую сторону корневого представления. После этого все необходимые ограничения должны быть успешно созданы. Результат выглядит так, как показано на рис. 10.22.

Рис. 10.22. ImageView с тремя ограничениями

Внутренние механизмы ConstraintLayout

Все изменения, вносимые в графическом редакторе, отражаются в разметке XML незаметно для разработчика. При этом ничто не мешает напрямую редактировать разметку XML, относящуюся к `ConstraintLayout`, но работать с графическим редактором обычно удобнее, потому что разметка `ConstraintLayout` занимает гораздо больше места, чем разметка других типов `ViewGroup`, поэтому добавление начальных ограничений вручную может оказаться сложным. Непосредственная работа с XML может быть более полезной, если необходимо внести небольшие изменения в макет.

(Графические инструменты для работы с макетом полезны особенно в случае с ConstraintLayout. Однако не все их любят. Тут не нужно выбирать сторону — вы можете в любой момент переключаться между графическим редактором и непосредственным редактированием XML. Используйте тот инструмент, который вам больше нравится, или даже оба сразу.)

Чтобы увидеть, что произошло с разметкой XML при создании трех ограничений для ImageView, переключитесь в текстовый режим.

```
<androidx.constraintlayout.widget.ConstraintLay
out
        ... >
    ...
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="8dp"
            app:layout_constraintBottom_toBotto
mOf="parent"
            app:layout_constraintEnd_toEndOf="p
arent"
            app:layout_constraintTop_toTopOf="p
arent"
        app:srcCompat="@drawable/ic_solved" />

</android.support.constraint.ConstraintLayout>
```

(Никуда не делись ошибки, связанные с двумя TextView. Оставьте как есть — мы исправим их позже.)

Все виджеты наследуются напрямую от ConstraintLayout, и вложенных макетов у нас нет. Для создания такого же макета с помощью LinearLayout вам пришлось бы вложить один в другой. Как мы уже говорили ранее, уменьшение вложенности также сокращает время, необходимое для рендеринга макета, что приводит к более быстрому и безупречному использованию.

Присмотритесь к верхнему ограничению:

```
app:layout_constraintTop_toTopOf="parent"
```

Атрибут начинается с префикса layout_. Все атрибуты, начинающиеся с этого префикса, называются *параметрами макета* (*layout parameters*). В отличие от других атрибутов параметры макета представляют собой инструкции для *родителя* виджета, а не для самого виджета. Они сообщают родительскому макету, как он должен расположить дочерний элемент внутри себя. Ранее вам уже встречались примеры параметров макетов layout_width и layout_height.

Имя ограничения constraintTop указывает на то, что это ограничение относится к верхней стороне ImageView.

Наконец, атрибут завершается суффиксом toTopOf="parent". Из него следует, что ограничение связывается с верхней стороной родителя. Родителем в данном случае выступает ConstraintLayout.

А теперь оставим позади низкоуровневую разметку XML и вернемся к графическому редактору.

Редактирование свойств

Элемент `ImageView` теперь расположен правильно. Пора сделать следующий шаг: разместить и задать размеры виджета `TextView` с заголовком.

Сначала выделите объект `CrimeDate` в представлении и перетащите его в сторону (рис. 10.23). Не забудьте, что любые изменения, внесенные в позицию в области предварительного просмотра, не будут представлены во время работы приложения. Во время выполнения останутся только ограничения.

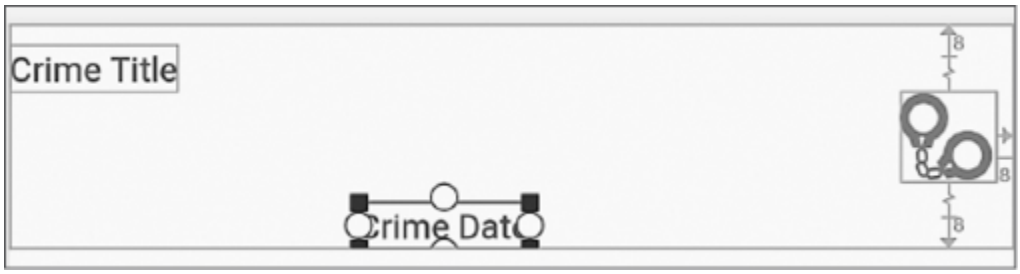


Рис. 10.23. Дата, в сторону!

Теперь выделите `crime_title` в дереве компонентов. Это также приведет к выделению объекта `CrimeTitle` в области предварительного просмотра.

Виджет `CrimeTitle` должен находиться в левой верхней части макета, в левой части нового представления `ImageView`.

Для этого необходимо задать три ограничения:

- связать левую сторону представления с левой стороной родителя;
- связать верхнюю сторону представления с верхней стороной родителя;
- связать правую сторону представления с левой стороной нового представления `ImageView`.

Измените ваш макет и создайте все перечисленные ограничения. Щелкните мышью внутри `TextView` и помните, что вы всегда можете нажать сочетание клавиш ⌘+Z (**Ctrl+Z**), чтобы отменить выполненное действие и попробовать снова.

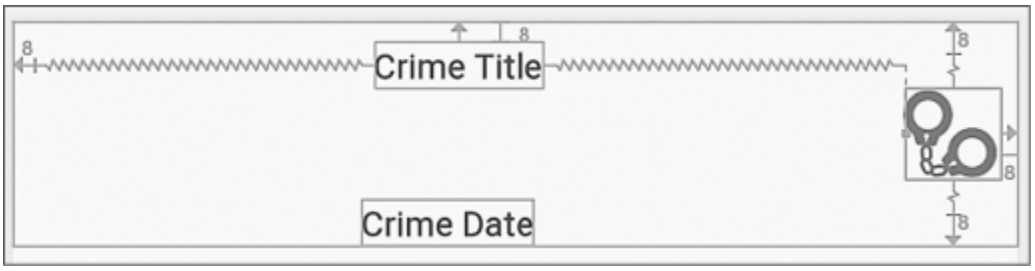


Рис. 10.24. Ограничения виджета TextView

Теперь добавим ограничения для `TextView`. Выбрав `CrimeTitle` на панели предварительного просмотра, проверьте атрибуты на панели справа. Поскольку вы добавили ограничения в верхней, левой и правой части `TextView`, появятся раскрывающиеся списки, позволяющие вам выбрать поле для каждого ограничения (рис. 10.25). Выберите `16dp` для левого и верхнего полей и `8dp` для правого поля.

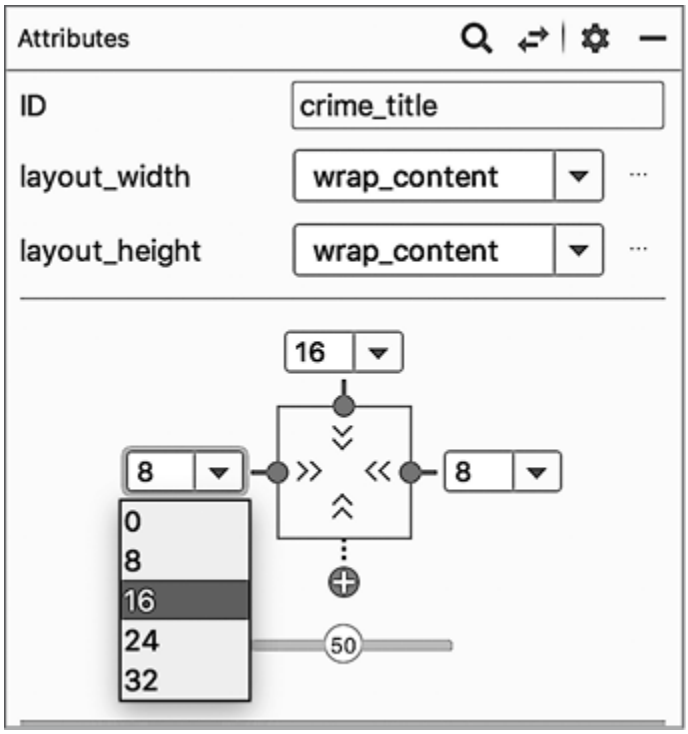


Рис. 10.25. Добавление полей в TextView

Обратите внимание, что по умолчанию Android Studio задает поля 16dp или 8dp. Эти значения соответствуют руководству по проектированию Android. Все руководства по проектированию Android вы можете найти на сайте **developer.android.com/design/index.html**. Ваши приложения для Android должны как можно точнее следовать этим рекомендациям.

Убедитесь, что ваши ограничения выглядят так, как показано на рис. 10.26 (у виджета будут отображаться волнистые линии для любого из ограничений, которые являются растягивающимися).

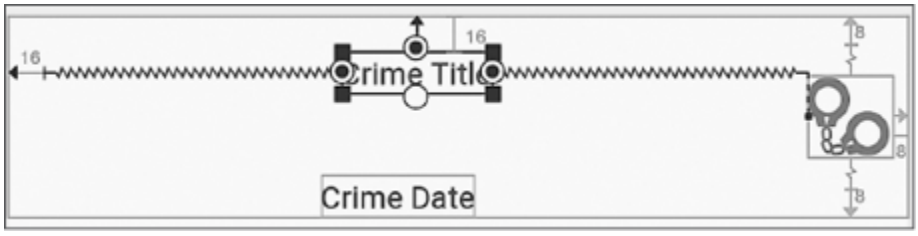


Рис. 10.26. Ограничения TextView с заголовком

После того как ограничения будут созданы, виджет `TextView` с заголовком можно восстановить во всей красе. Задайте его горизонтальное значение, чтобы виджет `TextView` с заголовком заполнил все свободное место с учетом имеющихся ограничений. Выберите для вертикального размера значение `wrap_content`, если это не было сделано ранее; в этом случае для `TextView` будет выбрана минимальная высота, необходимая для вывода списка. Убедитесь в том, что ваши изменения соответствуют приведенным на рис. 10.27.

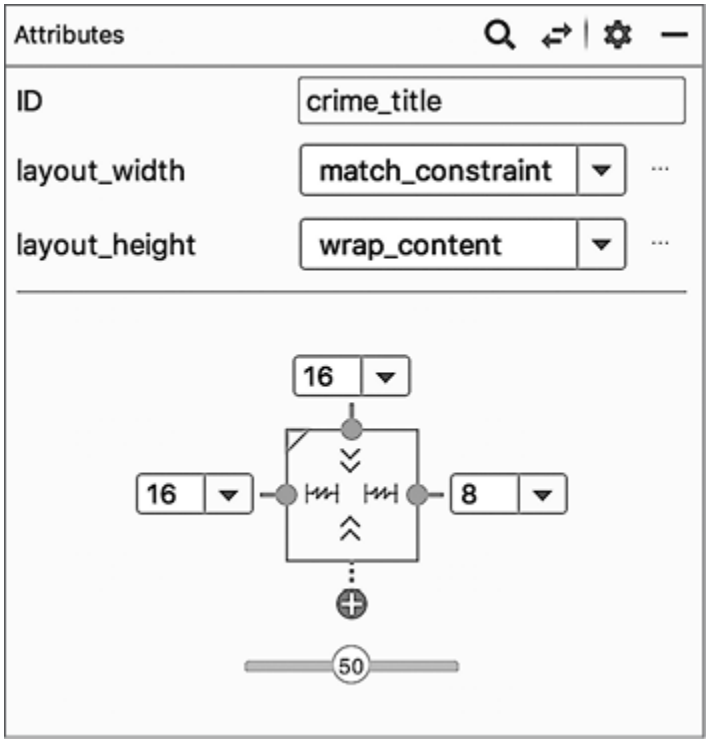


Рис. 10.27. Настройки представления crime_title

Теперь добавьте ограничения для виджета `TextView` с датой. Выберите `crime_date` в дереве компонентов. Нужно задать три ограничения:

- связать левую сторону представления с левой стороной родителя, с полем 16 dp;

- связать верхнюю сторону представления с нижней стороной заголовка, с полем 8 dp;
- связать правую сторону представления с левой стороной нового виджета `ImageView`, с полем 8 dp.

После добавления ограничений настройте свойства `TextView`. В качестве значения ширины виджета `TextView` с датой выбирается `match_constraint`, а в поле высоты — `wrap_content`, как и у виджета `TextView` с заголовком. Убедитесь в том, что ваши настройки соответствуют приведенным на рис. 10.28.

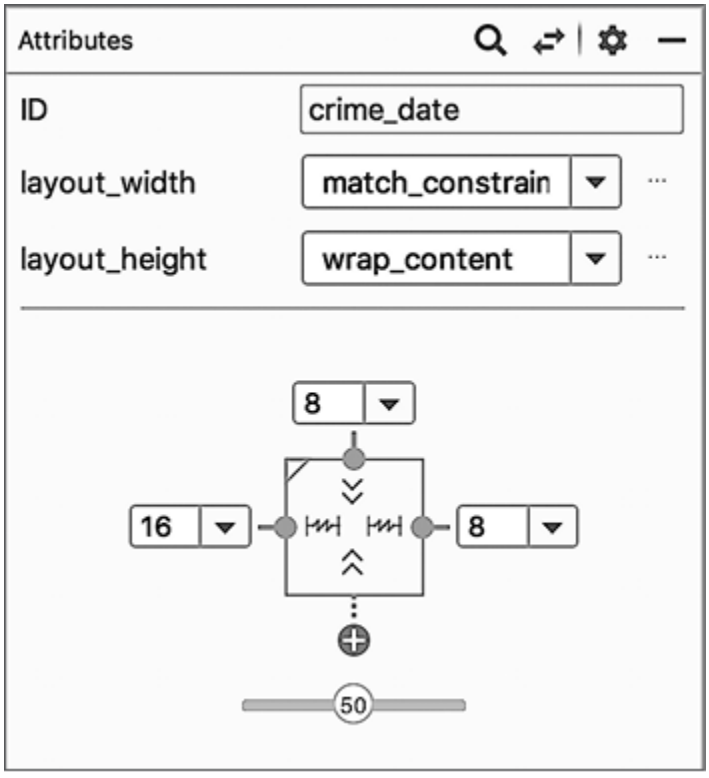


Рис. 10.28. Настройки представления crime_date

В результате макет в области предварительного просмотра должен выглядеть так, как показано на рис. 10.1 в начале главы.

При приближении вы должны видеть изображение, как показано на рис. 10.29.

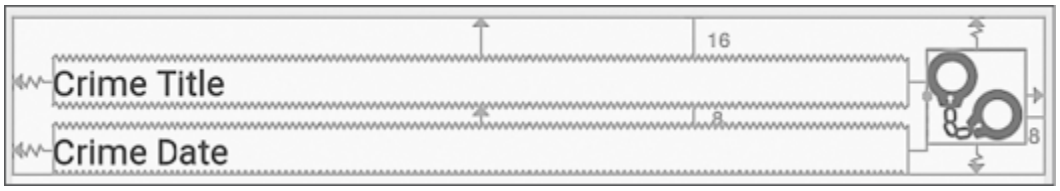


Рис. 10.29. Готовые ограничения крупным планом

Перейдите в текстовый режим в окне редактора, чтобы просмотреть файл XML, полученный в результате изменений в редакторе графического макета. Красные подчеркивания под `TextView` больше не видны. Это связано с тем, что виджеты `TextView` теперь правильно ограничены, поэтому `ConstraintLayout` правильно определяет, где расположить виджеты во время исполнения.

Два желтых предупреждающих индикатора остаются связанными с представлениями `TextView`, и если вы исследуете их, то увидите, что предупреждения связаны с их жестко закодированными строками. Эти предупреждения были бы важны для реального приложения, но в случае с CriminalIntent вы можете их проигнорировать. (Не стесняйтесь следовать советам по извлечению жестко закодированного текста в строковые ресурсы. Это решит проблему с предупреждениями.)

Кроме того, на `ImageView` осталось одно предупреждение, указывающее на то, что вы не установили описание контента. Пока что можно не обращать внимания на это предупреждение. Позже в главе 18 вы узнаете об описаниях контента. Ну а пока ваше приложение будет работать нормально, с тем исключением, что изображение не будет доступно пользователям, использующим программу чтения с экрана.

Запустите приложение CriminalIntent и убедитесь в том, что все три компонента аккуратно выстраиваются в каждой строке `RecyclerView` (рис. 10.30).

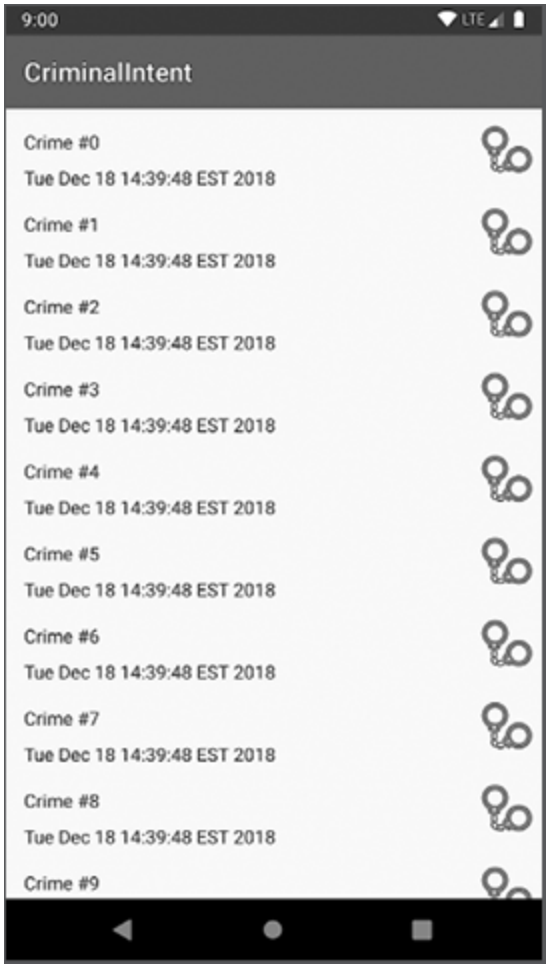


Рис. 10.30. Каждая строка содержит три представления

Динамическое поведение элементов списка

После того как для макета будут определены необходимые ограничения, следует обновить виджет `ImageView`, чтобы изображение наручников выводилось только для раскрытых преступлений.

Сначала обновите идентификатор `ImageView`. Когда вы добавили виджет `ImageView` в `ConstraintLayout`, ему было

присвоено имя по умолчанию — не слишком содержательное. Выберите виджет `ImageView` в файле `list_item_crime.xml`, затем в окне свойств замените текущее значение идентификатора на `crime_solved` (рис. 10.31). Android Studio спросит, нужно ли обновить все вхождения измененного идентификатора; выберите ответ **Yes**.

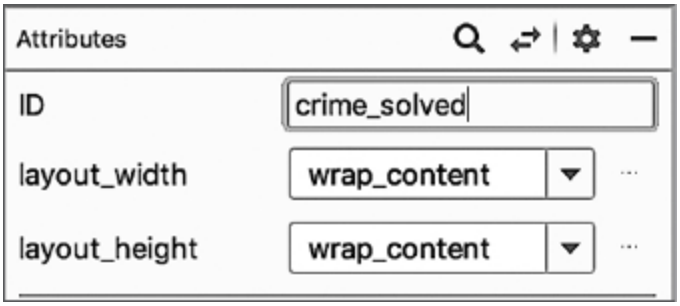


Рис. 10.31. Изменение идентификатора виджета

Вы могли заметить, что используете одни и те же идентификаторы представлений в разных макетах. Идентификатор `crime_solved` используется как в макетах `list_item_crime.xml`, так и во `fragment_crime.xml`. Вы можете подумать, что повторное использование идентификаторов — это неправильно, но в данном случае это не так. Идентификаторы макетов должны быть уникальными только в пределах макета. Так как ваши идентификаторы определены в разных файлах, проблем не возникнет.

Когда с идентификаторами разобрались, нужно обновить код. Откройте файл `CrimeListFragment.kt`. В `CrimeHolder` добавьте переменную экземпляра `ImageView` и переключите ее видимость в зависимости от состояния преступления.

Листинг 10.1. Изменение видимости изображения (CrimeListFragment.kt)

```
private inner class CrimeHolder(view: View)
```

```
            :   RecyclerView.ViewHolder(view),
View.OnClickListener {
    ...
    private val dateTextView: TextView
      private val solvedImageView: ImageView =
itemView.findViewById(R.id.crime_solved)

    init {
        ...
    }

    fun bind(crime: Crime) {
        this.crime = crime
        titleTextView.text = this.crime.title
                            dateTextView.text   =
this.crime.date.toString()
                solvedImageView.visibility  =  if
(crime.isSolved) {
            View.VISIBLE
        } else {
            View.GONE
        }
    }
    ...
}
```

Запустите приложение CriminalIntent и убедитесь в том, что в строках через одну теперь выводится изображение наручников.

Подробнее об атрибутах макетов

Давайте добавим еще несколько настроек в `list_item_crime.xml`, а заодно ответим на некоторые запоздалые вопросы о виджетах и атрибутах.

Вернитесь к режиму **Design** файла `list_item_crime.xml`. Выберите виджет `crime_title` и измените некоторые атрибуты на панели **Attributes**.

Щелкните мышью по стрелке рядом с `textAppearance`, чтобы открыть атрибуты текста и шрифта. Измените значение атрибута `textColor` на `@android:color/black`, а атрибута `textSize` на `18sp` (рис. 10.32).

Вы можете вводить эти значения непосредственно в поля для атрибутов из раскрывающегося списка (как, например, `textSize`) либо нажать кнопку **...** рядом с полем, чтобы назначить значение.

Запустите приложение CriminalIntent; вы удивитесь, насколько лучше все смотрится после небольшой доработки.

Стили, темы и атрибуты тем

Стиль (*style*) представляет собой ресурс XML, который содержит атрибуты, описывающие внешний вид и поведение виджета. Например, ниже приведен ресурс стиля, который настраивает виджет на использование увеличенного размера текста.

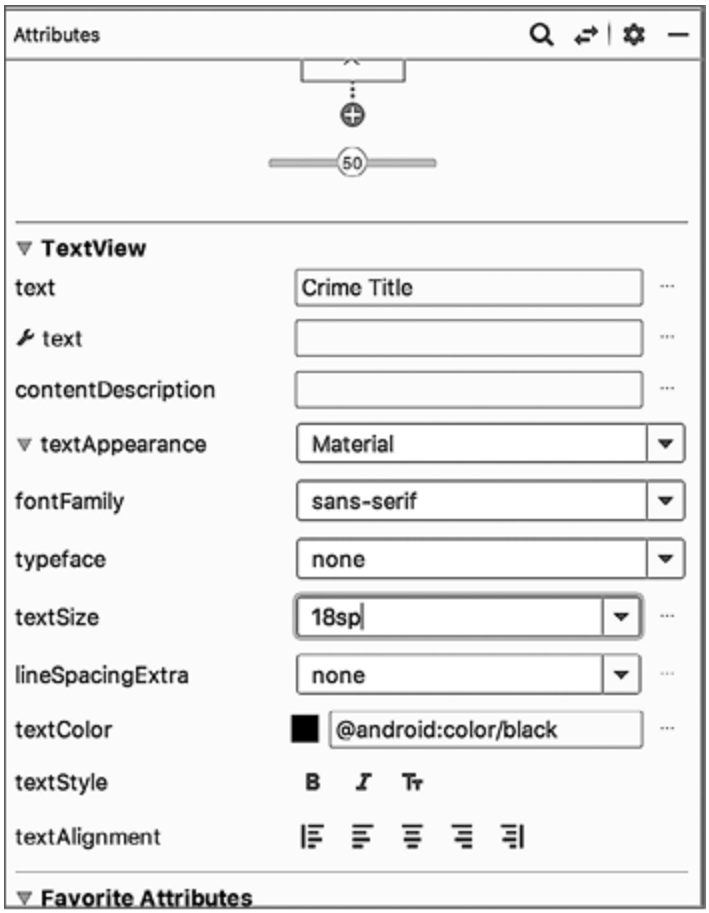


Рис. 10.32. Изменение цвета и размера текста

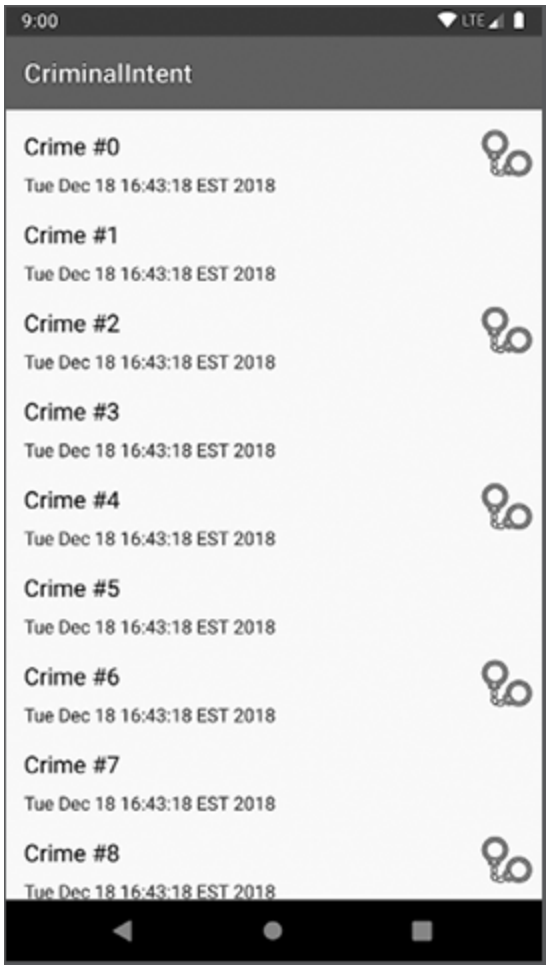


Рис. 10.33. После отделки

```
<style name="BigTextStyle">
  <item name="android:textSize">20sp</item>
  <item name="android:padding">3dp</item>
</style>
```

Вы можете создавать собственные стили (этим мы займемся в главе 21). Они добавляются в файл стилей из каталога `res/values/`, а ссылки на них в макетах выглядят так: `@style/my_own_style`.

Еще раз взгляните на виджеты `TextView` из файла `layout/fragment_crime.xml` (но не `list_item_crime.xml`, с которым мы работали в этой главе); каждый виджет имеет

атрибут `style`, который ссылается на стиль, созданный Android. С этим конкретным стилем виджеты `TextView` выглядят как разделители списка, а берется он из темы приложения.

Тема (*theme*) представляет собой набор стилей. Со структурной точки зрения тема сама по себе — это ресурс стиля, атрибуты которого ссылаются на другие ресурсы стилей.

Android предоставляет платформенные темы, которые могут использоваться вашими приложениями. При создании CriminalIntent мастер назначает тему приложения, которая определяется тегом `application` в манифесте.

Стиль из темы приложения можно применить к виджету при помощи *ссылки на атрибут темы* (*theme attribute reference*). Именно это мы делаем в файле `fragment_crime.xml`, используя значение `?android:listSeparatorTextViewStyle`.

Ссылка на атрибут темы приказывает менеджеру ресурсов Android: «Перейди к теме приложения и найди в ней атрибут `listSeparatorTextViewStyle`. Этот атрибут указывает на другой ресурс стиля. Помести значение этого ресурса сюда».

Каждая тема Android включает атрибут `listSeparatorTextViewStyle`, но его определение зависит от оформления конкретной темы. Использование ссылки на атрибут темы гарантирует, что оформление виджетов `TextView` будет соответствовать оформлению вашего приложения.

О том, как работают стили и темы, более подробно рассказано в главе 21.

Для любознательных: поля и отступы

В обоих наших приложениях, GeoQuiz и CriminalIntent, мы задавали виджетам атрибуты полей и отступов. Начинающие разработчики иногда путают их. Теперь, когда вы понимаете, что такое параметры макета, разницу легче объяснить.

Поле — это параметр макета. Оно определяет расстояние между виджетами. Учитывая, что виджет «знает» только свои параметры, за поля должны отвечать родители виджета.

Отступ не относится к параметрам макета. Атрибут `android:padding` сообщает виджету, сколько места он должен занимать. Например, вы хотите, чтобы кнопка даты была впечатляюще большой без изменения размера текста (рис. 10.34).

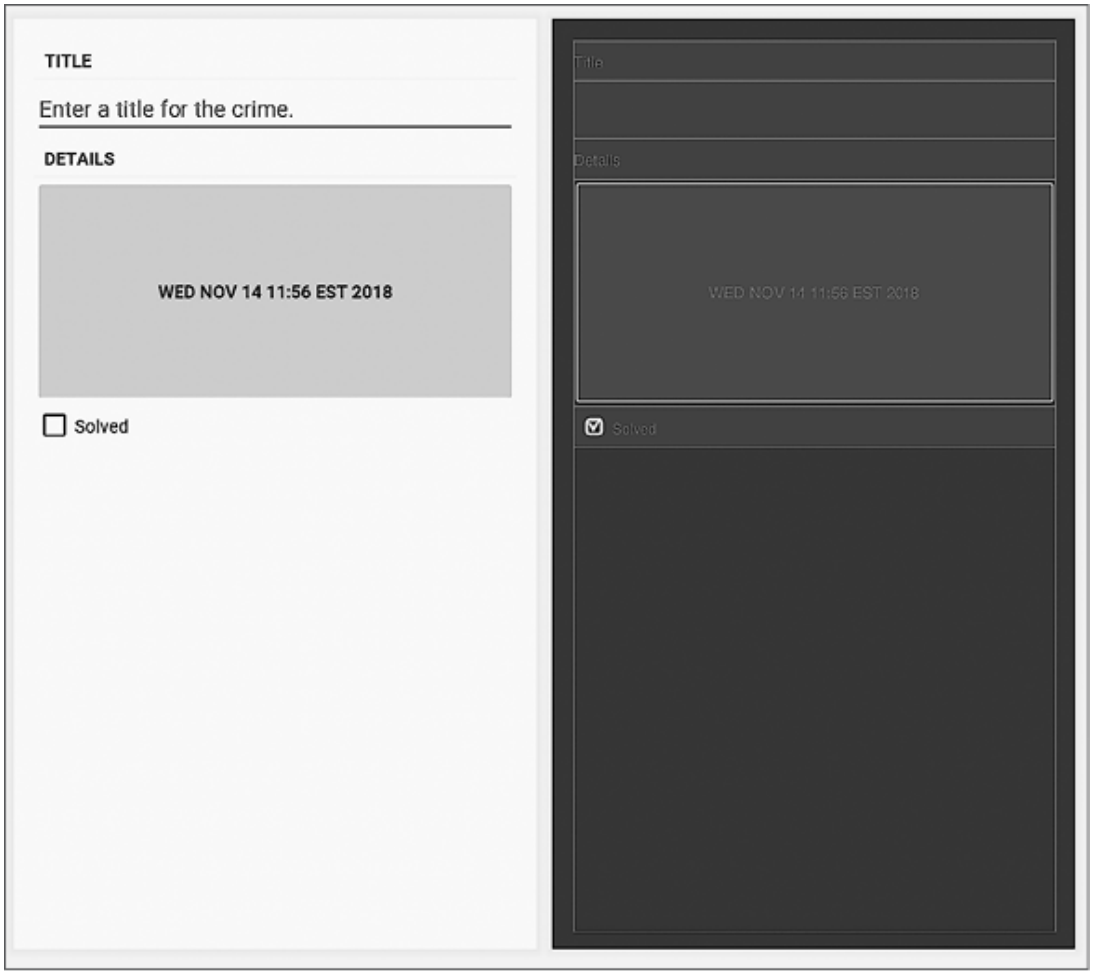


Рис. 10.34. Ну нравятся мне большие кнопки, что ж тут поделать

Вы можете добавить кнопке следующий атрибут:

```
<Button
    android:id="@+id/crime_date"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
       android:padding="80dp"
    tools:text="Wed Nov 14 11:56 EST 2018"/>
```

Вероятно, вам стоит удалить этот атрибут, прежде чем мы продолжим.

Для любознательных: нововведения в ConstraintLayout

У `ConstraintLayout` есть дополнительные возможности, упрощающие расположение дочерних элементов. В этой главе вы располагали элементы, ограничивая их со стороны родителя и других дочерних элементов.

В `ConstraintLayout` также есть вспомогательные виджеты, такие как `Guideline` (направляющие), которые упрощают расположение элементов на экране.

Направляющие не отображаются на экране приложения, но сами помогают расположить элементы так, как вам нужно. Существуют как горизонтальные, так и вертикальные направляющие, и их можно разместить в определенном месте на экране, используя значения dp или проценты от экрана. Потом эти направляющие можно использовать в качестве ограничений для элементов, даже если размер экрана будет изменяться.

На рис. 10.35 показан пример использования вертикальной направляющей `Guideline`. Она расположена на 20 процентах

от ширины родительского экрана. И заголовок преступления, и дата ограничены по направляющей, а не по родителю.

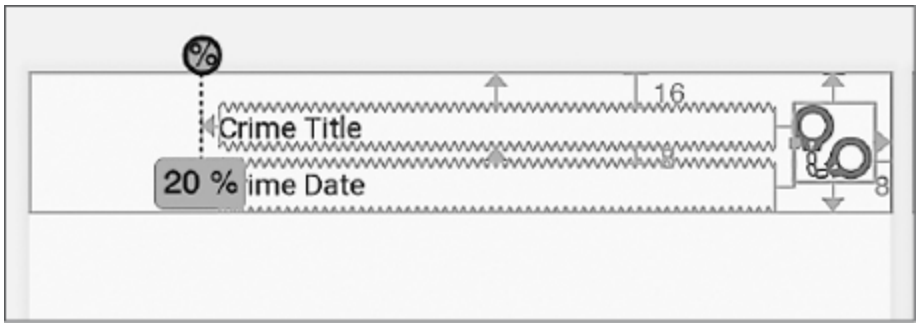


Рис. 10.35. Использование направляющих

`MotionLayout` — это расширение `ConstraintLayout`, которое упрощает добавление анимации для элементов. Для использования `MotionLayout` вы создаете файл MotionScene, в котором описывается, как должна выполняться анимация и какие элементы сопоставляются друг с другом в начальном и конечном макетах. Вы также можете установить ключевые кадры для промежуточных точек анимации. `MotionLayout` выполняет анимацию из начального состояния через различные предоставляемые вами ключевые кадры, и так до финального положения.

Упражнение. Форматирование даты

Объект `Date` больше напоминает временную метку (timestamp), чем традиционную дату. При вызове `toString()` для `Date` вы получаете именно временную метку, которая используется в строке `RecyclerView`. Временные метки хорошо подходят для отчетов, но на кнопке было бы лучше выводить дату в формате, более привычном для людей (например, «Jul 22, 2019»). Для этого можно воспользоваться экземпляром класса `android.text.format.DateFormat`.

Хорошей отправной точкой в работе станет описание этого класса в документации Android.

Используйте функции класса `DateFormat` для формирования строки в стандартном формате или же подготовьте собственную форматную строку. Чтобы задача стала более творческой, создайте форматную строку для вывода дня недели («Monday, Jul 22, 2019»).

11. Базы данных и Room Library

Почти каждому приложению необходимо какое-то место для долгосрочного хранения данных. В этой главе мы создадим базу данных для приложения CriminalIntent и заполним ее фиктивными данными. Затем мы научим приложение извлекать данные преступления из базы данных и отображать его в списке преступлений (рис. 11.1).

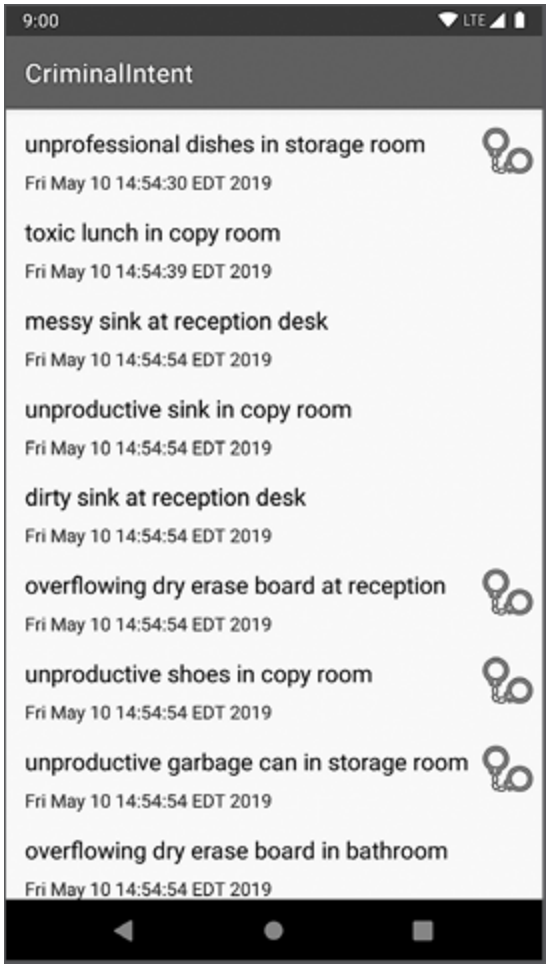


Рис. 11.1. Отображение преступления из базы данных

В главе 4 мы узнали, как сохранять переходные состояния интерфейса после поворота экрана и смерти процесса с

помощью `ViewModel` и сохраненного состояния экземпляра. Эти два подхода отлично подходят, если к пользовательскому интерфейсу привязаны небольшие объемы данных. Однако эти подходы не должны использоваться для хранения данных, не связанных с пользовательским интерфейсом, или данных, не связанных с activity или фрагментом, которые нужно сохранять независимо от состояния пользовательского интерфейса.

Вместо этого нужно хранить подобные данные приложения в локальном хранилище (в файловой системе или в базе данных, как вы будете делать для CriminalIntent) или на веб-сервере.

Библиотека компонентов архитектуры Room

Room — это библиотека компонентов архитектуры Jetpack, позволяющая упростить настройку и доступ к базе данных. Она помогает определить структуру базы данных и запросов с помощью аннотированных классов Kotlin.

Room состоит из API, аннотаций и компилятора. API содержит классы, которые вы будете расширять для определения базы данных и создания ее экземпляра. Вы можете использовать аннотации, чтобы указать, например, что должно быть сохранено в базе данных, какой класс представляет вашу базу данных и какой класс определяет функцию доступа к таблицам базы данных. Компилятор обрабатывает аннотированные классы и генерирует реализацию базы данных.

Чтобы использовать Room, сначала нужно добавить необходимые зависимости. Добавьте зависимости `room-runtime` и `room-compiler` в файл `app/build.gradle`.

Листинг 11.1. Добавление зависимостей (app/build.gradle)

```
apply plugin: 'kotlin-android-extensions'

apply plugin: 'kotlin-kapt'

android {
    ...
}
...
dependencies {
    ...
        implementation 'androidx.core:core-
ktx:1.1.0-alpha04'
        implementation 'androidx.room:room-
runtime:2.1.0-alpha04'
      kapt 'androidx.room:room-compiler:2.1.0-
alpha04'
    ...
}
```

В верхней части файла мы добавили новый *плагин* Android Studio. Плагины позволяют добавить IDE новые функции. Перейдите по ссылке **plugins.jetbrains.com/androidstudio**, где вы найдете много новых плагинов, которые можно добавить.

`kotlin-kapt` означает «инструмент обработки аннотаций Kotlin». Когда библиотека генерирует код, вам наверняка захочется использовать эти сгенерированные классы непосредственно в коде. Но по умолчанию эти сгенерированные классы не видны в Android Studio, так что при попытке импорта вы увидите сообщение об ошибке. Добавление плагина `kotlin-kapt` реализует поддержку

сгенерированных файлов в Android Studio, поэтому вы можете импортировать их в свои классы.

Первая добавленная вами зависимость `room-runtime` предназначена для API Room, содержащего все классы и аннотации, которые вам потребуются для определения базы данных. Вторая зависимость, `room-compiler`, нужна для компилятора Room, который будет генерировать вашу реализацию базы данных на основе заданных аннотаций. Компилятор использует ключевое слово `kapt` вместо `implementation`, поэтому сгенерированные классы от компилятора открыты для Android Studio благодаря плагину `kotlin-kapt`.

Не забудьте синхронизировать ваши файлы Gradle. Теперь мы можем перейти к подготовке слоя модели для хранения в базе данных.

Создание базы данных

Создание базы данных с помощью Room делается в три этапа:

- аннотирование класса модели, чтобы сделать его сущностью базы данных;
- создание класса, который будет представлять саму базу данных;
- создание конвертера типа, позволяющего базе данных обрабатывать ваши данные модели.

Room позволяет выполнить все эти три шага легким движением руки.

Определение сущностей

Room создает структуру таблицы базы данных для приложения, основываясь на определенных вами *сущностях*. Сущности — это классы моделей, аннотированные аннотацией `@Entity`. Room создаст таблицу базы данных для любого класса с такой аннотацией.

Поскольку вы хотите хранить в базе данных преступления, объект `Crime` должен быть сущностью Room. Откройте файл `Crime.kt` и добавьте две аннотации.

Листинг 11.2. Превращаем Crime в сущность (Crime.kt)

```
@Entity
data class Crime(@PrimaryKey val id: UUID =
UUID.randomUUID(),
                 var title: String = "",
                 var date: Date = Date(),
                 var isSolved: Boolean = false)
```

Первая аннотация, `@Entity`, применяется на уровне класса. Эта аннотация указывает, что класс определяет структуру таблицы или набора таблиц в базе данных. В этом случае каждая строка в таблице будет представлять собой отдельные преступления. Каждое свойство, определенное в классе, превратится в столбец в таблице, при этом имя свойства станет именем столбца. В нашем случае у таблицы будет четыре столбца: `id`, `title`, `date` и `isSolved`.

Вторая добавленная нами аннотация — это `@PrimaryKey`, которую вы добавили к свойству `id`. Эта аннотация указывает, какой столбец в базе данных является *первичнымключом*. Первичный ключ в базе данных — это такой столбец, который содержит данные, уникальные для каждой записи или строки.

Такой столбец можно использовать для вызова отдельных записей. Свойство `id` уникально для каждого преступления, поэтому добавление `@PrimaryKey` для этого свойства позволяет запрашивать преступления по идентификатору.

Теперь у класса `Crime` есть нужные аннотации, и вы можете переходить к созданию вашего класса базы данных.

Создание класса базы данных

Классы-сущности определяют структуру таблиц базы данных. Один класс-сущность может использоваться в нескольких базах данных, если у вашего приложения их несколько. Это редкая, но возможная ситуация. По этой причине такие классы не используются Room для создания таблицы, если вы явно не свяжете их его с базой данных, что мы и сделаем в ближайшее время.

Во-первых, нужно создать новый пакет под названием `database` для связанного с базой данных кода. На панели **Project** щелкните правой кнопкой мыши по папке `com.bignerdranch.android.criminalintent` и выберите команду **New⇒Package** в контекстном меню. Назовите свой новый пакет `database`.

Теперь создайте новый класс `CrimeDatabase` в пакете `database` и определите класс, как показано ниже.

Листинг 11.3. Инициализация класса CrimeDatabase (database/CrimeDatabase.kt)

```
@Database(entities = [ Crime::class ], version=1)
abstract class CrimeDatabase : RoomDatabase() {
}
```

Аннотация `@Database` сообщает Room о том, что этот класс представляет собой базу данных в приложении. Самой аннотации требуется два параметра. Первый параметр — это список классов-сущностей, который сообщает Room, какие использовать классы при создании и управлении таблицами для этой базы данных. В нашем случае мы передаем только класс `Crime`, так как это единственная сущность в приложении.

Второй параметр — версия базы данных. При первом создании базы данных версия должна быть равна 1. При будущей разработке приложения вы можете добавлять новые сущности и новые свойства существующим сущностям. В этом случае вам нужно будет изменить список сущностей и увеличить версию базы данных, чтобы обозначить факт изменения (мы сделаем это в главе 15).

Сам класс базы данных пока пустой. `CrimeDatabase` расширяется от `RoomDatabase` и помечается как абстрактный, так что вы не можете создать экземпляр непосредственно от него. Вы узнаете, как использовать Room, чтобы получить пригодный для использования экземпляр базы данных, позже в этой главе.

Создание преобразователя типа

Под капотом у Room используется SQLite. SQLite — это реляционная база данных с открытым исходным кодом, как MySQL или PostgreSQL (SQL — сокращение Structured Query Language, стандартный язык, используемый для взаимодействия с базами данных). В отличие от других баз данных, SQLite хранит свои данные в простых файлах, которые вы можете считывать и записывать с помощью библиотеки SQLite. В Android в стандартной библиотеке есть библиотека

SQLite наряду с некоторыми дополнительными вспомогательными классами.

Room делает использование SQLite еще проще и чище, выступая в роли объектно-реляционного отображения (ORM) или слоя между вашими объектами Kotlin и реализацией базы данных. В целом вам не нужно знать или заботиться о SQLite при использовании Room, но если вы хотите узнать больше, вы можете посетить сайт **www.sqlite.org**, на котором приведена полная документация по SQLite.

Room позволяет хранить примитивные типы в таблицах базы данных SQLite, но с другими типами будут проблемы. Ваш класс `Crime` опирается на объекты `Date` и `UUID`, которые по умолчанию не знает, как хранить. Вы должны помочь базе данных научиться правильно хранить эти типы и извлекать их из таблицы базы данных.

Чтобы научить Room преобразовывать типы данных, необходимо указать *преобразователь типов*. Преобразователь типа сообщает Room, как преобразовать специальный тип в формат для хранения в базе данных. Вам понадобятся две функции, к которым мы добавим аннотации `@TypeConverter` для каждого типа: одна сообщает Room, как преобразовывать тип, чтобы сохранить его в базе данных, а другая — как выполнить обратное преобразование.

Создайте класс `CrimeTypeConverters` в пакете `database` и добавьте две функции для каждого типа `Date` и `UUID`.

Листинг 11.4. Добавление функций TypeConverter (database/CrimeTypeConverters.kt)

```
class CrimeTypeConverters {

    @TypeConverter
```

```
    fun fromDate(date: Date?): Long? {
        return date?.time
    }

    @TypeConverter
     fun toDate(millisSinceEpoch: Long?): Date?
{
        return millisSinceEpoch?.let {
            Date(it)
        }
    }

    @TypeConverter
    fun toUUID(uuid: String?): UUID? {
        return UUID.fromString(uuid)
    }

    @TypeConverter
    fun fromUUID(uuid: UUID?): String? {
        return uuid?.toString()
    }
}
```

Первые две функции обрабатывают объект `Date`, а другие две — `UUID`. Убедитесь, что вы импортировали версию `java.util.Date` класса `Date`.

Объявление функции преобразователя не позволяет вашей базе данных использовать их. Вы должны явно добавить конвертеры к классу базы данных.

Листинг 11.5. Включение TypeConverter (database/CrimeDatabase.kt)

```
@Database(entities = [Crime::class], version=1)
@TypeConverters(CrimeTypeConverters::class)
abstract class CrimeDatabase : RoomDatabase() {
}
```

Добавив аннотацию `@TypeConverters` и передав класс `CrimeTypeConverters`, вы инструктируете базу данных использовать функции в этом классе при преобразовании типов.

Теперь определения баз данных и таблиц готовы.

Определение объекта доступа к данным

Таблица базы данных мало на что годится, если вы не можете редактировать и получать ее содержимое. Первый шаг к взаимодействию с таблицами базы данных — это создать *объект доступа к данным*, или DAO. DAO — это интерфейс, который содержит функции для каждой операции с базой данных, которые вы хотите реализовать. В этой главе DAO приложения CriminalIntent нужно две функции запроса: одна должна возвращать список всех преступлений в базе данных, а другая возвращать одно преступление, соответствующее данному `UUID`.

Добавьте в пакет `database` файл `CrimeDao.kt`. В нем определите пустой интерфейс `CrimeDao` и аннотацией `@Dao`.

Листинг 11.6. Создание пустого DAO (database/CrimeDao.kt)

```
@Dao
```

```
interface CrimeDao {
}
```

Аннотация `@Dao` сообщает Room, что `CrimeDao` — это один из ваших объектов доступа к данным. Когда вы прикрепляете `CrimeDao` к вашему классу базы данных, Room будет генерировать реализации функций, которые вы добавляете к этому интерфейсу.

Кстати о добавлении функций: время пришло. Добавьте две функции запроса в `CrimeDao`.

Листинг 11.7. Добавление функций запросов к базе данных (database/CrimeDao.kt)

```
@Dao
interface CrimeDao {
    @Query("SELECT * FROM crime")
    fun getCrimes(): List<Crime>
    @Query("SELECT * FROM crime WHERE id=(:id)")
    fun getCrime(id: UUID): Crime?
}
```

Аннотация `@Query` указывает, что `getCrimes()` и `getCrime(UUID)` предназначены для извлечения информации из базы данных, а не вставки, обновления или удаления элементов из базы данных. Возвращаемый тип каждой функции запроса в интерфейсе DAO отражает тип результата запроса.

Аннотация `@Query` в качестве входных данных ожидает строку, содержащую команду SQL. В большинстве случаев для работы с Room достаточно минимальных знаний SQL, но если

вы заинтересованы в получении дополнительной информации, обратитесь к разделу синтаксиса на сайте **www.sqlite.org**.

`SELECT*FROMcrime` выводит все столбцы для всех строк таблицы `crime`. Команда `SELECTFROMcrimeWHEREid=(:id)` запрашивает все столбцы только из строки с нужным идентификатором.

С `CrimeDao` мы пока закончили. В главе 12 мы добавим функцию для обновления существующего преступления. В главе 14 мы добавим функцию, позволяющую добавить в базу новое преступление.

Теперь нам нужно связать класс DAO с классом базы данных. Поскольку `CrimeDao` представляет собой интерфейс, Room будет сам генерировать конкретную версию класса. Но для этого вы должны приказать классу базы данных создать экземпляр DAO.

Чтобы подключить ваш DAO, откройте файл `CrimeDatabase.kt` и добавьте абстрактную функцию, тип возвращаемого значения у которой будет `CrimeDao`.

Листинг 11.8. Регистрация DAO в базе данных (database/CrimeDatabase.kt)

```
@Database(entities = [ Crime::class ], version=1)
@TypeConverters(CrimeTypeConverters::class)
abstract class CrimeDatabase : RoomDatabase() {

    abstract fun crimeDao(): CrimeDao
}
```

Теперь при создании базы данных Room будет генерировать конкретную реализацию в DAO, и вы можете получить к ней

доступ. Если у вас есть ссылки на DAO, вы можете вызвать любую из функций, чтобы взаимодействовать с базой данных.

Доступ к базе данных с помощью шаблона репозитория

Для доступа к базе данных мы будем использовать шаблон репозитория, рекомендованный Google в руководстве по архитектуре приложений (**developer.android.com/jetpack/docs/guide**).

Класс репозитория инкапсулирует логику для доступа к данным из одного источника или совокупности источников. Он определяет, как захватывать и хранить определенный набор данных — локально, в базе данных или с удаленного сервера. Ваш код UI будет запрашивать все данные из репозитория, потому что интерфейсу неважно, как фактически хранятся или извлекаются данные. Это детали реализации самого репозитория.

Поскольку CriminalIntent — довольно простое приложение, репозиторий будет обрабатывать только выборку данных из базы.

Создайте класс `CrimeRepository` в пакете `com.bignerdranch.android.criminalintent` и определите объект в классе.

Листинг 11.9. Реализация репозитория (CrimeRepository.kt)

```
class CrimeRepository private
constructor(context: Context) {

    companion object {
        private var INSTANCE: CrimeRepository? =
null
```

```
        fun initialize(context: Context) {
            if (INSTANCE == null) {
                                    INSTANCE  =
CrimeRepository(context)
            }
        }

        fun get(): CrimeRepository {
            return INSTANCE ?:
                                                throw
IllegalStateException("CrimeRepository must be
initialized")
        }
    }
}
```

`CrimeRepository` — это *одноэлементный* класс (*синглтон*). Это означает, что в вашем процессе приложения единовременно существует только один его экземпляр.

Синглтон существует до тех пор, пока приложение находится в памяти, поэтому хранение в нем любых свойств позволяет получить к ним доступ в течение жизненного цикла вашей activity и фрагмента. Будьте осторожны с синглтонами, так как они уничтожаются, когда Android удаляет приложение из памяти.

Синглтон `CrimeRepository` не подходит для долговременного хранения данных. Вместо этого он выдает данные о преступлении и дает возможность легко передавать эти данные между классами контроллера.

Чтобы класс `CrimeRepository` стал синглтоном, нужно добавить две функции в сопутствующий объект. Одна из них

будет инициализировать новый экземпляр в репозиторий, а другая обеспечивать к нему доступ. Вы также можете пометить конструктор как приватный, чтобы убедиться в отсутствии компонентов, которые могут пойти против системы и создать собственный экземпляр.

Функция запроса будет работать не очень хорошо, если не вызвать перед ней функцию `Initialize()`. Она выбросит исключение `IllegalStateException`, так что вам нужно обязательно инициализировать репозиторий при запуске приложения.

Чтобы выполнить работу, как только приложение будет готово, вы можете создать подкласс `Application`. Он позволит вам получить информацию о жизненном цикле самого приложения. Создайте класс `CriminalIntentApplication`, который расширяет `Application` и переопределяет функцию `Application.onCreate()` для инициализации репозитория.

Листинг 11.10. Создание подкласса приложения (CriminalIntentApplication.kt)

```
class CriminalIntentApplication : Application()
{

    override fun onCreate() {
        super.onCreate()
        CrimeRepository.initialize(this)
    }
}
```

Как и `Activity.onCreate(...)`, функция `Application.onCreate()` вызывается системой, когда

приложение впервые загружается в память. Это как раз подходящее место для разовой инициализации.

Экземпляр приложения не будет постоянно уничтожаться и создаваться вновь, в отличие от activity или фрагментов классов. Он создается, когда приложение запускается, и уничтожается, когда завершается процесс приложения. Единственная функция жизненного цикла, которую вам надо переопределить в CriminalIntent, — это `OnCreate()`.

Сейчас нам нужно передать экземпляр приложения в репозиторий в качестве объекта `Context`. Этот объект действителен до тех пор, пока процесс приложения находится в памяти, поэтому можно безопасно хранить ссылку на него в классе репозитория.

Чтобы класс приложения можно было использовать в системе, необходимо зарегистрировать его в манифесте. Откройте файл `AndroidManifest.xml` и задайте свойство `android:name` для настройки приложения.

Листинг 11.11. Подключение подкласса приложения (manifests/AndroidManifest.xml)

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.bignerdranch.android.criminalintent">
    <application
            android:name=".CriminalIntentApplication"
            android:allowBackup="true"
            ... >
        ...
```

```
    </application>

</manifest>
```

Когда класс приложения будет зарегистрирован в манифесте, операционная система создаст экземпляр `CriminalIntentApplication` при запуске вашего приложения. ОС будет вызывать функцию `OnCreate()` на экземпляре `CriminalIntentApplication`. `CrimeRepository` будет инициализирован, и вы сможете обращаться к нему из других компонентов.

Теперь нужно добавить два свойства классу `CrimeRepository`, чтобы он умел хранить ссылки на базу данных и объекты DAO.

Листинг 11.12. Настройка свойств репозитория (CrimeRepository.kt)

```
private const val DATABASE_NAME = "crime-database"

class CrimeRepository private constructor(context: Context) {

    private val database : CrimeDatabase = Room.databaseBuilder(
        context.applicationContext,
        CrimeDatabase::class.java,
        DATABASE_NAME
    ).build()

    private val crimeDao = database.crimeDao()
```

```
    companion object {
        ...
    }
}
```

Функция `Room.databaseBuilder()` создает конкретную реализацию вашего абстрактного класса `CrimeDatabase` с использованием трех параметров. Сначала ему нужен объект `Context`, так как база данных обращается к файловой системе. Контекст приложения нужно передавать, так как синглтон, скорее всего, существует дольше, чем любой из ваших классов activity.

Второй параметр — это класс базы данных, которую Room должен создать. Третий — имя файла базы данных, которую создаст Room. Нужно использовать приватную константу, определенную в том же файле, поскольку никакие другие компоненты не должны получать к ней доступ.

Затем нужно заполнить ваш `CrimeRepository`, чтобы остальные компоненты могли выполнять любые операции, необходимые для работы с базой данных. Добавьте функцию в репозиторий для каждой функции в вашем DAO.

Листинг 11.13. Добавление функций репозитория (CrimeRepository.kt)

```
class CrimeRepository private constructor(context: Context) {

    ...
    private val crimeDao = database.crimeDao()
```

```
            fun getCrimes(): List<Crime> =
    crimeDao.getCrimes()

            fun getCrime(id: UUID): Crime? =
    crimeDao.getCrime(id)

        companion object {
            ...
        }
    }
```

Поскольку Room обеспечивает реализацию запросов в DAO, мы будем обращаться к этим реализациям из вашего репозитория. Это поможет сохранить код репозитория простым и коротким.

Может показаться, что мы много сделали, а получили мало, поскольку репозиторий всего лишь вызывает функции через `CrimeDao`. Но не бойтесь: скоро мы добавим инкапсуляцию дополнительных задач, выполняемых репозиторием.

Тестирование запросов

Когда репозиторий установлен, нам необходимо выполнить последний шаг, прежде чем мы сможем проверить функции запроса. В настоящее время база данных пуста, потому что вы не еще добавили в нее преступления. Чтобы ускорить процесс, мы загрузим готовые файлы базы данных для заполнения. Файлы базы данных представлены в файле решений для этой главы (**www.bignerdranch.com/solutions/AndroidProgramming4e.zip**).

Вы можете программно генерировать и вставлять фиктивные данные в базу данных, как уже делали ранее. Но

дело в том, что мы еще не реализовали функцию DAO, позволяющую вставить в базу данных новую запись (мы будем делать это в главе 14). Загрузка готовой базы данных позволяет легко заполнять базу данных без лишнего изменения кода вашего приложения. Кроме того, это позволит вам попрактиковаться с панелью **DeviceFileExplorer** и просмотреть содержимое локальной системы хранения вашего эмулятора.

Отметим, что у этого метода тестирования есть один недостаток. Вы можете получить доступ только к приватным файлам приложения (то есть файлам базы данных), работая в эмуляторе (или на устройстве «с корневым доступом»). Вы не сможете загружать файлы базы данных на обычном устройстве без прав доступа.

Загрузка тестовых данных

У каждого приложения на устройстве Android есть каталог в *песочнице* устройства. Хранение файлов в изолированной программной среде защищает их от доступа других приложений и посторонних глаз пользователей (однако с корневым доступом пользователь может делать что угодно).

Каталог песочницы приложения является дочерним каталогом в `data/data` пакета приложения. В случае с CriminalIntent полный путь к каталогу будет иметь вид `data/data/com.bignerdranch.android.criminalintent`.

Чтобы загрузить файлы, убедитесь, что эмулятор работает, и откройте панель **DeviceFileExplorer** в Android Studio, нажав соответствующую вкладку в правом нижнем углу окна. На панели **DeviceFileExplorer** отобразятся все файлы, находящиеся на эмуляторе (рис. 11.2).

Чтобы найти папку CriminalIntent, откройте папку `data/data` и найдите папку с именем, соответствующим

идентификатору пакета проекта (рис 11.3). Файлы в этой папке являются приватными для вашего приложения по умолчанию, поэтому никакие другие приложения не могут их считывать. Именно сюда мы будем загружать файлы базы данных.

Чтобы загрузить папку `databases`, щелкните правой кнопкой мыши по папке с именем пакета и выберите пункт **Upload** в контекстном меню (рис. 11.4).

Вы увидите файловый навигатор, в котором можно выбрать файлы для загрузки. Перейдите в папку решений для этой главы. Убедитесь, что загрузили всю папку `databases`. Room нужно, чтобы файлы лежали в папке под названием `databases`, или он не будет работать. После завершения загрузки ваша папка песочницы приложения должна выглядеть, как показано на рис. 11.5.

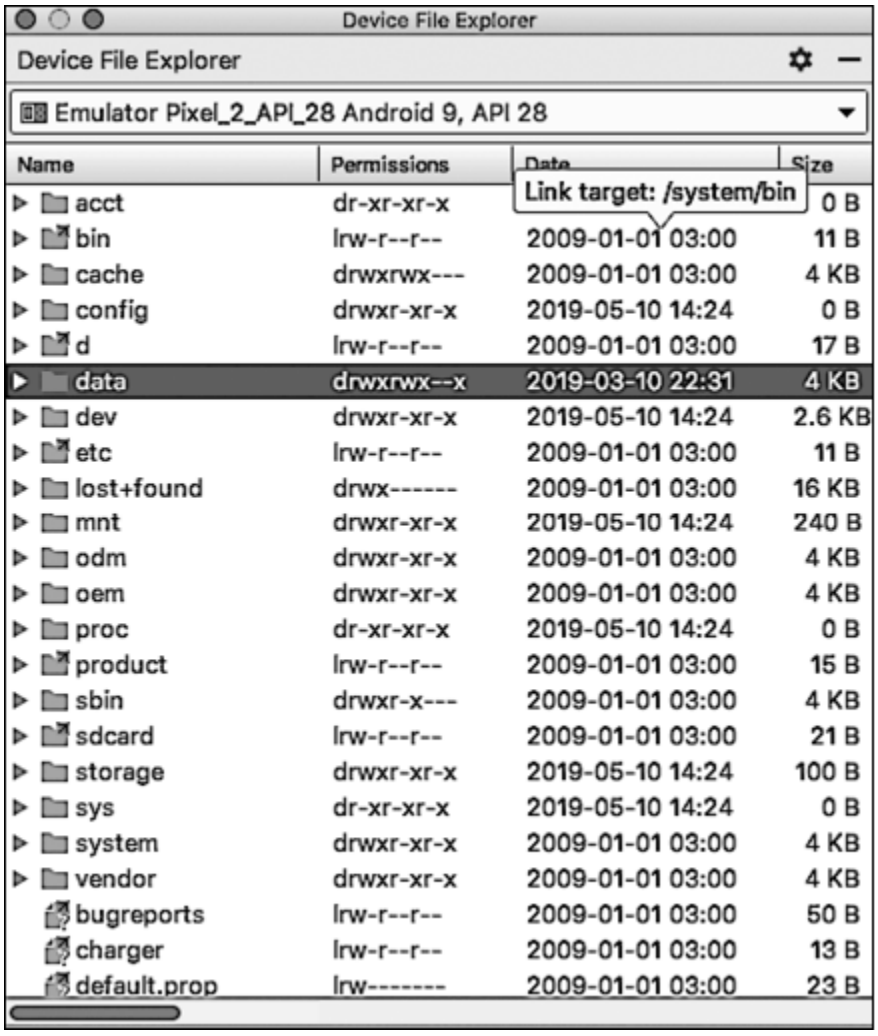


Рис. 11.2. Панель Device File Explorer

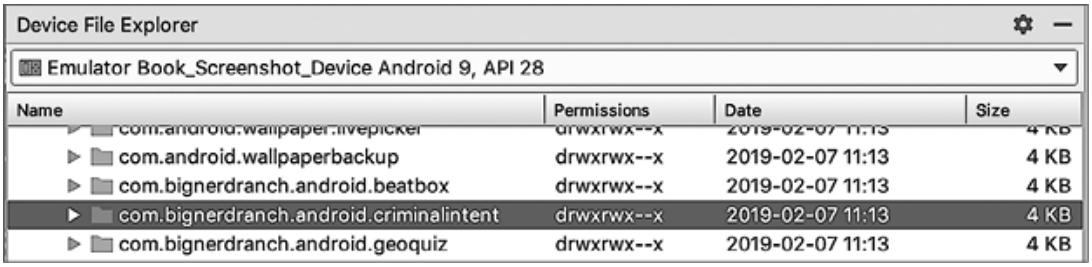


Рис. 11.3. Папка App в песочнице

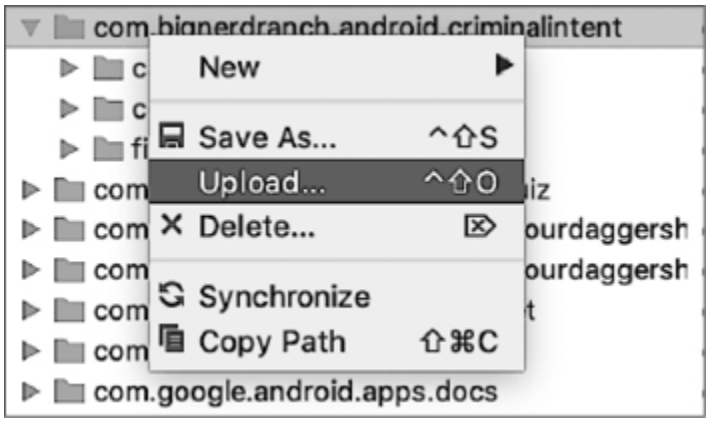


Рис. 11.4. Загрузка файлов базы данных

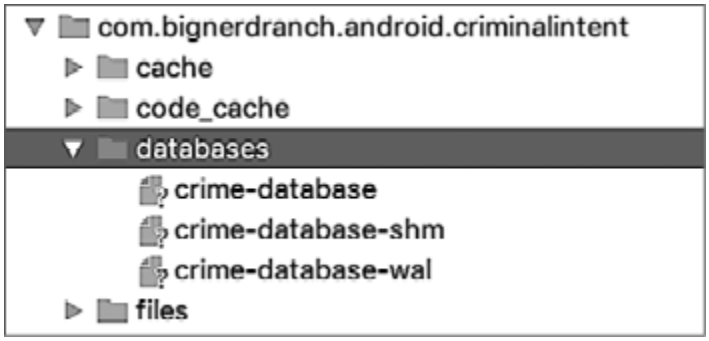


Рис. 11.5. После добавления файлов базы данных

После загрузки базы данных у вас есть данные, которые вы можете запросить с помощью репозитория. В настоящее время ваш `CrimeListViewModel` создаст 100 фиктивных преступлений и выведет их в список. Удалите код, который генерирует фиктивные преступления, и замените его вызовом функции `getCrimes()` на вашем `CrimeRepository`.

Листинг 11.14. Доступ к репозиторию в ViewModel (CrimeListViewModel.kt)

```
class CrimeListViewModel : ViewModel() {

    val crimes = mutableListOf<Crime>()
```

```
        init {
            for (i in 0 until 100) {
                val crime = Crime()
                crime.title = "Crime #$i"
                crime.isSolved = i % 2 == 0
                crimes += crime
            }
        }

        private val crimeRepository =
    CrimeRepository.get()
        val crimes = crimeRepository.getCrimes()
    }
```

Теперь запустите приложение, чтобы увидеть результат. Вы удивитесь, но ваше приложение… сломалось.

Не волнуйтесь, так и должно быть (по крайней мере, именно этого мы и ожидали). Посмотрите на исключение на панели **LogCat**, чтобы понять, что произошло.

```
java.lang.IllegalStateException: Cannot access
database on the main thread since it may
potentially lock the UI for a long period of
time.
```

Ошибка приходит из библиотеки Room. Room не нравится ваша попытка получить доступ к базе данных в основном потоке. В оставшейся части этой главы вы узнаете о том, как в Android используется потоковая модель, а также о назначении основного потока и какие задачи нужно выполнять в основном потоке. Кроме того, мы переместим взаимодействие с базой данных в фоновый поток. Room успокоится и уберет исключение, и все заработает.

Потоки приложения

Чтение из базы данных происходит не мгновенно. Поскольку доступ может занимать много времени, Room запрещает все операции с базой данных в основном потоке. Если вы попытаетесь нарушить это правило, Room выбросит исключение `IllegalStateException`, как вы уже видели выше.

Почему? Для ответа на этот вопрос надо понять, что такое поток и чем занимается основной поток.

Поток представляет собой единую последовательность выполнения. Код, работающий в одном потоке, будет выполняться пошагово. Любое приложение на Android начинает работу в основным потоке. Но основной поток не является конкретным списком действий. Он находится в бесконечном цикле и ожидает наступления событий, инициированных пользователем или системой. Затем он выполняет код в ответ на эти события (рис. 11.6).

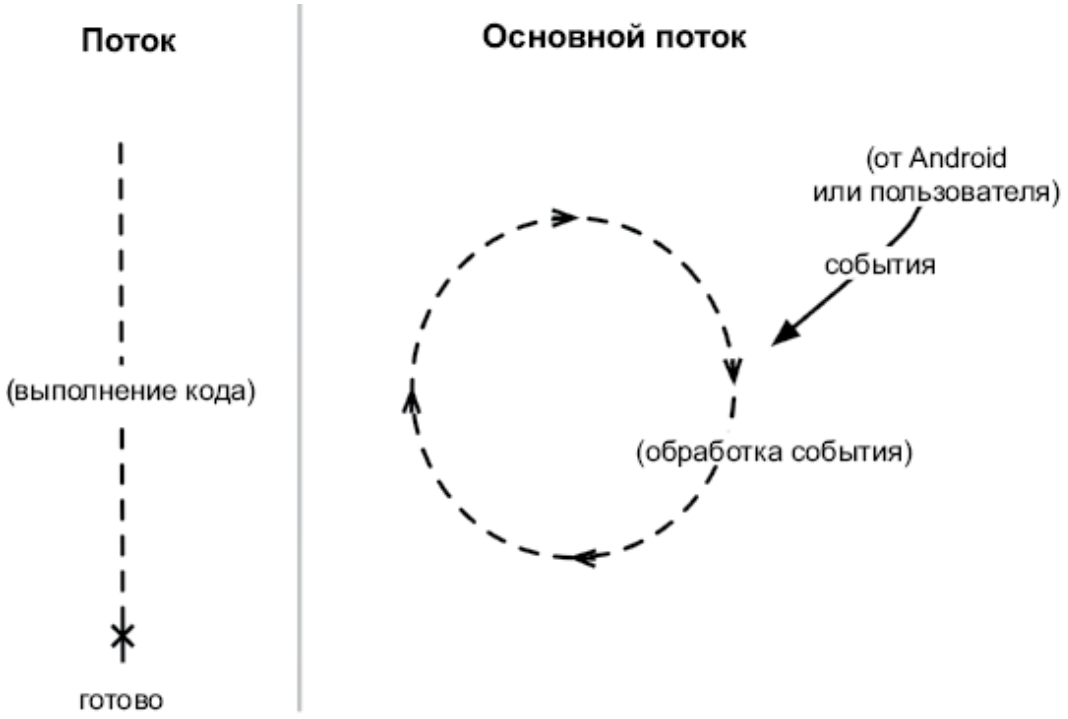


Рис. 11.6. Обычный поток и основной поток

Представьте, что ваше приложение — это огромный обувной магазин, и в нем всего один сотрудник — Флеш (герой DC Comics). В магазине надо много чего делать, чтобы клиенты были довольны: раскладывать товар, подбирать обувь для клиентов, работать с измерителем Браннока[2]. Флеш настолько быстрый, что успевает делать все вовремя.

Чтобы такая схема работала, Флеш не может тратить слишком много времени на одну задачу. Что делать, если партия обуви заканчивается? Кому-то придется долго сидеть на телефоне и решать вопрос. Тем временем клиенты будут гневаться и ждать, так как Флеш завис на другой задаче.

Флеш — это основной поток в приложении. Он обрабатывает весь код, который обновляет пользовательский интерфейс. Сюда входит код, который выполняется в ответ на различные связанные с пользовательским интерфейсом события — запуск activity, нажатие кнопки и так далее. (Поскольку все события в некотором роде связаны с пользовательским интерфейсом, основной поток иногда называют *UI-потоком*.)

Цикл событий выполняет код пользовательского интерфейса. Он обеспечивает, что ни одна из этих операций не накладывается на другую, в то же время гарантируя своевременное выполнение кода. До сих пор весь написанный нами код выполнялся в основном потоке.

Фоновые потоки

Доступ к базе данных похож на телефонный звонок вашему дистрибьютору обуви: он занимает много времени по сравнению с другими задачами. В течение этого времени пользовательский интерфейс перестанет отвечать на запросы, которые могут привести к *зависанию приложения*.

Сообщение «Application not responding» появляется тогда, когда сторожевой таймер Android видит, что основной поток не смог ответить на важное событие, например нажатие на кнопку. Для пользователя это выглядит так, как показано на рис. 11.7.

Рис. 11.7. Приложение не отвечает

В вашем магазине эта проблема решается (естественно) путем найма второго Флеша, который бы общался с поставщиком обуви. В Android мы делаем нечто подобное —

создаем фоновый поток и получаем доступ к базе данных из него.

Есть два важных правила, которые следует учитывать при добавлении фоновых потоков:

- *Все долго выполняющиеся задачи должны выполняться в фоновом потоке*. Это гарантирует, что основной поток сможет спокойно работать с интерфейсом и не нервировать пользователей.

- *Пользовательский интерфейс обновляется только из основного потока*. Вы получите сообщение об ошибке, если попытаетесь изменить пользовательский интерфейс из фонового потока, поэтому нужно убедиться, что любые данные, полученные из фонового потока, направляются в основной поток для обновления пользовательского интерфейса.

Существует много способов выполнения работы в фоновом потоке на Android. Вы узнаете, как создать асинхронную суть запросов в главе 24, научитесь использовать `Handler` для выполнения многих небольших фоновых операций в главе 25, а также поработаете с WorkManager в главе 27.

Для приложения CriminalIntent мы будем использовать два варианта для выполнения ваших вызовов базы данных в фоновом режиме. В этой главе мы будем использовать `LiveData`, чтобы обернуть ваши запросы. В главах 12 и 14 для вставки и обновления данных будем использовать `Executor`.

Использование LiveData

LiveData — это класс — контейнер данных из Jetpack-библиотеки lifecycle-extensions. Room предназначен для работы с LiveData. Так как вы уже включили библиотеку lifecycle-extensions в файл app/build.gradle, у вас есть доступ к классу LiveData.

Цель LiveData состоит в том, чтобы упростить передачу данных между различными частями вашего приложения, например от фрагмента CrimeRepository, который должен отображать данные о преступности. LiveData также позволяет организовать передачу данных между потоками. Именно поэтому эта библиотека идеально подходит для соблюдения правил, которые мы изложили выше.

При настройке запросов в DAO на возврат LiveData Room будет автоматически выполнять эти запросы в фоновом потоке, а затем выводить результаты в объект LiveData. Вы можете настроить activity или фрагмент на наблюдение за LiveData, и в этом случае ваша activity или фрагмент будет уведомлен в основном потоке, когда они будут готовы.

В этой главе мы сосредоточились на использовании функции межпоточной связи LiveData для выполнения запросов к базе данных. Для начала откройте файл CrimeDao.kt и измените возвращаемый тип у функций запроса, так как они должны возвращать объект LiveData, который оборачивает оригинальный возвращаемый тип.

Листинг 11.15. Возвращение LiveData в DAO (database/CrimeDao.kt)

```
@Dao
interface CrimeDao {

    @Query("SELECT * FROM crime")
```

```
        ~~fun getCrimes(): List<Crime>~~
    fun getCrimes(): LiveData<List<Crime>>

        @Query("SELECT * FROM crime WHERE id=
(:id)")
        ~~fun getCrime(id: UUID): Crime?~~
    fun getCrime(id: UUID): LiveData<Crime?>
}
```

Возвращая экземпляр `LiveData` из вашего класса DAO, вы запускаете запрос в фоновом потоке. Когда запрос завершается, объект `LiveData` будет обрабатывать отправку данных преступлений в основной поток и сообщать о любых наблюдателях.

Кроме того, `CrimeRepository` должен возвращать объект `LiveData` из своих функций запроса.

Листинг 11.16. Возврат LiveData из хранилища (CrimeRepository.kt)

```
class CrimeRepository private
constructor(context: Context) {
    ...
    private val crimeDao = database.crimeDao()

        ~~fun getCrimes(): List<Crime> =
crimeDao.getCrimes()~~
    fun getCrimes(): LiveData<List<Crime>> =
crimeDao.getCrimes()

        ~~fun getCrime(id: UUID): Crime? =
crimeDao.getCrime(id)~~
```

```
    fun getCrime(id: UUID): LiveData<Crime?> =
crimeDao.getCrime(id)
    ...
}
```

Наблюдение за LiveData

Чтобы отобразить преступления из базы данных в списке на экране, нужно чтобы `CrimeListFragment` наблюдал за `LiveData`, возвращаемым из функции `CrimeRepository.getCrimes()`.

Сначала откройте файл `CrimeListViewModel.kt` и переименуйте свойство `crimes`, чтобы стало яснее, что оно делает.

Листинг 11.17. Доступ к хранилищу в ViewModel (CrimeListViewModel.kt)

```
class CrimeListViewModel : ViewModel() {

    private val crimeRepository =
CrimeRepository.get()
    val ~~crimes~~ crimeListLiveData =
crimeRepository.getCrimes()
}
```

Затем очистите `CrimeListFragment`, чтобы отразить тот факт, что `CrimeListViewModel` сейчас открывает `LiveData`, возвращенный из хранилища (листинг 11.18). Удалите реализацию `OnCreate(...)`, так как она ссылается на `CrimeListViewModel.crimes`, которого больше не существует. В `updateUI()` удалите ссылку на `CrimeList-`

ViewModel.crimes и добавьте параметр, позволяющий брать на вход список преступлений.

Наконец, удалите вызов updateUI() из onCreateView(...). Вызов updateUI() будет выполняться из другого места.

Листинг 11.18. Удаления ссылки на старую версию ViewModel (CrimeListFragment.kt)

```
private const val TAG = "CrimeListFragment"
class CrimeListFragment : Fragment() {
    ...
                                override fun
onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
                        Log.d(TAG, "Total crimes:
${crimeListViewModel.crimes.size}")
        }
    ...
    override fun onCreateView(
        ...
    ): View? {
        ...
              crimeRecyclerView.layoutManager =
LinearLayoutManager(context)

            updateUI()

        return view
    }
```

```
        ~~private fun updateUI() {~~
    private fun updateUI(crimes: List<Crime>) {
                                ~~val crimes =~~
~~crimeListViewModel.crimes~~
        adapter = CrimeAdapter(crimes)
        crimeRecyclerView.adapter = adapter
    }
    ...
}
```

Теперь обновим класс `CrimeListFragment`, чтобы он наблюдал за объектом `LiveData`, который оборачивает список преступлений, возвращенный из базы данных. Так как фрагмент будет ждать результатов из базы данных, прежде чем он сможет заполнить утилизатор (recycler view) преступлениями, инициализируйте адаптер утилизатора пустым списком преступлений.

Затем настройте адаптер утилизатора на новый список преступлений, когда новые данные будут опубликованы в `LiveData`.

Листинг 11.19. Подсоединение RecyclerView (CrimeListFragment.kt)

```
private const val TAG = "CrimeListFragment"

class CrimeListFragment : Fragment() {

      private lateinit var crimeRecyclerView:
RecyclerView
        ~~private var adapter: CrimeAdapter? =~~
~~null~~
```

```
        private var adapter: CrimeAdapter? =
CrimeAdapter(emptyList())
    ...
    override fun onCreateView(
        ...
    ): View? {
        ...
             crimeRecyclerView.layoutManager =
LinearLayoutManager(context)
        crimeRecyclerView.adapter = adapter
        return view
    }

       override fun onViewCreated(view: View,
savedInstanceState: Bundle?) {
                    super.onViewCreated(view,
savedInstanceState)
        crimeListViewModel.crimeListLiveData.ob
serve(
            viewLifecycleOwner,
            Observer { crimes ->
                crimes?.let {
                        Log.i(TAG, "Got crimes
${crimes.size}")
                    updateUI(crimes)
                }
        })
    }
    ...
}
```

Функция
`LiveData.observe(LifecycleOwner,Observer)`
используется для регистрации наблюдателя за экземпляром `LiveData` и связи наблюдения с жизненным циклом другого компонента.

Второй параметр функции `observe(...)` — это реализация `Observer`. Этот объект отвечает за реакцию на новые данные из `LiveData`. В этом случае блок кода наблюдателя выполняется всякий раз, когда обновляется список в `LiveData`. Наблюдатель получает список преступлений из `LiveData` и печатает сообщение журнала, если свойство не равно нулю.

Если вы не снимали наблюдателя с прослушивания изменений в `LiveData`, реализация наблюдателя может попытаться обновить элемент вашего фрагмента, когда представление находится в нерабочем состоянии (например, когда элемент уничтожается). И если вы пытаетесь обновить недопустимый элемент, ваше приложение может сломаться.

Здесь на сцену выходит параметр `LifecycleOwner` из `LiveData.observe(...)`. Время жизни наблюдателя длится столько же, сколько и у компонента, представленного `LifecycleOwner`. В примере выше наблюдатель связан с фрагментом.

Пока *владелец жизненного цикла*, которому вы передаете наблюдателя, находится в допустимом состоянии жизненного цикла, объект `LiveData` уведомляет наблюдателя о получении новых данных. Объект `LiveData` автоматически снимает наблюдателя, когда соответствующий жизненный цикл становится недоступен. Поскольку `LiveData`*реагирует на изменения в жизненном цикле*, он относится к категории *lifecycle-aware component*. Вы узнаете больше о таких компонентах в главе 25.

Владелец жизненного цикла — это компонент, который реализует интерфейс `LifecycleOwner` и содержит объект жизненного цикла. Жизненный цикл — это объект, который отслеживает текущее состояние Android на протяжении жизненного цикла (напомним, что activity, фрагменты, элементы и даже само приложение имеют свой собственный жизненный цикл). Состояния жизненного цикла, такие как создание и возобновление, перечислены в `Lifecycle.State`. Вы можете запросить состояние жизненного цикла, используя функцию `Lifecycle.getCurrentState()`, или подписаться на уведомления об изменениях состояния.

`Fragment` в AndroidX является владельцем жизненного цикла непосредственно — `Fragment` реализует `LifecycleOwner` и имеет объект `Lifecycle`, представляющий состояние жизненного цикла экземпляра фрагмента.

Жизненный цикл вида фрагмента является собственностью и отслеживается отдельно во `FragmentViewLifecycleOwner`. Каждый `Fragment` имеет экземпляр `Fragment-ViewLifecycleOwner`, который отслеживает жизненный цикл этого фрагмента.

В приведенном выше коде мы определим области наблюдения, чтобы отслеживать представления жизненного цикла фрагмента, а не сам фрагмент, передав `viewLifecycleOwner` функции `observe(...)`. Жизненный цикл представления фрагмента, хотя и отделен от жизненного цикла самого экземпляра фрагмента, отражает жизненный цикл фрагмента. Можно изменить поведение по умолчанию, *удержав* фрагмент (что мы не будем делать в CriminalIntent). Вы узнаете больше о жизненном цикле представления и удержании фрагментов в главе 25.

`Fragment.onViewCreated(...)` вызывается после возврата `Fragment.onCreateView(...)`, давая понять, что

иерархия представления фрагмента находится на месте. Мы наблюдали за `LiveData` от `onViewCreated(...)`, чтобы убедиться, что виджет готов к отображению данных о преступлении. Именно поэтому вы передаете функции `observe()` объект `viewLifecycleOwner`, а не сам фрагмент. Нам нужно получать список преступлений только в том случае, если представление в должном состоянии, и тогда вы не будете получать уведомления, когда виджет не находится на экране.

Когда список преступлений готов, наблюдатель выводит сообщение и посылает список в `updateUI()`для подготовки адаптера.

Теперь запустите CriminalIntent. Ошибок больше быть не должно. Вместо этого вы должны увидеть преступления из файла базы данных, загруженного в ваш эмулятор (рис. 11.8).

В следующей главе мы подключим список преступлений к экрану и заполним этот экран подробными данными.

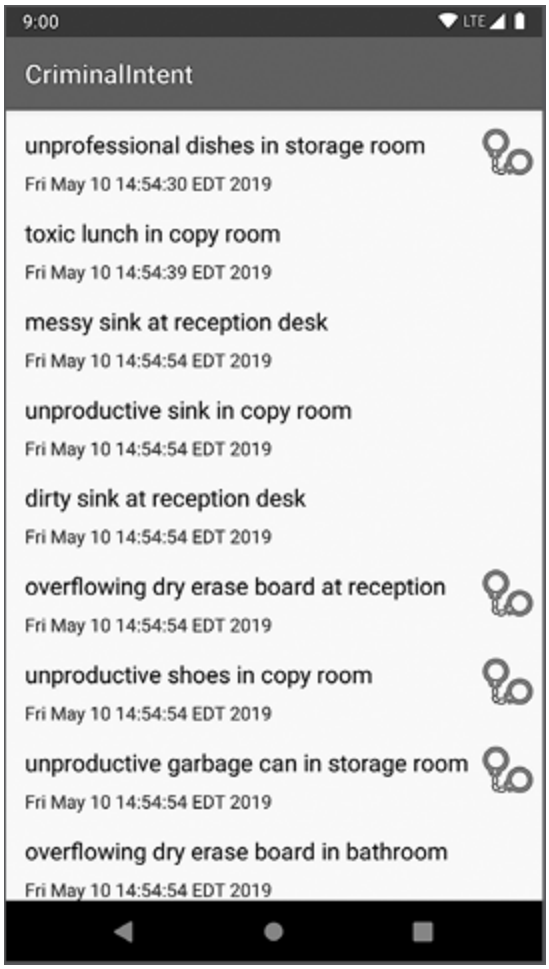


Рис. 11.8. База данных преступлений

Упражнение. Ошибка доступа к схеме

Если вы посмотрите журналы сборки, вы увидите предупреждение о том, что ваше приложение не предоставляет каталог для экспорта схемы:

```
warning: Schema export directory is not
provided to the annotation processor so we
cannot export the schema. You can either
provide `room.schemaLocation` annotation
processor argument OR set exportSchema to
false.
```

Схема базы данных — это структура базы данных, а именно: какие таблицы в базе данных, какие столбцы в этих таблицах, а также любые ограничения и отношения между этими таблицами. Room поддерживает экспорт схемы базы данных в файл, чтобы ее можно было хранить в системе управления версиями. Экспорт схемы часто бывает полезен, чтобы иметь разные версии вашей базы данных.

Предупреждение означает, что вы не указали местоположение файла, где Room мог бы сохранить схему базы данных. Вы можете указать местоположение схемы для аннотации `@Database` либо отключить экспорт, чтобы удалить предупреждение. Для данного упражнения можно выбрать один из этих вариантов.

Чтобы указать место для экспорта, нужно указать путь для свойства обработчика аннотаций `room.schemaLocation`. Для этого добавьте блок `kapt{}` в файл `app/build.gradle`:

```
...
android {
    ...
    buildTypes {
        ...
    }
    kapt {
        arguments {
                    arg("room.schemaLocation",
"some/path/goes/here/")
        }
    }
}
...
```

Чтобы отключить экспорт, присвойте значение `false` свойству `exportSchema`:

```
@Database(entities = [ Crime::class ], version=1, exportSchema = false)
@TypeConverters(CrimeTypeConverters::class)
abstract class CrimeDatabase : RoomDatabase() {

    abstract fun crimeDao(): CrimeDao
}
```

Для любознательных: синглтоны

Шаблон синглтон, используемый в `CrimeRepository` — это очень распространенное явление в Android. У синглтонов (одноэлементных классов) не очень хорошая репутация, потому что их часто неправильно используют, портя тем самым обслуживаемость приложения.

Синглтоны часто используются в Android, потому что они живут дольше, чем фрагмент или activity. Синглтон продолжает существовать после вращения и переключения между разными activity и фрагментами в приложении.

Синглтоны также удобны как владельцы ваших объектов модели. Представьте себе более сложное приложение CriminalIntent с большим количеством activity и фрагментов для модификации преступлений. Когда один контроллер модифицирует преступления, как убедиться, что обновленное преступление было направлено на другие контроллеры?

Если `CrimeRepository` — владелец преступлений и все модификации преступления проходят через него, вносить изменения становится гораздо проще. При переключении между контроллерами вы можете передать ID преступления в

качестве идентификатора конкретного преступления и заставить каждый контроллер забирать преступление целиком из `CrimeRepository` с помощью этого ID.

Однако у синглтонов есть несколько недостатков. У них, несмотря на возможность хранения данных, ограниченный срок жизни. Синглтон уничтожается вместе со всеми переменными экземпляра, так как Android в какой-то момент после переключения из приложения высвобождает память. То есть они не подходят для долгосрочного хранения (тут нужна запись файлов на диск или отправка на веб-сервер).

Синглтоны также затрудняют проведение модульных тестов (мы поговорим о модульном тестировании в главе 20). Не существует хорошего способа заменить экземпляр `CrimeRepository` болванкой. На практике разработчики Android обычно решают эту проблему с помощью инструмента под названием *инжектор зависимости*. Этот инструмент позволяет объектам быть общими как синглтоны, в то же время давая возможность заменить их в случае необходимости. Чтобы узнать больше об инъекции зависимостей, прочитайте раздел «Для любознательных: управление зависимостями» в главе 24.

И как мы уже говорили, у синглтонов есть потенциал неправильного использования. Заманчиво использовать их везде и всюду, потому что они удобны, — вы можете добраться до них откуда угодно и хранить в них любую информацию, которую нужно получить позже. Но когда вы делаете это, вы избегаете ответов на важные вопросы: где используются эти данные? где эта функция важна?

Синглтоны не отвечают на эти вопросы. В итоге их повсеместное использование превращает программу в большую свалку — непонятно, где что находится и зачем.

Таким образом, синглтоны могут стать ключевым компонентом хорошо спроектированного приложения (при

правильном использовании).

2 Измеритель Браннока — прибор, созданный Чарльзом Бранноком в 1927 году, для измерения длины стопы

12. Навигация по фрагментам

В этой главе мы наладим совместную работу списка и детализации в приложении CriminalIntent. Когда пользователь щелкает на элементе списка преступлений, `MainActivity` выгружает `CrimeListFragment` и выводит новый экземпляр `CrimeFragment` с подробной информацией о конкретном экземпляре Crime (рис. 12.1).

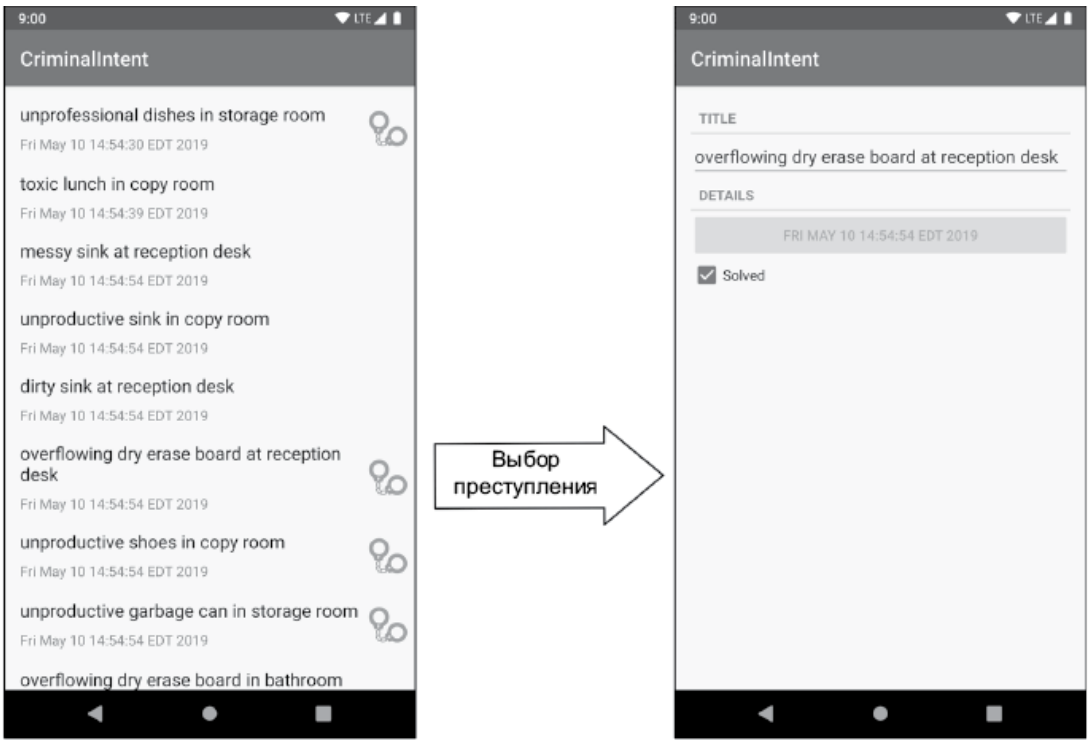


Рис. 12.1. Передача CrimeListFragment в CrimeFragment

Чтобы это заработало, нам надо реализовать навигацию, заставляя хост-activity заменять фрагменты в ответ на действия пользователя. Также вы научитесь передавать данные экземпляру фрагмента с помощью аргументов фрагмента. Наконец, вы узнаете, как использовать *преобразования* `LiveData` для загрузки неизменяемых данных в ответ на изменения пользовательского интерфейса.

Одиночная activity: главный фрагмент

В приложении GeoQuiz одна activity (`MainActivity`) запускала другую (`CheatActivity`). В CriminalIntent мы реализуем *архитектуру «одна activity»*. В приложении с такой архитектурой — всего одна activity и несколько фрагментов. Задача activity заключается в загрузке и выгрузке фрагментов в ответ на пользовательские события.

Для реализации навигации от `CrimeListFragment` к `CrimeFragment` в ответ на нажатие пользователем на преступление в списке можно подумать об инициировании операции с фрагментами в менеджер фрагментов хост-activity в функции `CrimeHolder.onClick(View)` у `CrimeListFragment`. Функция `onClick(View)` получит `FragmentManager` от `MainActivity` и совершит операцию с фрагментом, который заменит `CrimeListFragment` на `CrimeFragment`.

Код в `CrimeListFragment.CrimeHolder` будет выглядеть следующим образом.

```
fun onClick(view: View) {
                        val      fragment      =
CrimeFragment.newInstance(crime.id)
    val fm = activity.supportFragmentManager
    fm.beginTransaction()
                .replace(R.id.fragment_container,
fragment)
        .commit()
}
```

Это работает, но крутые программисты на Android делают не так. Фрагменты предназначены быть автономными составными единицами. Если вы пишете фрагмент, который

добавляет фрагменты во FragmentManager activity, то этот фрагмент «предполагает», как работает хост-activity, и больше не является самостоятельной единицей.

Например, в приведенном выше коде CrimeListFragment добавляет фрагмент CrimeFragment в MainActivity и «предполагает», что в компоновке MainActivity есть fragment_container. Эта задача обрабатывается хост-activity CrimeListFragment, а не CrimeListFragment.

Для сохранения независимости ваших фрагментов вы будете делегировать работу обратно в хост-activity, определяя во фрагментах интерфейсы обратного вызова. Хост-activity реализует эти интерфейсы для выполнения задач по надзору за фрагментами и поведения, зависящего от компоновки.

Интерфейсы обратного вызова фрагментов

Для передачи функциональности обратно хостингу в фрагменте обычно определяется пользовательский интерфейс обратного вызова под именем Callbacks. Этот интерфейс определяет работу, которую должна выполнить хост-activity. Любая activity, которая будет содержать фрагмент, должна реализовывать этот интерфейс.

С помощью интерфейса обратного вызова фрагмент способен вызывать функции, связанные с его хост-activity, без необходимости знать что-либо о том, какая activity является хостом.

Интерфейс обратного вызова позволяет передавать события кликов из CrimeListFragment обратно на хост-activity. Откройте CrimeListFragment и определите интерфейс обратного вызова с одной функцией обратного вызова. Добавьте свойство callbacks, чтобы удерживать объект, реализующий Callbacks. Переопределите функции

onAttach(Context) и onDetach() для установки и отмены свойства callbacks.

Листинг 12.1. Добавление интерфейса обратного вызова (CrimeListFragment.kt)

```
class CrimeListFragment : Fragment() {

    /**
     * Требуемый интерфейс
     */
     interface Callbacks {
         fun onCrimeSelected(crimeId: UUID)
     }

     private var callbacks: Callbacks? = null
       private lateinit var crimeRecyclerView:
RecyclerView
        private var adapter: CrimeAdapter =
CrimeAdapter(emptyList())
            private val crimeListViewModel:
CrimeListViewModel by lazy {
        ViewModelProviders.of(this).get(CrimeLi
stViewModel::class.java)
    }

     override fun onAttach(context: Context) {
         super.onAttach(context)
         callbacks = context as Callbacks?
     }
```

```
    override fun onCreateView(
        ...
    ): View? {
        ...
    }

    override fun onViewCreated(view: View,
savedInstanceState: Bundle?) {
        ...
    }

    override fun onDetach() {
        super.onDetach()
        callbacks = null
    }
    ...
}
```

Функция жизненного цикла `Fragment.onAttach(Context)` вызывается, когда фрагмент прикреплется к activity. Здесь вы помещаете аргумент `Context`, переданный функции `onAttach(...)`, в свойство `callback`. Так как `CrimeListFragment` размещается в activity, то объект `Context`, переданный `onAttach(...)`, является экземпляром activity, в которой размещен фрагмент.

Помните, что `Activity` является подклассом `Context`, поэтому функция `onAttach(...)` передает в качестве параметра объект `Context`, который более гибок. Убедитесь, что для `onAttach(...)` используется сигнатура `onAttach(Context)`, а не устаревшая функция

onAttach(Activity), которая может быть удалена в будущих версиях API.

Аналогичным образом нужно установить переменной значение null в соответствующей функции жизненного цикла Fragment.onDetach(). Здесь переменную устанавливают равной нулю, так как в дальнейшем вы не сможете получить доступ к activity или рассчитывать на то, что она будет продолжать существовать.

Обратите внимание, что CrimeListFragment выполняет неконтролируемую передачу своей activity в CrimeListFragment.Callbacks. Это означает, что хост-activity должна реализовывать функцию CrimeListFragment.Callbacks. Это правильная зависимость, но важно ее задокументировать.

Теперь у CrimeListFragment есть возможность вызывать функции на хост-activity. Не имеет значения, какой activity занимается хостинг. Пока эта activity реализует CrimeListFragment.Callbacks, все в CrimeListFragment может работать одинаково.

Теперь нужно обновить слушателя кликов для отдельных элементов в списке преступлений таким образом, чтобы нажатие на преступление уведомляло хост-activity через интерфейс Callbacks. Вызовите функцию onCrimeSelected(Crime) в CrimeHolder.onClick(View).

Листинг 12.2. Вызов всех обратных вызовов! (CrimeListFragment.kt)

```
class CrimeListFragment : Fragment() {
    ...
    private inner class CrimeHolder(view: View)
```

```
                : RecyclerView.ViewHolder(view),
View.OnClickListener {
        ...
        fun bind(crime: Crime) {
            ...
        }

        override fun onClick(v: View?) {
                        Toast.makeText(context,
"${crime.title} clicked!", Toast.LENGTH_SHORT)
                    .show()
            callbacks?.onCrimeSelected(crime.id
)
        }
    }
    ...
}
```

Наконец, обновите `MainActivity` для реализации `CrimeListFragment.Callbacks`. Запишите отладочный отчет в журнал `onCrimeSelected(UUID)`.

Листинг 12.3. Реализация обратных вызовов (MainActivity.kt)

```
private const val TAG = "MainActivity"
class MainActivity : AppCompatActivity(),
    CrimeListFragment.Callbacks {

     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
    }
```

```
    override fun onCrimeSelected(crimeId: UUID)
{
                                        Log.d(TAG,
"MainActivity.onCrimeSelected: $crimeId")
    }
}
```

Запустите приложение CriminalIntent. Выполните поиск и отфильтруйте результаты на панели **Logcat** для просмотра отчетов журнала `MainActivity`. Каждый раз, когда вы нажимаете на преступление в списке, вы должны видеть отчет журнала, показывающий, что событие клика было передано из `CrimeListFragment` в `MainActivity` через `Callbacks.onCrimeSelected(UUID)`.

Замена фрагмента

Теперь, когда интерфейс обратного вызова подключен корректно, обновите в `MainActivityonCrimeSelected(UUID)`, чтобы заменить `CrimeListFragment` на экземпляр `CrimeFragment`, когда пользователь нажимает на преступление в списке `CrimeListFragment`. Пока что будем игнорировать идентификатор преступления, переданный на обратный вызов.

Листинг 12.4. Замена CrimeListFragment на CrimeFragment (MainActivity.kt)

```
class MainActivity : AppCompatActivity(),
    CrimeListFragment.Callbacks {
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        ...
    }

    override fun onCrimeSelected(crimeId: UUID)
{
                                        Log.d(TAG,
"MainActivity.onCrimeSelected: $crimeId")
        val fragment = CrimeFragment()
        supportFragmentManager
            .beginTransaction()
              .replace(R.id.fragment_container,
fragment)
            .commit()
    }
}
```

Функция `FragmentTransaction.replace(Int,Fragment)` заменяет фрагмент, размещенный в activity (в контейнере с указанным целочисленным идентификатором ресурса), на новый фрагмент. Если фрагмент еще не размещен в указанном контейнере, то добавляется новый фрагмент, как если бы вы вызвали `FragmentTransaction.add(Int,Fragment)`.

Запустите приложение CriminalIntent. Нажмите на преступление в списке. Должен появиться экран подробной информации о преступлении (рис. 12.2).

Пока что экран с деталями преступления пуст, потому что вы не сказали `CrimeFragment`, какое именно преступление показывать. В ближайшее время мы это исправим. Но сначала

вам нужно отшлифовать оставшийся острый край в вашей реализации навигации.

Нажмите кнопку «Назад». Activity `MainActivity` будет закрыта. Это потому, что во внутренней кухне вашего приложения есть лишь экземпляр `MainActivity`, который был запущен при запуске приложения.

Рис. 12.2. Пустой CrimeFragment

Пользователи будут ожидать, что нажатие кнопки «Назад» на экране подробностей преступления вернет их обратно к списку преступлений. Чтобы реализовать это поведение, добавьте транзакцию замены в обратный стек.

Листинг 12.5. Добавление транзакции фрагмента в обратный стек (MainActivity.kt)

```
class MainActivity : AppCompatActivity(),
    CrimeListFragment.Callbacks {
    ...
    override fun onCrimeSelected(crimeId: UUID)
{
        val fragment = CrimeFragment()
        supportFragmentManager
            .beginTransaction()
              .replace(R.id.fragment_container,
fragment)
            .addToBackStack(null)
            .commit()
    }
}
```

Теперь при нажатии пользователем кнопки «Назад» транзакция будет обращена. Таким образом, `CrimeFragment` будет заменен на `CrimeListFragment`.

Вы можете дать название состоянию обратного стека, передав строку во `FragmentTransaction.addToBackStack(String)`. Делать это необязательно, и так как в этой реализации это не нужно, вы передаете `null`.

Запустите ваше приложение. Выберите преступление из списка, чтобы запустить `CrimeFragment`. Нажмите кнопку «Назад», чтобы вернуться к `CrimeListFragment`. Наслаждайтесь простой навигацией по приложению, которую и ожидают пользователи.

Аргументы фрагментов

`CrimeListFragment` теперь уведомляет хост-activity (`MainActivity`) о выборе преступления и передает идентификатор выбранного преступления.

Это все прекрасно и стильно. Но на самом деле вам нужен способ передать выбранный идентификатор преступления из `MainActivity` в `CrimeFragment`. Таким образом, `CrimeFragment` сможет извлечь данные для этого преступления из базы данных и заполнить пользовательский интерфейс этими данными.

Аргументы фрагментов позволяют хранить фрагменты данных в том месте, которое принадлежит фрагменту. Это самое место называется *пакетом аргументов*. Фрагмент может получить данные из пакета аргументов, не полагаясь на свою родительскую activity или другой внешний источник.

Аргументы фрагментов помогают сохранить инкапсуляцию фрагмента. Хорошо инкапсулированный фрагмент — это многоразовый кусок кода, который можно будет использовать в любой activity.

Чтобы создать аргументы фрагментов, вы сначала создаете объект `Bundle` (пакет). Этот пакет содержит пары «ключ — значение», которые работают точно так же, как и интенты activity. Каждая пара известна как аргумент. Мы используем `put`-функции `Bundle` соответствующего типа (по аналогии с функциями `Intent`) для добавления аргументов в пакет:

```
val args = Bundle().apply {
    putSerializable(ARG_MY_OBJECT, myObject)
    putInt(ARG_MY_INT, myInt)
    putCharSequence(ARG_MY_STRING, myString)
}
```

Каждый экземпляр фрагмента может иметь прикрепленные к нему объекты Bundle.

Присоединение аргументов к фрагменту

Чтобы присоединить пакет аргументов к фрагменту, вызовите функцию Fragment.setArguments(Bundle). Присоединение должно быть выполнено после создания фрагмента, но до его добавления в activity.

Для этого программисты Android используют схему с добавлением в класс Fragment статической функции newInstance(...). Эта функция создает экземпляр фрагмента, упаковывает и задает его аргументы.

Когда хост-activity потребуется экземпляр этого фрагмента, она вместо прямого вызова конструктора вызывает функцию newInstance(...). Activity может передать newInstance(...) любые параметры, необходимые фрагменту для создания аргументов.

Включите в CrimeFragment функцию newInstance(UUID), который получает UUID, создает пакет аргументов, создает экземпляр фрагмента, а затем присоединяет аргументы к фрагменту.

Листинг 12.6. Функция newInstance(UUID) (CrimeFragment.kt)

```
private const val ARG_CRIME_ID = "crime_id"

class CrimeFragment : Fragment() {
    ...
    override fun onStart() {
        ...
    }
```

```
    companion object {

                fun newInstance(crimeId: UUID):
CrimeFragment {
            val args = Bundle().apply {
                  putSerializable(ARG_CRIME_ID,
crimeId)
            }
            return CrimeFragment().apply {
                arguments = args
            }
        }
    }
}
```

Теперь класс MainActivity должен вызывать CrimeFragment.newInstance(UUID) каждый раз, когда ему потребуется создать CrimeFragment. При вызове передается значение UUID, полученное из MainActivity.onCrimeSelected(UUID).

Листинг 12.7. Использование CrimeFragment.newInstance(UUID) (MainActivity.kt)

```
class MainActivity : AppCompatActivity(),
    CrimeListFragment.Callbacks {
    ...
    override fun onCrimeSelected(crimeId: UUID)
{
            val fragment = CrimeFragment()
```

```
                                val fragment =
CrimeFragment.newInstance(crimeId)
        supportFragmentManager
            .beginTransaction()
              .replace(R.id.fragment_container,
fragment)
            .addToBackStack(null)
            .commit()
    }
}
```

Учтите, что потребность в независимости не двусторонняя. Класс `MainActivity` должен многое знать о классе `CrimeFragment`, например то, что он содержит функцию `newInstance(UUID)`. Это нормально; хост-activity должна располагать конкретной информацией о том, как управлять фрагментами, но фрагментам такая информация об их activity не нужна (по крайней мере, если вы хотите сохранить гибкость независимых фрагментов).

Получение аргументов

Когда фрагменту требуется получить доступ к его аргументам, он ссылается на свойство `arguments` класса `Fragment`, а затем один из геттеров `Bundle` для конкретного типа.

В функции `CrimeFragment.onCreate(...)` нужно извлечь `UUID` из аргументов фрагмента. Запишем идентификатор в журнал, чтобы убедиться, что аргумент прикрепился правильно.

Листинг 12.8. Получение идентификатора преступления из аргументов (CrimeFragment.tk)

```
private const val TAG = "CrimeFragment"
private const val ARG_CRIME_ID = "crime_id"

class CrimeFragment : Fragment() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        crime = Crime()
                    val crimeId: UUID =
arguments?.getSerializable(ARG_CRIME_ID) as
UUID
            Log.d(TAG, "args bundle crime ID:
$crimeId")
        // Загрузка преступления из базы данных
    }
    ...
}
```

Запустите приложение CriminalIntent. Оно будет работать точно так же, но архитектура с независимостью `CrimeFragment` должна вызвать у вас приятные чувства.

Преобразования LiveData

Теперь, когда у `CrimeFragment` есть идентификатор преступления, ему нужно извлечь объект преступления из базы данных, чтобы отобразить данные о преступлении. Так как для этого требуется поиск по базе данных, который вы не хотите без необходимости повторять при повороте устройства, добавьте `CrimeDetailViewModel` для управления запросом к базе данных.

Когда CrimeFragment запрашивает из базы преступление с заданным идентификатором, его CrimeDetailViewModel должна запустить запрос к базе данных getCrime(UUID). После выполнения запроса CrimeDetailViewModel должна уведомить CrimeFragment и пройтись по объекту преступления, который возник в результате запроса.

Создайте новый класс под именем CrimeDetailViewModel и откройте свойство LiveData для сохранения объекта Crime, полученного из базы данных. Используйте LiveData для реализации отношений, при которых изменение идентификатора преступления вызывает новый запрос к базе данных.

Листинг 12.9. Добавление ViewModel в CrimeFragment (CrimeDetailViewModel.kt)

```
class CrimeDetailViewModel() : ViewModel() {

            private val crimeRepository =
CrimeRepository.get()
            private val crimeIdLiveData =
MutableLiveData<UUID>()

    var crimeLiveData: LiveData<Crime?> =
        Transformations.switchMap(crimeIdLiveDa
ta) { crimeId ->
            crimeRepository.getCrime(crimeId)
        }

    fun loadCrime(crimeId: UUID) {
        crimeIdLiveData.value = crimeId
    }
```

```
}
```

В свойстве `CrimeRepository` хранится связь с `CrimeRepository`. Сейчас в этом нет необходимости, но в дальнейшем `CrimeDetailViewModel` будет взаимодействовать с репозиторием более чем в одном месте, поэтому свойство окажется полезным.

`CrimeIdLiveData` хранит идентификатор отображаемого в данный момент преступления (или выводимого на отображение) фрагментом `CrimeFragment`. При первом создании `CrimeDetailViewModel` идентификатор преступления не устанавливается. В конце концов `CrimeFragment` вызовет функцию `CrimeDetailViewModel.loadCrime(UUID)`, чтобы `ViewModel` понял, какое преступление ему нужно загрузить.

Обратите внимание, что вы явно определили тип `crimeLiveData` как `LiveData<Crime?>`. Поскольку `crimeLiveData` публичен, нужно убедиться, что речь не идет о `MutableLiveData`. `ViewModel`-и никогда не должны выставлять публично `MutableLiveData`.

Может показаться странным метод обертывания идентификатора преступления в `LiveData`, так как он является приватным для `CrimeDetailViewModel`. Вы можете спросить: что именно в `CrimeDetailViewModel` отслеживает изменения значения приватного идентификатора?

Ответ заключается в операторе `Transformation` (преобразование). *Преобразование данных в реальном времени* — это способ установить отношения «триггер — ответ» между двумя объектами `LiveData`. Функция преобразования принимает два объекта: объект `LiveData`, используемый в качестве *триггера*, и *функцию отображения*, которая должна вернуть объект `LiveData`. Функция преобразования

возвращает новый объект LiveData, который мы называем результатом преобразования, значение которого обновляется каждый раз, когда изменяется значение триггерного объекта LiveData.

Значение результата преобразования вычисляется путем выполнения функции отображения. Свойство value объекта LiveData, возвращаемое из функции отображения, используется для установки свойства value для результата преобразования.

Такое использование преобразования означает, что фрагмент CrimeFragment должен лишь однажды наблюдать открытые данные CrimeDetailViewModel.crimeLiveData. Когда фрагмент изменяет идентификатор, который он хочет отобразить, ViewModel просто публикует новые данные о преступлении в существующем потоке данных.

Откройте файл CrimeFragment.kt. Проассоциируйте CrimeFragment с CrimeDetailViewModel. Запросите у ViewModel загрузку Crime в onCreate(...).

Листинг 12.10. Загрузка фрагмента CrimeFragment в CrimeDetailViewModel (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {

    private lateinit var crime: Crime
    ...
        private lateinit var solvedCheckBox:
CheckBox
        private val crimeDetailViewModel:
CrimeDetailViewModel by lazy {
        ViewModelProviders.of(this).get(CrimeDe
tailViewModel::class.java)
```

```
    }

      override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        crime = Crime()
                    val crimeId: UUID =
arguments?.getSerializable(ARG_CRIME_ID) as
UUID
              Log.d(TAG, "args bundle crime ID:
$crimeId")
              // Загрузка преступления из базы
данных
            crimeDetailViewModel.loadCrime(crim
eId)
    }
    ...
}
```

Теперь наблюдайте за значением `crimeLiveData` в `CrimeDetailViewModel` и обновляйте пользовательский интерфейс при публикации новых данных.

Листинг 12.11. Наблюдение за изменениями (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {

    private lateinit var crime: Crime
    ...

    override fun onCreateView(
        ...
```

```
    ): View? {
        ...
    }

       override fun onViewCreated(view: View,
savedInstanceState: Bundle?) {
                        super.onViewCreated(view,
savedInstanceState)
        crimeDetailViewModel.crimeLiveData.obse
rve(
            viewLifecycleOwner,
            Observer { crime ->
                crime?.let {
                    this.crime = crime
                    updateUI()
                }
            })
    }

    override fun onStart() {
        ...
    }

    private fun updateUI() {
        titleField.setText(crime.title)
        dateButton.text = crime.date.toString()
                    solvedCheckBox.isChecked  =
crime.isSolved
    }
    ...
}
```

(Обязательно импортируйте `androidx.lifeecycle.Observer`.)

Вы, возможно, заметили, что `CrimeFragment` имеет свое состояние `Crime`, которое хранится в соответствующем свойстве `crime`. Значения в этом свойстве `crime` отражают правки, которые пользователь делает в данный момент. Преступление в `crimeDetailViewModel.crimeLiveData` — это данные в том виде, в котором они в настоящее время хранятся в базе данных. `CrimeFragment` «публикует» пользовательские правки, когда фрагмент переходит в остановленное состояние, записывая обновленные данные в базу данных.

Запустите приложение. Нажмите на преступление в списке. Если все пойдет по плану, появится экран с подробностями преступления, заполненный данными о том преступлении, на которое вы нажали в списке (рис. 12.3).

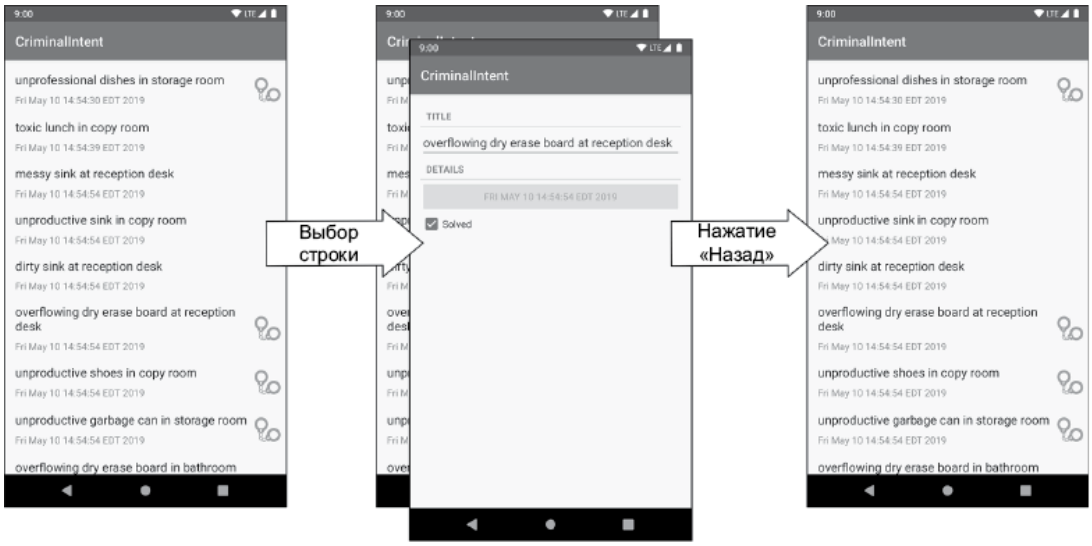


Рис. 12.3. Стек переходов назад CriminalIntent

При отображении фрагмента `CrimeFragment`, если вы просматриваете преступление, помеченное как раскрытое, вы можете увидеть флажок, анимированный до отмеченного

состояния. Так и должно быть, так как состояние чекбокса устанавливается в результате асинхронной операции. Запрос к базе данных для данного преступления начинается при первом запуске пользователем `CrimeFragment`. Когда запрос к БД завершается, наблюдатель фрагмента `crimeDetailViewModel.crimeLiveData` получает уведомление и, в свою очередь, обновляет данные, отображаемые в виджетах.

Исправьте это, пропустив анимацию, когда вы программно устанавливаете флажок в выбранное состояние. Для этого вызывается функция `View.jumpDrawables-ToCurrentState()`. Обратите внимание, что, если задержка при загрузке экрана с подробностями преступления слишком велика, вы можете загрузить данные о преступлении в память заранее (например, при запуске приложения) и спрятать их в общем месте. В CriminalIntent задержка очень мала, поэтому достаточно простого пропуска анимации.

Листинг 12.12. Пропуск анимации флажком (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {
    ...
    private fun updateUI() {
        titleField.setText(crime.title)
        dateButton.text = crime.date.toString()
                    ~~solvedCheckBox.isChecked = crime.isSolved~~
        solvedCheckBox.apply {
            isChecked = crime.isSolved
            jumpDrawablesToCurrentState()
        }
    }
```

```
    ...
}
```

Запустите ваше приложение снова. Нажмите на раскрытое преступление в списке. Обратите внимание, что флажок больше не анимируется при повороте экрана. Если вы нажмете на этот флажок, анимация воспроизведется, как и должна.

Теперь отредактируйте название преступления. Нажмите кнопку «Назад», чтобы вернуться к экрану списка преступлений. К сожалению, внесенные изменения не были сохранены. К счастью, это легко исправить.

Обновление базы данных

База данных служит единым источником данных о преступлениях. В CriminalIntent, когда пользователь выходит из экрана подробностей, все сделанные им правки должны сохраняться в базе данных (у других приложений на этот счет могут быть другие требования, например кнопка «Сохранить» или сохранение сразу во время ввода со стороны пользователя).

Сначала добавьте в `CrimeDao` функцию для обновления существующего преступления. Также добавьте функцию для вставки нового преступления. Пока что мы игнорируем функцию вставки, но будем использовать ее в главе 14, когда добавим в пользовательский интерфейс меню для создания новых преступлений.

Листинг 12.13. Добавление функций обновления и вставки в базу данных (database/CrimeDao.kt)

```
@Dao
```

```
interface CrimeDao {

    @Query("SELECT * FROM crime")
    fun getCrimes(): LiveData<List<Crime>>

       @Query("SELECT * FROM crime WHERE id=
(:id)")
    fun getCrime(id: UUID): LiveData<Crime?>

    @Update
    fun updateCrime(crime: Crime)

    @Insert
    fun addCrime(crime: Crime)
}
```

Аннотациям для этих функций не нужно никаких параметров. Room может сам сгенерировать соответствующую SQL-команду для этих операций.

В функции `updateCrime()` используется аннотация `@Update`. Эта функция принимает объект преступления, используя идентификатор, сохраненный в этом преступлении, чтобы найти соответствующую строку, а затем обновляет данные в этой строке, основываясь на новых данных в объекте преступления.

В функции `addCrime()` используется аннотация `@Insert`. Параметр — это преступление, которое вы хотите добавить в таблицу базы данных.

Теперь добавим в хранилище вызов новых функций вставки и обновления DAO. Вспомните, что Room автоматически выполняет запросы к базе данных `CrimeDao.getCrimes()` и

`CrimeDao.getCrime(UUID` в фоновом потоке, потому что эти функции DAO возвращают `LiveData`. В этих случаях `LiveData` обрабатывает пересылку данных и отправляет в ваш основной поток, чтобы вы могли обновить ваш пользовательский интерфейс.

Однако Room не будет автоматически запускать взаимодействие с базой данных для выполнения вставки и обновления. Вместо этого вам нужно будет явно выполнить вызовы DAO. Обычный способ сделать это — использовать исполнителя.

Использование исполнителя

Исполнитель — это объект, который ссылается на поток. Экземпляр исполнителя имеет функцию, называемую `execute`, которая принимает на выполнение блок кода. Код, который находится в этом блоке, будет выполняться в любом потоке, на который ссылается исполнитель.

Мы создадим исполнителя, использующего новый поток, который всегда будет фоновым. Любой код в блоке будет выполняться в этом потоке, так что вы сможете безопасно работать с базой данных в нем.

Вы не можете напрямую реализовать исполнителя в `CrimeDao`, потому что Room генерирует для вас реализацию функции на основе заданного вами интерфейса. Вместо этого реализуйте исполнителя в `CrimeRepository`. Добавьте свойство исполнителя для хранения ссылки, затем выполните функции вставки и обновления с помощью исполнителя.

Листинг 12.14. Вставка и обновление с помощью исполнителя (CrimeRepository.kt)

```
class          CrimeRepository          private
constructor(context: Context) {
    ...
    private val crimeDao = database.crimeDao()
                private    val    executor    =
Executors.newSingleThreadExecutor()

      fun getCrimes(): LiveData<List<Crime>> =
crimeDao.getCrimes()

     fun getCrime(id: UUID): LiveData<Crime?> =
crimeDao.getCrime(id)

    fun updateCrime(crime: Crime) {
        executor.execute {
            crimeDao.updateCrime(crime)
        }
    }

    fun addCrime(crime: Crime) {
        executor.execute {
            crimeDao.addCrime(crime)
        }
    }
    ...
}
```

Функция newSingleThreadExecutor() возвращает экземпляр исполнителя, который указывает на новый поток. Таким образом, любая работа, которую вы выполняете с исполнителем, будет происходить вне основного потока.

Как updateCrime(), так и addCrime() оборачивают вызовы в DAO внутри блока execute{}. Он выталкивает эти операции из основного потока, чтобы не блокировать работу пользовательского интерфейса.

Привязка записи в базу данных к жизненному циклу фрагмента

И последнее, но не по значению: обновите ваше приложение, чтобы записать в базу данных значения, которые пользователь вводит на экране детализации преступления, когда закрывает экран.

Откройте файл CrimeDetailViewModel.kt и добавьте функцию сохранения объекта преступления в базу данных.

Листинг 12.15. Добавление возможности сохранения (CrimeDetailViewModel.kt)

```
class CrimeDetailViewModel() : ViewModel() {
    ...
    fun loadCrime(crimeId: UUID) {
        crimeIdLiveData.value = crimeId
    }

    fun saveCrime(crime: Crime) {
        crimeRepository.updateCrime(crime)
    }
}
```

Функция saveCrime(Crime) принимает объект Crime и записывает его в базу данных. Поскольку CrimeRepository обрабатывает запрос на обновление в фоновом потоке, интеграция с базой данных реализуется просто.

Теперь обновите CrimeFragment, чтобы сохранить отредактированные пользователем данные о преступлении в базе данных.

Листинг 12.16. Сохранение в onStop() (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {
    ...
    override fun onStart() {
        ...
    }

    override fun onStop() {
        super.onStop()
        crimeDetailViewModel.saveCrime(crime)
    }

    private fun updateUI() {
        ...
    }
    ...
}
```

Функция Fragment.onStop() вызывается всякий раз, когда ваш фрагмент переходит в состояние остановки (то есть всякий раз, когда фрагмент оказывается вне поля зрения). Это означает, что данные будут сохранены, когда пользователь закроет экран подробностей (например, нажав кнопку «Назад»). Данные также будут сохранены, когда пользователь переключит задачи (например, путем нажатия кнопки «Главный экран» или использования обзорного экрана). Таким образом, сохранение в onStop() отвечает требованию

сохранения данных при выходе пользователя с экрана подробностей, а также гарантирует, что в случае прекращения процесса для высвобождения памяти отредактированные данные не будут потеряны.

Запустите приложение CriminalIntent и выберите преступление из списка. Для того чтобы изменить данные этого преступления, нажмите кнопку «Назад» и похвалите себя за успех: изменения, которые вы сделали на экране подробностей, теперь отображаются на экране со списком. В следующей главе мы подключим кнопку ввода даты, чтобы пользователь мог выбрать дату, когда было совершено преступление.

Для любознательных: зачем использовать аргументы фрагментов?

В этой главе мы добавили во фрагмент функцию `newInstance(...)` для передачи аргументов при создании нового экземпляра фрагмента. Этот шаблон не только полезен с точки зрения организации кода, но и необходим в случае использования аргументов фрагмента. Нельзя использовать конструктор для передачи аргументов экземпляру фрагмента.

Например, можно рассмотреть возможность добавления в `CrimeFragment` однопараметрического конструктора, ожидающего идентификатор преступления, вместо добавления функции `newInstance(UUID)`. Однако это плохой подход. При изменении конфигурации менеджер фрагментов текущей activity автоматически заново создает фрагмент, который был размещен до изменения конфигурации. Затем менеджер фрагментов добавляет этот новый экземпляр фрагмента в новую activity.

Когда менеджер фрагментов заново создает фрагмент после изменения конфигурации, получается беспараметрический

конструктор фрагмента по умолчанию. Это означает, что после поворота новый экземпляр фрагмента не будет получать идентификатор преступления.

Так чем же отличаются аргументы фрагмента? Аргументы фрагментов сохраняются после уничтожения фрагмента. Менеджер прикрепляет аргументы к новому фрагменту, который он создает заново после поворота. Затем этот новый фрагмент может использовать пакет аргументов для воссоздания своего состояния.

Все это выглядит как-то сложно. Почему бы просто не задать переменную экземпляра в `CrimeFragment` при создании?

Потому что такое решение будет работать не всегда. Когда ОС создает заново ваш фрагмент (либо вследствие изменения конфигурации, либо при переходе пользователя к другому приложению с последующим освобождением памяти ОС), все переменные экземпляров теряются. Также помните о возможной нехватке памяти.

Если вы хотите, чтобы решение работало во всех случаях, придется сохранять аргументы.

Один из возможных вариантов — использование механизма сохранения состояния экземпляров. Идентификатор преступления заносится в обычную переменную экземпляра, сохраняется вызовом `onSaveInstanceState(Bundle)`, а затем извлекается из `Bundle` в `onCreate(Bundle?)`. Такое решение будет работать всегда.

С другой стороны, оно усложняет сопровождение. Если вы вернетесь к фрагменту через несколько лет и добавите еще один аргумент, нельзя исключать, что вы забудете сохранить его в `onSaveInstanceState(Bundle)`. Смысл этого решения не столь очевиден.

Разработчики Android предпочитают решение с аргументами фрагментов, потому что в этом случае они

предельно явно и четко обозначают свои намерения. Через несколько лет вы вернетесь к коду и будете знать, что идентификатор преступления — это аргумент, который надежно передается новым экземплярам этого фрагмента. При добавлении нового аргумента вы знаете, что его нужно сохранить в пакете аргументов.

Для любознательных: библиотека компонентов архитектуры навигации

Как вы уже видели, существует несколько способов организации навигации пользователей для вашего приложения. Вы можете завести несколько activity, по одной для каждого экрана, и запускать их по мере того, как пользователь взаимодействует с вашим пользовательским интерфейсом. Вы также можете использовать одну activity, фрагменты которой будут составлять ваш пользовательский интерфейс.

Стремясь упростить навигацию в приложениях, команда разработчиков Android создала библиотеку компонентов архитектуры навигации в составе Jetpack. Эта библиотека упрощает реализацию навигации, в ней есть графический редактор, позволяющий настроить поток навигации (рис. 12.4).

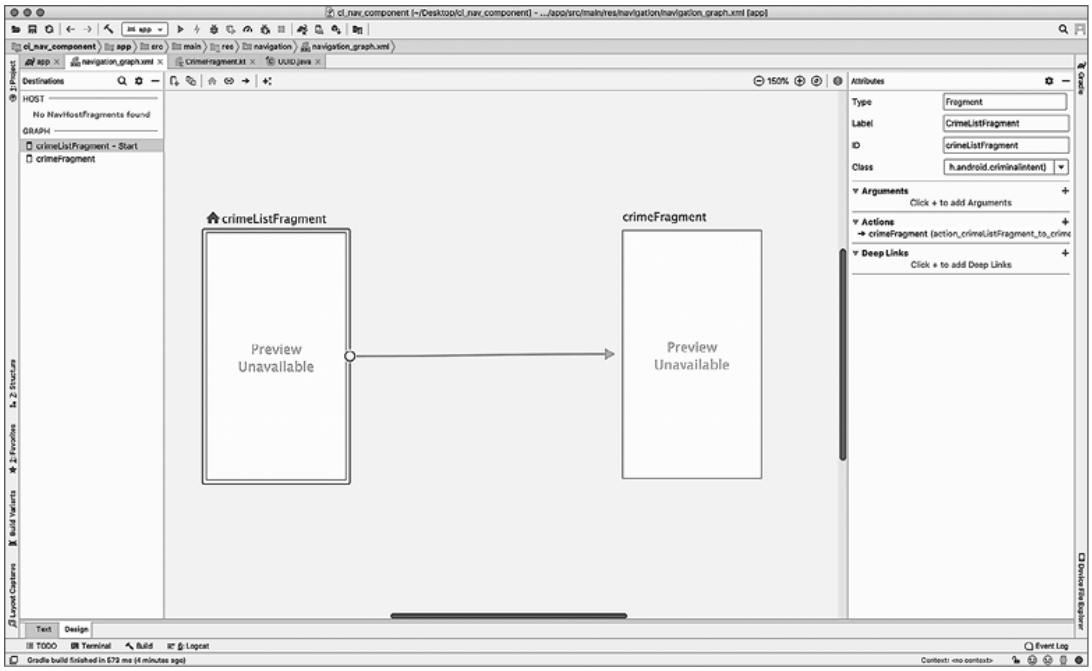


Рис. 12.4. Редактор навигации

Библиотека Navigation предвзято относится к вопросу о том, как вам следует реализовывать навигацию в вашем проекте. В ней предпочтение отдается навигации по фрагментам с одной activity. Аргументы, необходимые фрагментам, вы можете передавать средствами редактора навигации.

Чтобы усложнить задачу, создайте копию CriminalIntent и реализуйте навигацию в приложении с помощью этой библиотеки (`developer.android.com/topic/libraries/architecture/navigation`). Документация вам очень пригодится. Вам потребуется много кусочков, но после настройки будет проще простого добавлять в приложение новые экраны и передавать действия между ними.

На момент написания этой главы стабильная версия библиотеки была только что официально выпущена. Сроки не позволили нам включить ее в это издание, но этот инструмент имеет большие перспективы, и мы верим, что это путь в

будущее для многих разработчиков Android. Мы рекомендуем вам попробовать его и посмотреть, как он работает. Мы тоже сейчас это делаем.

Упражнение. Эффективная перезагрузка RecyclerView

Сейчас, когда пользователь возвращается на экран списка после редактирования преступления, `CrimeListFragment` заново выводит все видимые преступления в `RecyclerView`. Это крайне неэффективно, так как в большинстве случаев изменяется всего одно преступление.

Обновите реализацию `RecyclerView` в `CrimeListFragment`, чтобы заново выводилась только строка, связанная с измененным преступлением. Для этого обновите `CrimeAdapter`, чтобы расширить его до `androidx.recyclingerview.widget.ListAdapter<Crime, CrimeHolder>` вместо `RecyclerView.Adapter<CrimeHolder>`.

`ListAdapter` — это `RecyclerView.Adapter`, который определяет разницу между текущим и новым набором данных и который вы задаете сами. Сравнение происходит в фоновом потоке, поэтому оно не замедляет работу пользовательского интерфейса. Адаптер `ListAdapter`, в свою очередь, дает команду утилизатору перерисовывать только измененные строки.

В `ListAdapter` используется `androidx.recyclingerview.widget.DiffUtil` для определения того, какие части набора изменились. Чтобы закончить эту задачу, необходимо добавить реализацию `DiffUtil.itemCallback<Crime>` в ваш `ListAdapter`.

Вам также потребуется обновить `CrimeListFragment`, чтобы отправлять обновленный список преступлений в адаптер утилизатора, а не переназначать адаптер каждый раз, когда вы захотите обновить пользовательский интерфейс. Вы можете отправить новый список, вызвав функцию `ListAdapter.submitList(MutableList<T>?)`, или вы можете настроить `LiveData` и наблюдать за изменениями.

(См. справку по API для `androidx.recyclingerview.widget.DiffUtil` и `androidx.recyclingerview.widget.ListAdapter` на **developer.android.com/reference/kotlin** для получения более подробной информации о том, как использовать эти инструменты.)

13. Диалоговые окна

Диалоговые окна требуют от пользователя внимания и ввода данных. Обычно они используются для принятия решений или отображения важной информации. В этой главе мы добавим диалоговое окно, в котором пользователь может изменить дату преступления.

При нажатии кнопки даты в `CrimeFragment` открывается диалоговое окно (рис. 13.1).

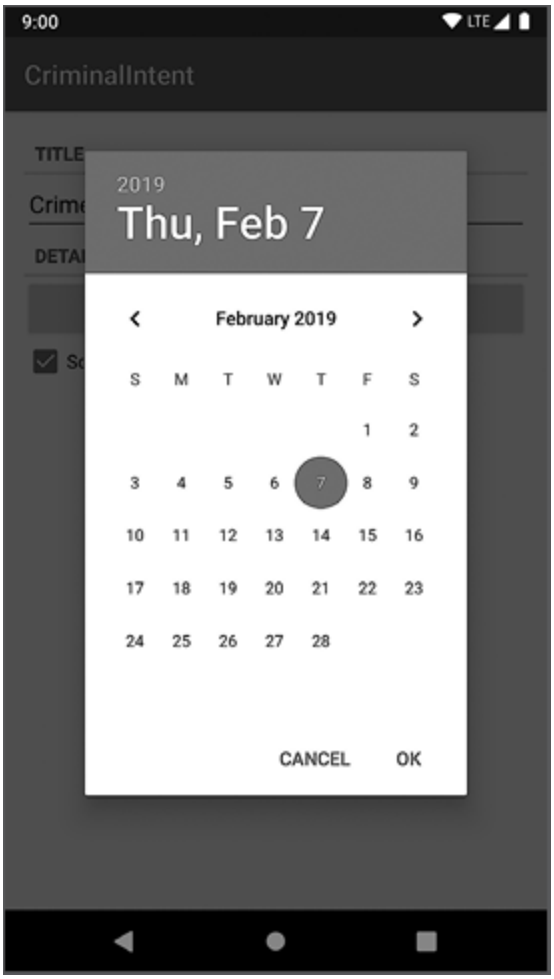


Рис. 13.1. Диалоговое окно выбора даты

Диалоговое окно, показанное на рис. 13.1, является экземпляром DatePickerDialog подкласса AlertDialog. DatePickerDialog отображает пользователю запрос на выбор даты и имеет интерфейс слушателя, который вы можете реализовать для получения результата выбора. AlertDialog является универсальным подклассом Dialog, который вы будете использовать чаще всего для создания большего количества пользовательских диалоговых окон.

Создание экземпляра DialogFragment

При использовании объекта AlertDialog обычно удобно упаковывать его в экземпляр DialogFragment подкласса Fragment. Вообще говоря, экземпляр AlertDialog может отображаться и без DialogFragment, но Android так поступать не рекомендует. Управление DatePickerDialog из FragmentManager открывает больше возможностей для его отображения.

Кроме того, «минимальный» экземпляр DatePickerDialog исчезнет при повороте устройства. С другой стороны, если экземпляр DatePickerDialog упакован во фрагмент, после поворота диалоговое окно будет создано заново и появится на экране.

Для приложения CriminalIntent мы создадим подкласс DialogFragment с именем DatePickerFragment. В коде DatePickerFragment создается и настраивается экземпляр DatePickerDialog. В качестве хоста DatePickerFragment используется экземпляр MainActivity.

На рис. 13.2 изображена схема этих отношений.

Наши первоочередные задачи:

- создание класса DatePickerFragment;
- построение DatePickerFragment;
- вывод диалогового окна на экран с использованием FragmentManager.

Ниже в этой главе мы организуем передачу необходимых данных между CrimeFragment и DatePickerFragment.

Создайте класс DatePickerFragment и назначьте DialogFragment его суперклассом. Обязательно выберите Jetpack-версию DialogFragment из библиотеки поддержки: androidx.fragment.app.DialogFragment.

Класс DialogFragment содержит следующую функцию:

```
onCreateDialog(savedInstanceState: Bundle?):
Dialog
```

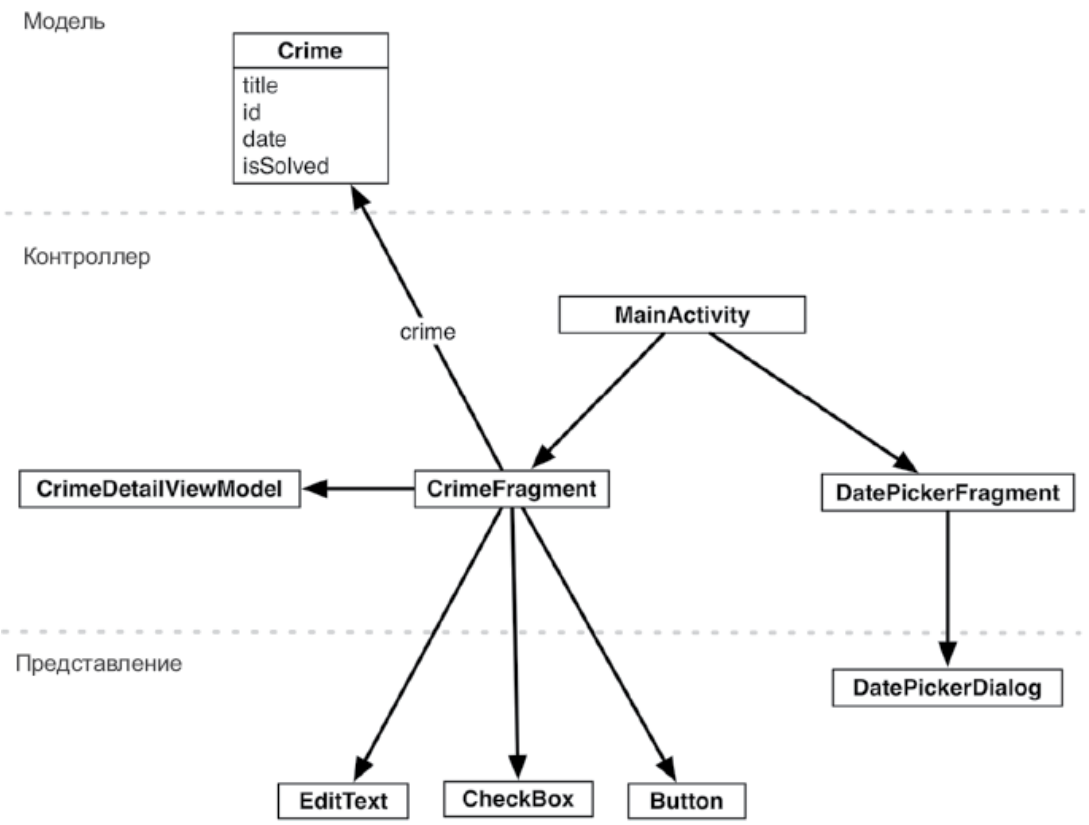


Рис. 13.2. Диаграмма объектов для двух фрагментов с хостом MainActivity

Экземпляр `FragmentManager` хост-activity вызывает эту функцию в процессе вывода `DialogFragment` на экран.

Добавьте в файл `DatePickerFragment.kt` реализацию `onCreateDialog(Bundle?)`, которая создает `DatePickerDialog`, инициализируемый с текущей датой (листинг 13.1).

Листинг 13.1. Создание DialogFragment (DatePickerFragment.kt)

```
class DatePickerFragment : DialogFragment() {

                                override fun
onCreateDialog(savedInstanceState: Bundle?):
Dialog {
        val calendar = Calendar.getInstance()
```

```
                    val initialYear =
calendar.get(Calendar.YEAR)
                    val initialMonth =
calendar.get(Calendar.MONTH)
                    val initialDay =
calendar.get(Calendar.DAY_OF_MONTH)

            return DatePickerDialog(
                requireContext(),
                null,
                initialYear,
                initialMonth,
                initialDay
            )
        }
    }
```

Конструктор `DatePickerDialog` принимает несколько параметров. Первый — это контекстный объект, который необходим для доступа к необходимым ресурсам элемента. Второй параметр — слушатель дат, который вы добавите позже в этой главе. Последние три параметра — это год, месяц и день, к которым должно быть инициализировано окно выбора даты. Пока вы не узнаете дату преступления, вы можете просто инициализировать его на текущую дату.

Отображение DialogFragment

Как и все фрагменты, экземпляры `DialogFragment` находятся под управлением экземпляра `FragmentManager` хост-activity.

Для добавления экземпляра `DialogFragment` во `FragmentManager` и вывода его на экран используются

следующие функции экземпляра фрагмента:

```
show(manager: FragmentManager, tag: String)
show(transaction: FragmentTransaction, tag: String)
```

Строковый параметр однозначно идентифицирует `DialogFragment` в списке `FragmentManager`. Выбор версии (с `FragmentManager` или `FragmentTransaction`) зависит только от вас: если передать `FragmentTransaction`, за создание и закрепление транзакции отвечаете вы. При передаче `FragmentManager` транзакция автоматически создается и закрепляется для вас.

В нашем примере передается `FragmentManager`.

Добавьте в `CrimeFragment` константу для метки `DatePickerFragment`. Затем в функции `onCreateView(...)` удалите код, блокирующий кнопку даты, и назначьте слушателя `View.OnClickListener`, который отображает `DatePickerFragment` при нажатии кнопки даты в `onStart()`.

Листинг 13.2. Отображение DialogFragment (CrimeFragment.kt)

```
private const val TAG = "CrimeFragment"
private const val ARG_CRIME_ID = "crime_id"
private const val DIALOG_DATE = "DialogDate"

class CrimeFragment : Fragment() {
    ...
        override fun onCreateView(inflater: LayoutInflater,
                                  container: ViewGroup?,
```

```
                                savedInstanceStat
e: Bundle?): View? {
        ...
                               solvedCheckBox    =
view.findViewById(R.id.crime_solved)         as
CheckBox

            ~~dateButton.apply {~~
                ~~text = crime.date.toString()~~
                ~~isEnabled = false~~
            ~~}~~

        return view
    }
    ...
    override fun onStart() {
        ...
        solvedCheckBox.apply {
            ...
        }

        dateButton.setOnClickListener {
            DatePickerFragment().apply {
                show(this@CrimeFragment.require
FragmentManager(), DIALOG_DATE)
            }
        }
    }
    ...
}
```

Фрагмент `this@CrimeFragment` необходим для вызова функции `requireFragmentManager()` из `CrimeFragment`, а не из `DatePickerFragment`. Он ссылается на `DatePickerFragment` внутри блока `apply`, поэтому необходимо указать `this` из внешней области видимости.

Функции `show(FragmentManager,String)` в `DialogFragment` требуется ненулевое значение для аргумента менеджера фрагментов. Свойство `Fragment.fragmentManager` допускает значение `null`, поэтому его нельзя передавать напрямую в `show(...)`. Вместо этого используется функция `requireFragmentManager()` в `Fragment`, возвращаемый тип которой — не null `FragmentManager`. Если при вызове функции `Fragment.requireFragmentManager()` свойство `FragmentManager` равно нулю, то функция выбросит `IllegalStateException`, утверждая, что в данный момент фрагмент не связан с менеджером.

Запустите CriminalIntent и нажмите кнопку даты, чтобы увидеть диалоговое окно (рис. 13.3).

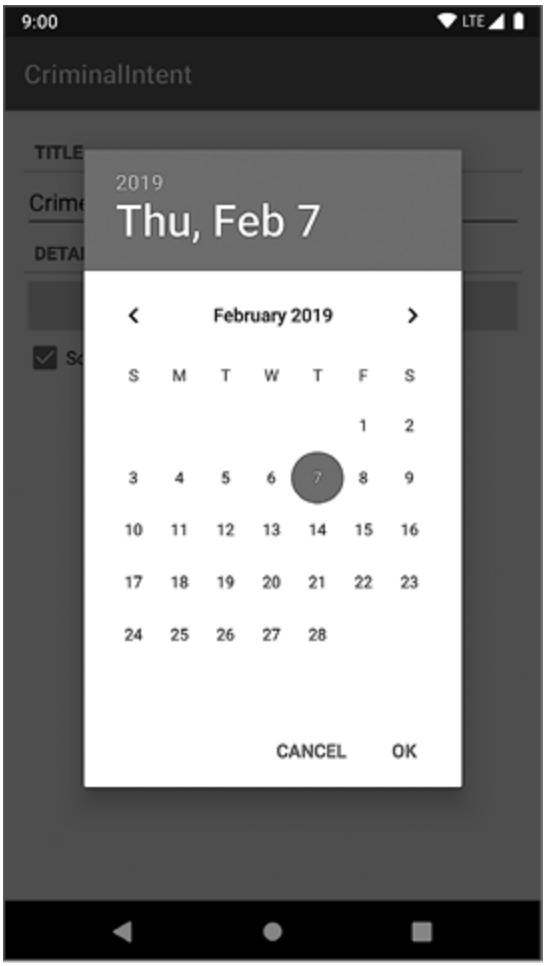


Рис. 13.3. Настроенное диалоговое окно

Диалоговое окно выглядит хорошо. В следующем разделе мы привяжем его в дате преступления и разрешим пользователю изменить его.

Передача данных между фрагментами

Мы передавали данные между двумя activity с использованием интентов, передавали данные между фрагментом и activity с помощью интерфейса обратного вызова, а также передавали данные от activity к фрагменту activity с помощью аргументов фрагмента. Теперь нужно передать данные между двумя

фрагментами, хостом которых является одна activity, — `CrimeFragment` и `DatePickerFragment` (рис. 13.4).

Рис. 13.4. Взаимодействие между CrimeFragment и DatePickerFragment

Чтобы передать дату преступления в `DatePickerFragment`, мы напишем функцию `newInstance(Date)` и сделаем объект `Date` аргументом фрагмента.

Чтобы передать новые данные обратно в `CrimeFragment` и тем самым дать ему возможность обновить слой модели и свое представление, нужно объявить функцию интерфейса обратного вызова в `DatePickerFragment`, которая принимает новый параметр даты, как показано на рис. 13.5.

Рис. 13.5. Взаимодействие между CrimeFragment и DatePickerFragment

Передача данных в DatePickerFragment

Чтобы получить данные в DatePickerFragment, мы сохраним дату в пакете аргументов DatePickerFragment, где DatePickerFragment сможет обратиться к ней.

Создание аргументов фрагмента и присваивание им значений обычно выполняется в функции newInstance(...), заменяющем конструктор фрагмента (мы говорили об этом в главе 12). Добавьте в файл DatePickerFragment.kt функцию newInstance(Date).

Листинг 13.3. Добавление функции newInstance(Date) (DatePickerFragment.kt)

```
private const val ARG_DATE = "date"

class DatePickerFragment : DialogFragment() {

                                override        fun
onCreateDialog(savedInstanceState:    Bundle?):
Dialog {
        ...
    }

    companion object {
                 fun  newInstance(date:  Date):
DatePickerFragment {
            val args = Bundle().apply {
                putSerializable(ARG_DATE, date)
            }

            return DatePickerFragment().apply {
```

```
                arguments = args
            }
        }
    }
}
```

В классе CrimeFragment удалите вызов конструктора DatePickerFragment и замените его вызовом DatePickerFragment.newInstance(Date).

Листинг 13.4. Добавление вызова newInstance(...) (CrimeFragment.kt)

```
override fun onStart() {
    ...
    dateButton.setOnClickListener {
            DatePickerFragment().apply {
        DatePickerFragment.newInstance(crime.da
te).apply {
            show(this@CrimeFragment.requireFrag
mentManager(), DIALOG_DATE)
        }
    }
}
```

Экземпляр DatePickerFragment должен инициализировать DatePickerDialog по информации, хранящейся в Date. Однако для инициализации DatePickerDialog необходимо иметь целочисленные значения месяца, дня и года. Объект Date больше напоминает временную метку и не может предоставить нужные целые значения напрямую.

Чтобы получить нужные значения, следует создать объект Calendar и использовать Date для определения его конфигурации. После этого вы сможете получить нужную информацию из Calendar.

В функции onCreateDialog(Bundle?) получите объект Date из аргументов и используйте его с Calendar для инициализации DatePicker.

Листинг 13.5. Извлечение даты и инициализация DatePickerDialog (DatePickerFragment.kt)

```
class DatePickerFragment : DialogFragment() {

                                override        fun
onCreateDialog(savedInstanceState:    Bundle?):
Dialog {
                                    val    date    =
arguments?.getSerializable(ARG_DATE) as Date
        val calendar = Calendar.getInstance()
        calendar.time = date
                             val    initialYear    =
calendar.get(Calendar.YEAR)
                            val    initialMonth    =
calendar.get(Calendar.MONTH)
                             val    initialDate    =
calendar.get(Calendar.DAY_OF_MONTH)

        return DatePickerDialog(
            requireContext(),
            null,
            initialYear,
            initialMonth,
```

```
                initialDate
            )
        }
        ...
    }
```

Теперь `CrimeFragment` успешно сообщает `DatePickerFragment`, какую дату следует отобразить. Вы можете запустить приложение CriminalIntent и убедиться в том, что все работает так же, как прежде.

Возвращение данных CrimeFragment

Чтобы экземпляр `CrimeFragment` получал данные от `DatePickerFragment`, нам необходимо каким-то образом отслеживать отношения между двумя фрагментами.

С activity вы вызываете `startActivityForResult(...)`, а `ActivityManager` отслеживает отношения между родительской и дочерней activity. Когда дочерняя activity прекращает существование, `ActivityManager` знает, какая activity должна получить результат.

Назначение целевого фрагмента

Для создания аналогичной связи можно назначить `CrimeFragment`*целевым фрагментом* (*target fragment*) для `DatePickerFragment`. Эта связь будет автоматически восстановлена после того, как и `CrimeFragment`, и `DatePickerFragment` будут уничтожены и заново созданы ОС. Для этого вызывается следующая функция `Fragment`:

```
setTargetFragment(fragment: Fragment,
requestCode: Int)
```

Функция получает фрагмент, который станет целевым, и код запроса, аналогичный передаваемому `startActivityForResult(...)`. По коду запроса целевой фрагмент позднее может определить, какой фрагмент возвращает информацию.

`FragmentManager` сохраняет целевой фрагмент и код запроса. Чтобы получить их, обратитесь к свойствам `targetFragment` и `targetRequestCode` для фрагмента, назначившего целевой фрагмент.

В файле `CrimeFragment.kt` создайте константу для кода запроса, а затем назначьте `CrimeFragment` целевым фрагментом экземпляра `DatePickerFragment`.

Листинг 13.6. Назначение целевого фрагмента (CrimeFragment.kt)

```
private const val DIALOG_DATE = "DialogDate"
private const val REQUEST_DATE = 0

class CrimeFragment : Fragment() {
    ...
    override fun onStart() {
        ...
        dateButton.setOnClickListener {
            DatePickerFragment.newInstance(crim
e.date).apply {
                setTargetFragment(this@CrimeFra
gment, REQUEST_DATE)
                show(this@CrimeFragment.require
FragmentManager(), DIALOG_DATE)
            }
        }
```

```
        }
        ...
    }
```

Передача данных целевому фрагменту

Теперь, когда у вас есть связь между `CrimeFragment` и `DatePickerFragment`, вам нужно отправить дату обратно в `CrimeFragment`. В `DatePickerFragment` вы создадите интерфейс обратного вызова, который будет реализован классом `CrimeFragment`.

В `DatePickerFragment` создайте интерфейс обратного вызова с единственной вызываемой функцией `onDateSelected()`.

Листинг 13.7. Создание интерфейса обратного вызова (DatePickerFragment.kt)

```
class DatePickerFragment : DialogFragment() {

    interface Callbacks {
        fun onDateSelected(date: Date)
    }

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        ...
    }
    ...
}
```

Теперь внедрите интерфейс обратных вызовов в CrimeFragment. В функции onDateSelected() установите дату в свойстве Crime и обновите пользовательский интерфейс.

Листинг 13.8. Реализация интерфейса обратных вызовов (CrimeFragment.kt)

```
class CrimeFragment : Fragment(),
DatePickerFragment.Callbacks {
    ...
    override fun onStop() {
        ...
    }

    override fun onDateSelected(date: Date) {
        crime.date = date
        updateUI()
    }
    ...
}
```

Теперь, когда CrimeFragment отвечает на новые даты, DatePickerFragment должен отправлять новую дату, когда пользователь выбирает ее. В DatePickerFragment добавьте слушателя в DatePickerDialog, который будет отправлять дату обратно в CrimeFragment (листинг 13.9).

Листинг 13.9. Возврат даты (DatePickerFragment.kt)

```
class DatePickerFragment : DialogFragment() {
    ...
```

```
                                override fun
onCreateDialog(savedInstanceState:    Bundle?):
Dialog {
                            val   dateListener   =
DatePickerDialog.OnDateSetListener {
                        _: DatePicker, year: Int,
month: Int, day: Int ->

                        val resultDate : Date =
GregorianCalendar(year, month, day).time

            targetFragment?.let { fragment ->
                                    (fragment as
Callbacks).onDateSelected(resultDate)
            }
        }

                                    val    date    =
arguments?.getSerializable(ARG_DATE) as Date
        ...
        return DatePickerDialog(
            requireContext(),
               null,
            dateListener,
            initialYear,
            initialMonth,
            initialDate
        )
    }
    ...
}
```

`OnDateSetListener` используется для получения выбранной пользователем даты. Первый параметр –это `DatePicker`, от которого исходит результат. Поскольку в данном случае этот параметр не используется, он имеет имя «_». Это соглашение по именованию в Kotlin для параметров, которые не используются.

Выбранная дата предоставляется в формате года, месяца и дня, но для отправки обратно в `CrimeFragment` необходим объект `Date`. Вы передаете эти значения в `GregorianCalendar` и получаете доступ к свойству `time`, чтобы получить объект `Date`.

Полученная дата должна передаваться обратно в `CrimeFragment`. В свойстве `targetFragment` хранится экземпляр фрагмента, который запустил ваш `DatePickerFragment`. Так как в нем значение `null`, его нужно обернуть в безопасный вызов `let`. Затем экземпляр фрагмента передается в интерфейс `Callbacks` и вызывается функция `onDateSelected()`, передающая новую дату.

Теперь круг замкнулся. Даты должны меняться. Тот, кто контролирует даты, контролирует само время. Запустите CriminalIntent, чтобы убедиться, что вы можете контролировать даты. Измените дату преступления и проверьте результат в представлении `CrimeFragment`. Затем вернитесь к списку преступлений и проверьте дату, чтобы убедиться, что слой модели был обновлен.

Упражнение. Новые диалоговые окна

Напишите еще один диалоговый фрагмент `TimePickerFragment` для выбора времени преступления.

Используйте виджет `TimePicker`, добавьте в `CrimeFragment` еще одну кнопку для отображения `TimePickerFragment`.

14. Панель приложения

Панель приложения (*app bar*) — ключевой компонент любого хорошо спроектированного Android-приложения. Панель приложения содержит действия, которые могут выполняться пользователем, и новые средства навигации, а также обеспечивает единство дизайна и фирменного стиля.

В этой главе мы создадим для приложения CriminalIntent меню для добавления нового преступления (рис. 14.1).

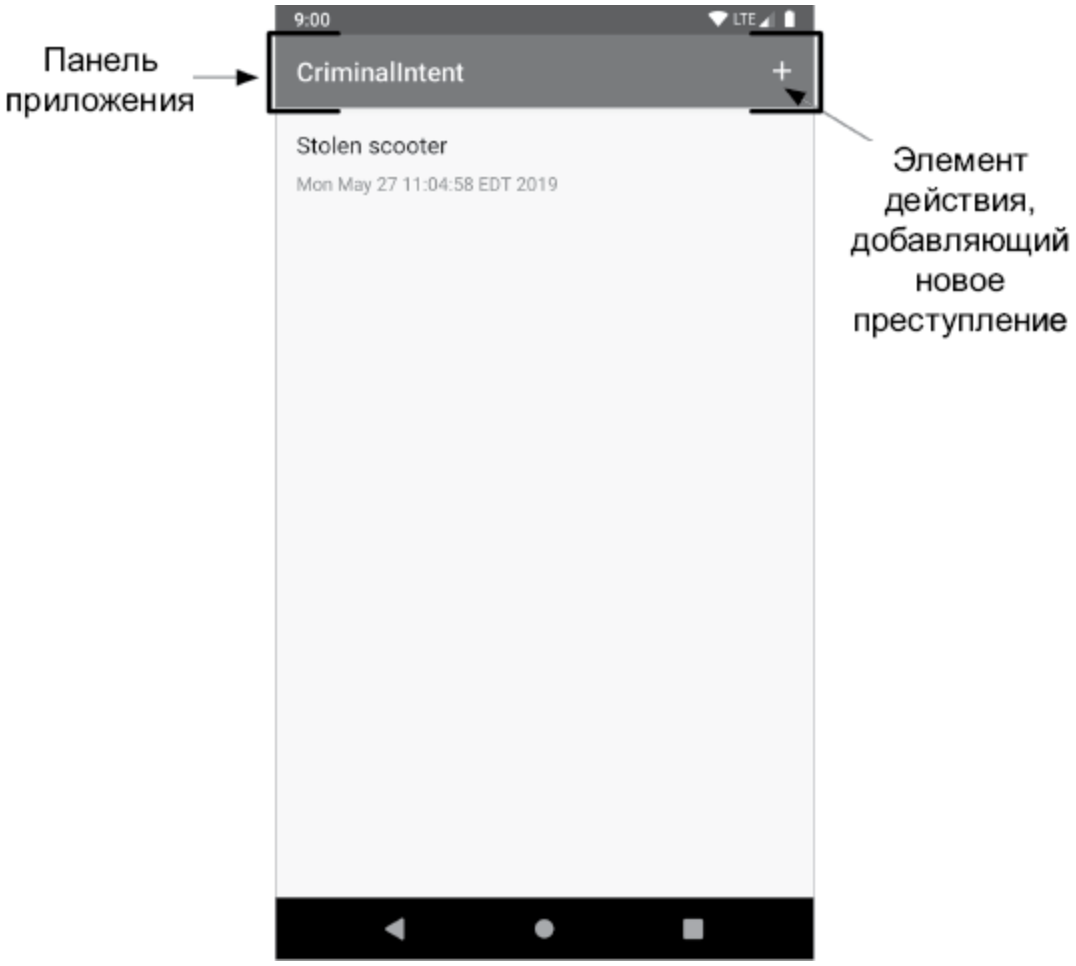


Рис. 14.1. Панель приложения CriminalIntent

Панель приложения часто называют *панелью действий* или *панелью инструментов*. Подробнее об этих схожих терминах вы узнаете в разделе «Для любознательных: панель приложения/действий/инструментов» в конце этой главы.

Панель приложения AppCompat

У приложения CriminalIntent уже есть простая панель приложения.

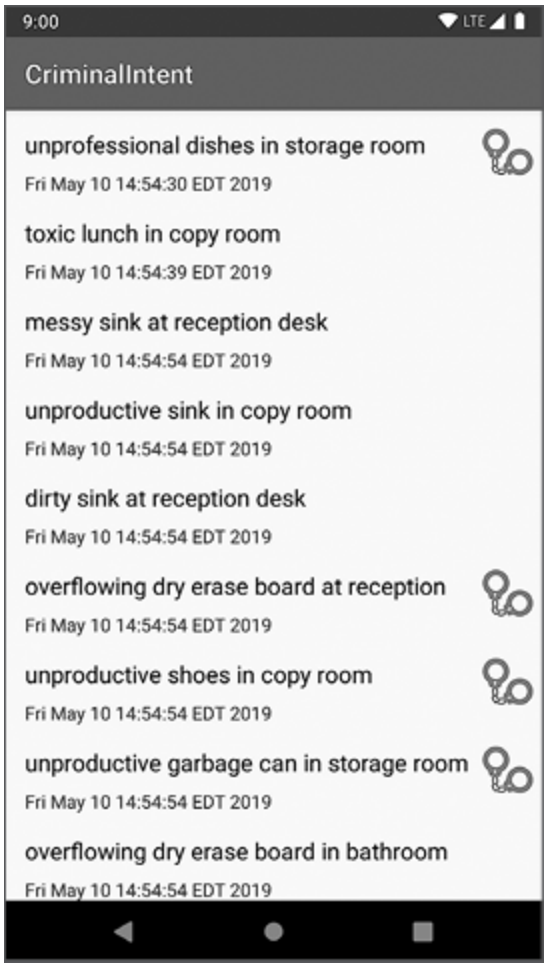


Рис. 14.2. Панель приложения

Это потому, что Android Studio настраивает все новые проекты на использование панели приложения по умолчанию

для всех видов activity, которые наследуются от `AppCompatActivity`. Это делается путем:

- добавления зависимости от базовой Jetpack-библиотеки `AppCompat`;
- применения одной из тем AppCompat, куда входит панель приложений.

Откройте файл `app/build.gradle`, чтобы увидеть зависимость от AppCompat.

```
dependencies {
    ...
                                    implementation
'androidx.appcompat:appcompat:1.0.0-beta01'
    ...
```

AppCompat — сокращение от словосочетания «совместимость приложений». В базовой Jetpack-библиотеке AppCompat содержатся классы и ресурсы, которые являются ядром для создания правильного пользовательского интерфейса для различных версий Android. Вы можете изучить содержимое подпакетов AppCompat в официальных списках API на сайте **developer.android.com/reference/kotlin/androidx/packages**.

Android Studio при создании проекта автоматически устанавливает *тему* приложения `Theme.AppCompat.Light.DarkActionBar`. Тема, которая задает стиль по умолчанию для всего вашего приложения, определена в файле `res/values/styles.xml`:

```
<resources>
```

```
    <!-- Base application theme. -->
                        <style     name="AppTheme"
parent="    Theme.AppCompat.Light.DarkActionBar
">
        <!-- Customize your theme here. -->
                                                <item
name="colorPrimary">@color/colorPrimary</item>
                                                <item
name="colorPrimaryDark">@color/colorPrimaryDark
</item>
                                                <item
name="colorAccent">@color/colorAccent</item>
    </style>

</resources>
```

Тема приложения указывается на уровне приложения и может задаваться отдельно для каждой activity в манифесте. Откройте файл `manifests/AndroidManifest.xml` и посмотрите на тег `<application>`. Обратите внимание на атрибут `android:theme`. Вы должны увидеть что-то подобное

```
<manifest ... >
    <application
        ...
            android:theme="@style/AppTheme" >
        ...
    </application>
</manifest>
```

Вы узнаете больше о стилях и темах в главе 21. Теперь пришло время добавить функций панели приложения.

Меню

Правая верхняя часть панели приложения зарезервирована для меню. Меню состоит из элементов действий (иногда также называемых *элементами меню*), выполняющих действия на текущем экране или в приложении в целом. Мы добавим элемент действия, при помощи которого пользователь сможет создать описание нового преступления.

Для работы нового действия потребуется строковый ресурс для метки. Добавьте метку в файл `strings.xml` (листинг 14.1).

Листинг 14.1. Добавление строк для меню (res/values/strings.xml)

```
<resources>
    ...
                                                <string
name="crime_solved_label">Solved</string>
    <string name="new_crime">New Crime</string>

</resources>
```

Определение меню в XML

Меню определяются такими же ресурсами, как и макеты. Вы создаете описание меню в формате XML и помещаете файл в каталог `res/menu` своего проекта. Android генерирует идентификатор ресурса для файла меню, который затем используется для заполнения меню в коде.

На панели **Project** щелкните правой кнопкой мыши по каталогу `res` и выберите команду **New⇒Androidresourcefile** в контекстном меню. Выберите тип ресурса **Menu**, присвойте

ресурсу меню имя **fragment_crime_list** и нажмите кнопку **OK** (рис. 14.3).

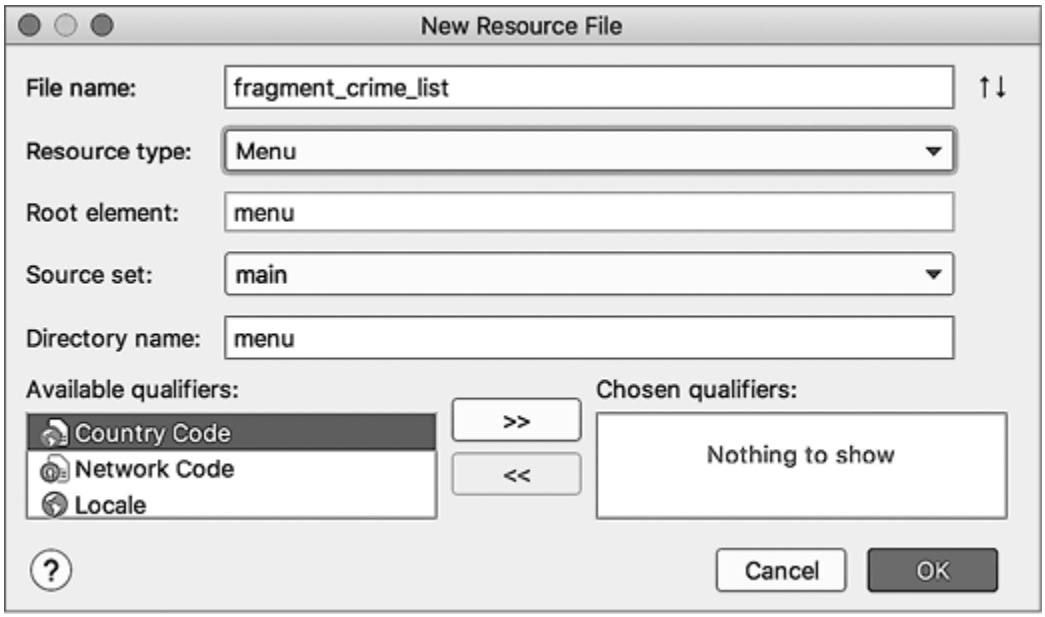


Рис. 14.3. Создание файла меню

Здесь для файлов меню используется та же схема формирования имен, что и для файлов макетов. Android Studio генерирует файл `res/menu/fragment_crime_list.xml`: его имя совпадает с именем файла макета `CrimeListFragment`, но файл находится в папке `menu`. В новом файле переключитесь в режим **Text** и добавьте элемент `item` (листинг 14.2).

Листинг 14.2. Создание ресурса меню CrimeListFragment (res/menu/fragment_crime_list.xml)

```
<menu
xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
```

```
            android:id="@+id/new_crime"
            android:icon="@android:drawable/ic_menu
_add"
            android:title="@string/new_crime"
            app:showAsAction="ifRoom|withText"/>
</menu>
```

Атрибут `showAsAction` определяет, должна ли команда меню отображаться на самой панели приложения или в *дополнительном меню* (*overflow menu*). Мы объединили два значения, `ifRoom` и `withText`, чтобы при наличии свободного места на панели приложения отображался значок и текст команды. Если на панели не хватает места для текста, то отображается только значок. Если места нет ни для того ни для другого, команда перемещается в дополнительное меню.

Дополнительное меню вызывается значком в виде трех точек в правой части панели приложения (рис. 14.4).

Рис. 14.4. Дополнительное меню на панели приложения

Также атрибут `showAsAction` может принимать значения `always` и `never`. Выбирать `always` не рекомендуется; лучше использовать `ifRoom` и предоставить решение ОС. Вариант `never` хорошо подходит для редко выполняемых действий. Как правило, на панели приложения следует размещать только часто используемые команды меню, чтобы не загромождать экран.

Пространство имен app

Обратите внимание, что в файле `fragment_crime_list.xml` тег `xmlns` используется для определения нового пространства имен `app` отдельно от обычного объявления пространства имен `android`. Затем пространство имен `app` используется для назначения атрибута `showAsAction`.

Это необычное объявление пространства имен существует для обеспечения совместимости с библиотекой AppCompat. API панели приложения впервые появились в Android 3.0. Изначально библиотека AppCompat была создана для включения совместимой версии панели приложения, поддерживающей более ранние версии Android, чтобы панель приложения могла использоваться на любых устройствах, даже на не поддерживающих встроенную панель приложениях. На устройствах с Android 2.3 и более ранними версиями меню и соответствующая разметка XML не существовали, атрибут `android:showAsAction` добавился только с выпуском панели приложения.

Библиотека AppCompat определяет собственный атрибут `showAsAction`, игнорируя системную реализацию `showAsAction`.

Создание меню

Для управления меню в коде используются функции обратного вызова класса `Activity`. Когда возникает необходимость в меню, Android вызывает функцию `Activity` с именем `onCreateOptionsMenu(Menu)`.

Однако архитектура нашего приложения требует, чтобы реализация находилась во фрагменте, а не в activity. Класс `Fragment` содержит собственный набор функций обратного вызова для меню, которые мы реализуем в

CrimeListFragment. Для создания меню и обработки выбранных команд используются следующие функции:

```
onCreateOptionsMenu(menu: Menu, inflater: MenuInflater)
onOptionsItemSelected(item: MenuItem): Boolean
```

В файле CrimeListFragment.kt переопределите функцию onCreateOptionsMenu(Menu,MenuInflater) так, чтобы она заполняла меню, определенное в файле fragment_crime_list.xml.

Листинг 14.3. Заполнение ресурса меню (CrimeListFragment.kt)

```
class CrimeListFragment : Fragment() {
    ...
    override fun onDetach() {
        super.onDetach()
        callbacks = null
    }

    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        super.onCreateOptionsMenu(menu, inflater)
        inflater.inflate(R.menu.fragment_crime_list, menu)
    }
    ...
}
```

Мы вызываем функцию MenuInflater.inflate(int,Menu) и передаем

идентификатор ресурса своего файла меню. Вызов заполняет экземпляр Menu командами, определенными в файле.

Обратите внимание на вызов реализации onCreateOptionsMenu(...) суперкласса. Он не обязателен, но мы рекомендуем вызывать версию суперкласса просто для соблюдения общепринятой схемы, чтобы работала вся функциональность меню, определяемая в суперклассе. Впрочем, в данном случае это лишь формальность — базовая реализация этой функции из Fragment не делает ничего.

FragmentManager отвечает за вызов Fragment.onCreateOptionsMenu(Menu,MenuInflater) при получении activity обратного вызова onCreateOptionsMenu(...) от ОС. Вы должны явно указать FragmentManager, что фрагмент должен получить вызов onCreateOptionsMenu(...). Для этого вызывается следующая функция Fragment:

```
setHasOptionsMenu(hasMenu: Boolean)
```

В функции CrimeListFragment.onCreate(Bundle?) сообщите FragmentManager, что экземпляр CrimeListFragment должен получать обратные вызовы меню.

Листинг 14.4. Получение обратных вызовов (CrimeListFragment.kt)

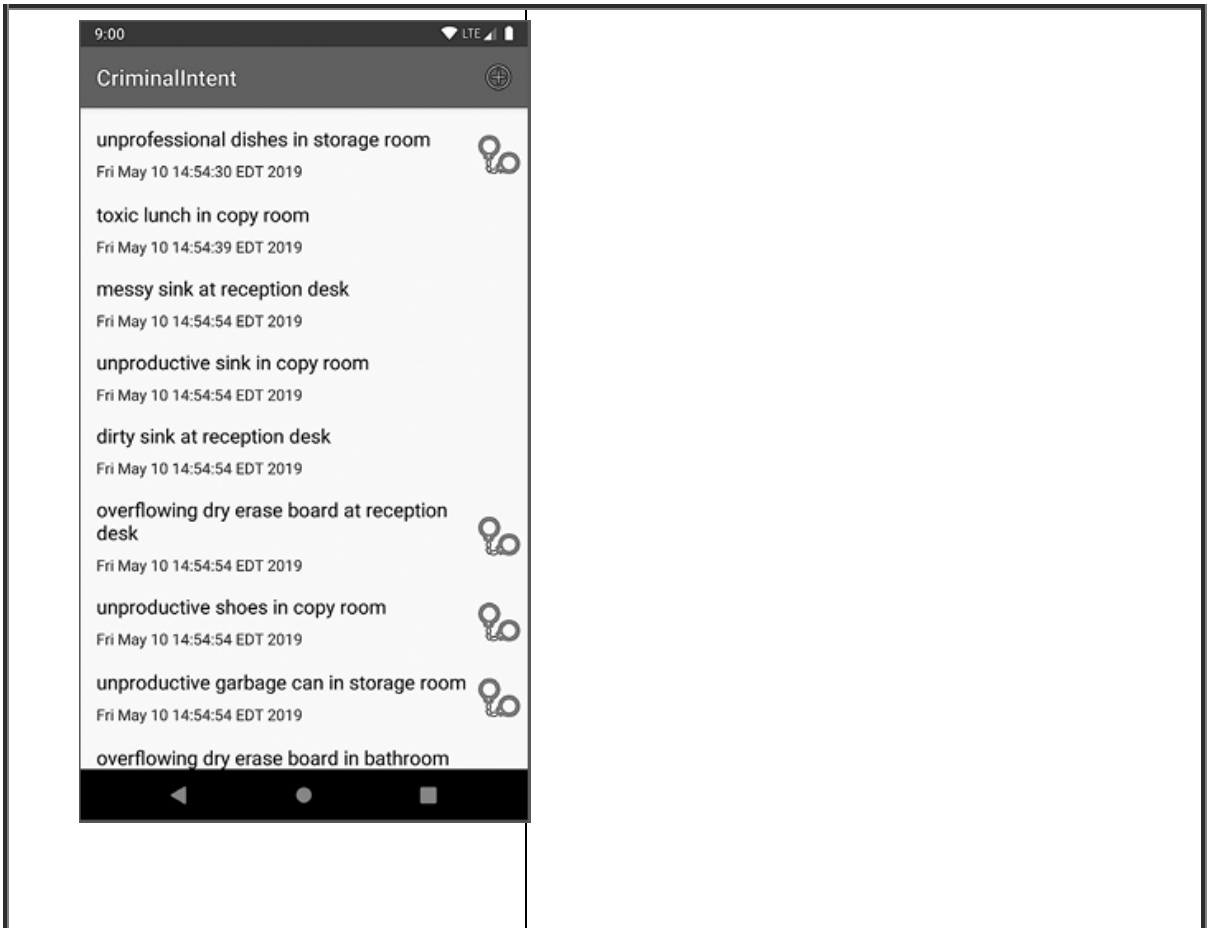
```
class CrimeListFragment : Fragment() {
    ...
    override fun onAttach(context: Context) {
        ...
    }
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setHasOptionsMenu(true)
    }
    ...
}
```

В приложении CriminalIntent появится меню (рис. 14.5).

Где текст элемента меню? У большинства телефонов в книжной ориентации хватает места только для значка. Текст команды открывается удерживанием значка на панели приложения (рис. 14.6).

В альбомной ориентации на панели приложения хватает места как для значка, так и для текста (рис. 14.7).



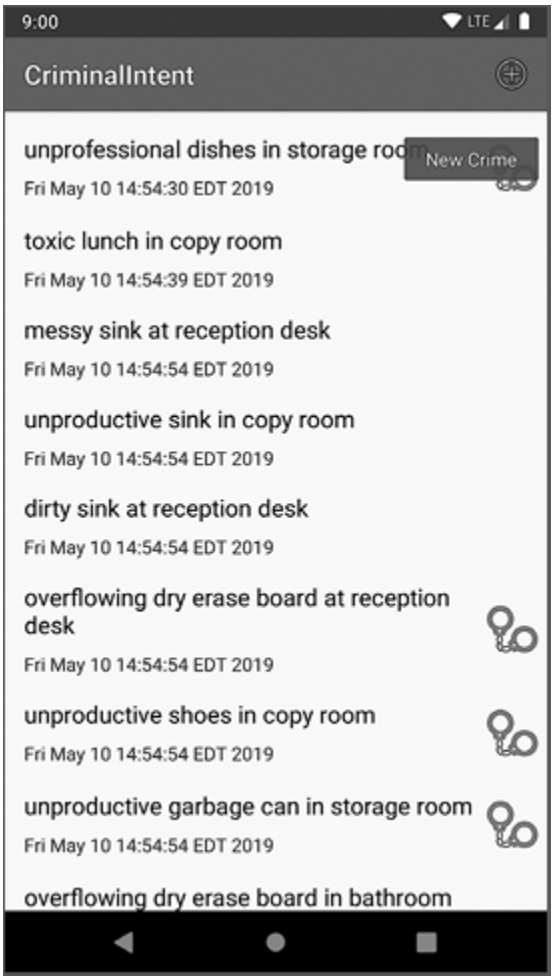


Рис. 14.5. Значок дествия New Crime на панели приложения

Рис. 14.6. Удерживание значка на панели приложения выводит текст команды

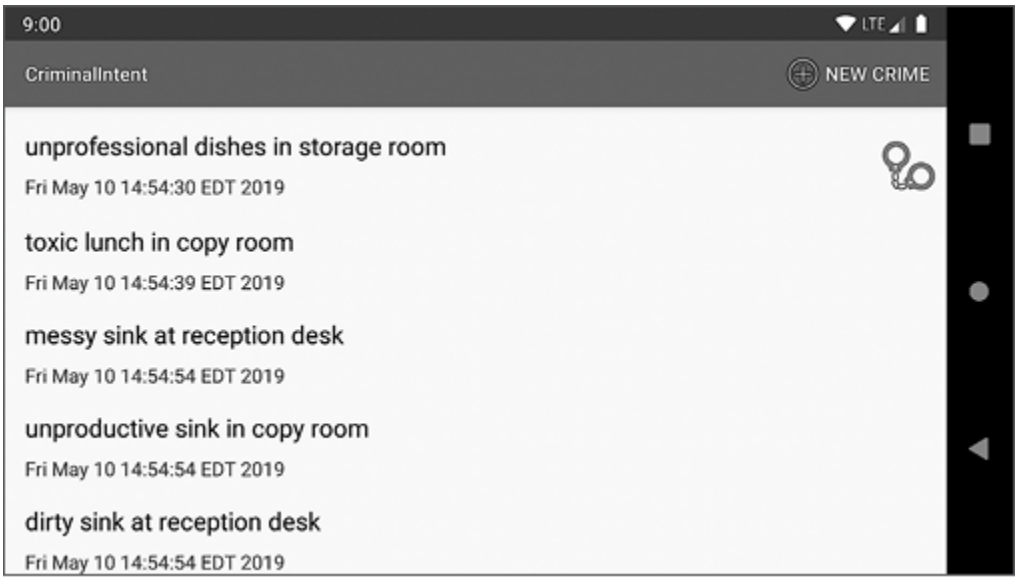


Рис. 14.7. Значок и текст на панели приложения в альбомной ориентации

Реакция на выбор команд

Чтобы ответить пользователю, запросившему действие **NewCrime**, вам понадобится способ для `CrimeListFragment` добавить новое преступление в базу данных. Нужно добавить функцию в `CrimeListViewModel` для обертывания вызова функции `addCrime(Crime)` репозитория.

Листинг 14.5. Добавление нового объекта Crime (CrimeListViewModel.kt)

```
class CrimeListViewModel : ViewModel() {

    private val crimeRepository = CrimeRepository.get()
    val crimeListLiveData = crimeRepository.getCrimes()

    fun addCrime(crime: Crime) {
```

```
            crimeRepository.addCrime(crime)
        }
    }
```

Когда пользователь выбирает команду в меню, фрагмент получает обратный вызов функции `onOptionsItemSelected(MenuItem)`. Эта функция получает экземпляр `MenuItem`, описывающий выбор пользователя.

И хотя наше меню состоит всего из одной команды, в реальных меню их обычно больше. Чтобы определить, какая команда меню была выбрана, проверьте идентификатор `MenuItem` и отреагируйте соответствующим образом. Этот идентификатор соответствует идентификатору, назначенному `MenuItem` в файле меню.

В файле `CrimeListFragment.kt` реализуйте функцию `onOptionsItemSelected(MenuItem)`, реагирующую на выбор команды меню. Реализация создает новый объект `Crime`, добавляет его в базу данных и затем уведомляет родительскую activity о том, что запрошено добавление нового преступления.

Листинг 14.6. Реакция на выбор команды меню (CrimeListFragment.kt)

```
class CrimeListFragment : Fragment() {
    ...
        override fun onCreateOptionsMenu(menu:
Menu, inflater: MenuInflater) {
                super.onCreateOptionsMenu(menu,
inflater)
            inflater.inflate(R.menu.fragment_crime_
list, menu)
    }
```

```
        override fun onOptionsItemSelected(item:
    MenuItem): Boolean {
            return when (item.itemId) {
                R.id.new_crime -> {
                    val crime = Crime()
                    crimeListViewModel.addCrime(cri
    me)
                    callbacks?.onCrimeSelected(crim
    e.id)
                    true
                }
                                    else -> return
    super.onOptionsItemSelected(item)
            }
        }
        ...
    }
```

Обратите внимание, что эта функция возвращает значение логического типа. После того как обработали `MenuItem`, вы должны вернуть `true`, чтобы указать, что дальнейшая обработка не требуется. Если вернете `false`, обработка меню будет продолжена вызовом функции `onOptionsItemSelected(MenuItem)` из хост-activity (или, если activity содержит другие фрагменты, на этих фрагментах будет вызвана функция `onOptionsItemSelected`). По умолчанию вызывается реализация суперкласса, если в вашей реализации идентификатор элемента отсутствует.

Теперь вы можете добавлять преступления самостоятельно, поэтому случайная база данных, которую вы загрузили, больше не нужна. Удалите файлы базы данных, чтобы начать заново с

пустого списка преступлений: откройте панель **DeviceFileExplorer**, откройте папку `data/data`. Найдите и разверните папку с именем вашего пакета. Щелкните правой кнопкой мыши по папке `databases` и выберите команду **Delete** в контекстном меню. После этого папка базы данных исчезнет (рис. 14.8).

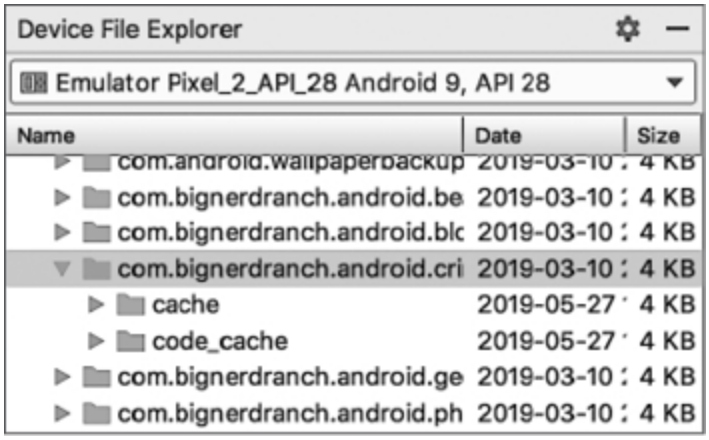


Рис. 14.8. Файлы базы данных удалены

Обратите внимание, что каталоги, оставшиеся в папке `data/data/имя.вашего.пакета`, могут немного отличаться от тех, что показаны на рис. 14.8. Это нормально, так как папка базы данных пока отсутствует.

Запустите приложение CriminalIntent. Сначала вы увидите пустой список. Создайте новый пункт меню, позволяющий добавить новое преступление. В списке новых преступлений вы должны его увидеть (рис. 14.9).

Пустой список может сбить вас с толку. Если вы решите упражнение в конце этой главы, то получите подсказку по этой проблеме.

Использование Android Asset Studio

В атрибуте `android:icon` значение `@android:drawable/ic_menu_add` ссылается на *системный значок* (*system icon*). Системные значки находятся на устройстве, а не в ресурсах проекта.

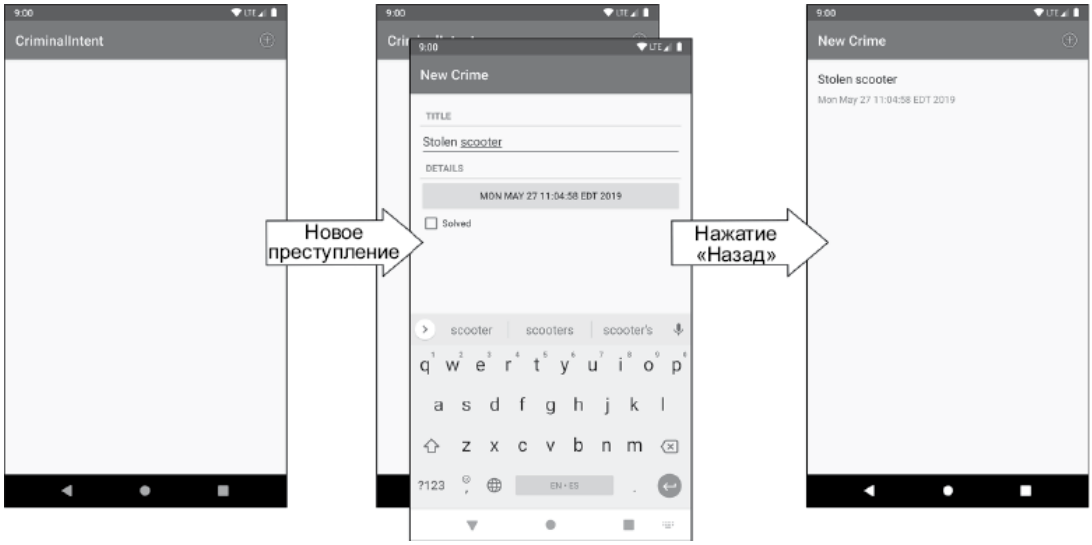


Рис. 14.9. Поток новых преступлений

В прототипе приложения ссылки на системные значки работают нормально. Однако в приложении, готовом к выпуску, лучше быть уверенным в том, чтˆо именно пользователь увидит на экране. Системные значки могут сильно различаться между устройствами и версиями ОС, а на некоторых устройствах системные значки могут не соответствовать дизайну приложения.

Одно из возможных решений — создание собственных значков. Вам придется подготовить версии для каждого разрешения экрана, а возможно, и для других конфигураций устройств. За дополнительной информацией обращайтесь к руководству «Значки приложения» по адресу **developer.android.com/design/style/iconography.html**.

Также можно действовать иначе: найти системные значки, соответствующие потребностям вашего приложения, и

скопировать их прямо в графические ресурсы проекта.

Системные значки находятся в каталоге Android SDK. В операционной системе macOS это обычно каталог вида `/Users/пользователь/Library/Android/sdk`. В Windows по умолчанию используется путь `\Users\пользователь\sdk`. Также для определения местонахождения SDK можно открыть панель **ProjectStructure** (**File⇒ProjectStructure**) и выбрать категорию `AndroidSDKlocation`.

В каталоге SDK вы найдете разные ресурсы Android, включая `ic_menu_add`. Эти ресурсы находятся в каталоге `/platforms/android-XX/data/res`, где XX — уровень API Android-версии. К примеру, если используется API уровня 28, вы найдете ресурсы Android в каталоге `platforms/android-28/data/res`.

Третий и самый простой вариант — использовать программу Android Asset Studio, включенную в Android Studio. Asset Studio позволяет создать и настроить изображение для использования на панели приложения.

Щелкните правой кнопкой мыши по каталогу `drawable` на панели **Project** и выберите команду **New⇒ImageAsset** в контекстном меню. На экране появится Asset Studio (рис. 14.10).

Вы можете сгенерировать несколько типов значков. Попробуйте создать новый значок для действия **NewCrime**. В раскрывающемся списке **IconType** выберите пункт **ActionBarandTabIcons**. Теперь присвойте активу имя **ic_menu_add** и установите переключатель **AssetType** в положение **ClipArt**.

Обновите тему, чтобы использовать **HOLO_DARK**. Так как ваша панель приложения оформлена темной темой, ваше изображение должно быть светлым. Результат показан на рис. 14.10. В примере отображается фрагмент выбранного

изображения, а на экране появится очаровательный логотип Android.

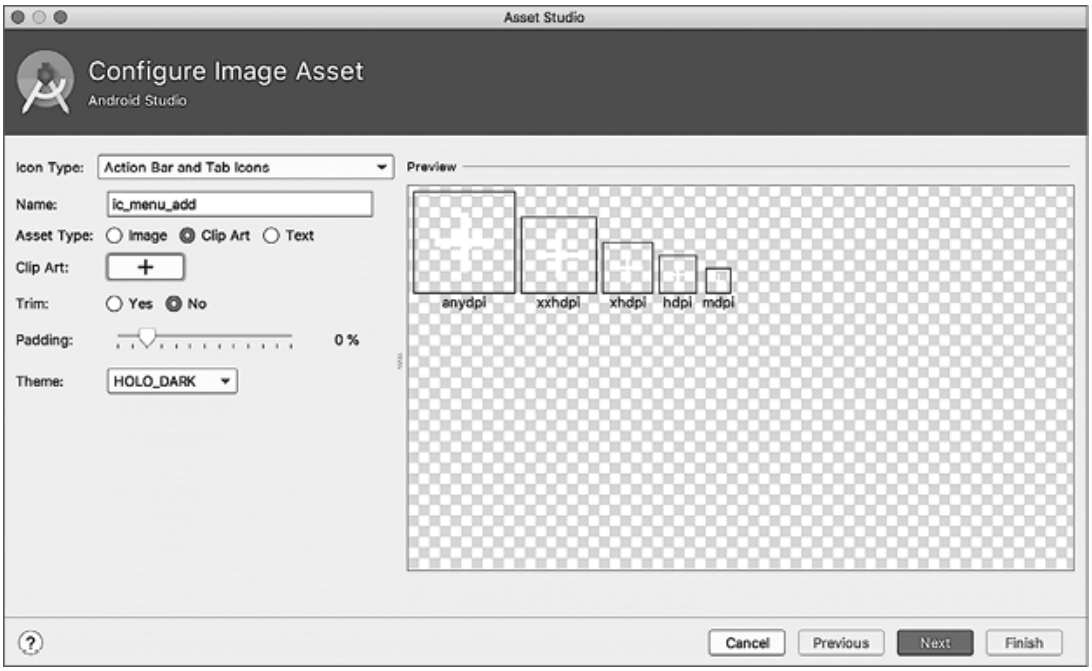


Рис. 14.10. Asset Studio

Нажмите кнопку **ClipArt**, чтобы выбрать графическую заготовку. В открывшемся окне выберите изображение, напоминающее знак + (рис. 14.11). (Вы можете ввести слово «add» в поле поиска слева вверху, чтобы облегчить поиск.)

Рис. 14.11. Галерея графических заготовок с символом +

На основном экране нажмите кнопку **Next**, чтобы перейти к последнему шагу мастера. Asset Studio также выводит план работы, которую выполнит Asset Studio (рис. 14.12). Обратите внимание: значки mdpi, hdpi, xhdpi и xxhdpi будут созданы автоматически. Класс!

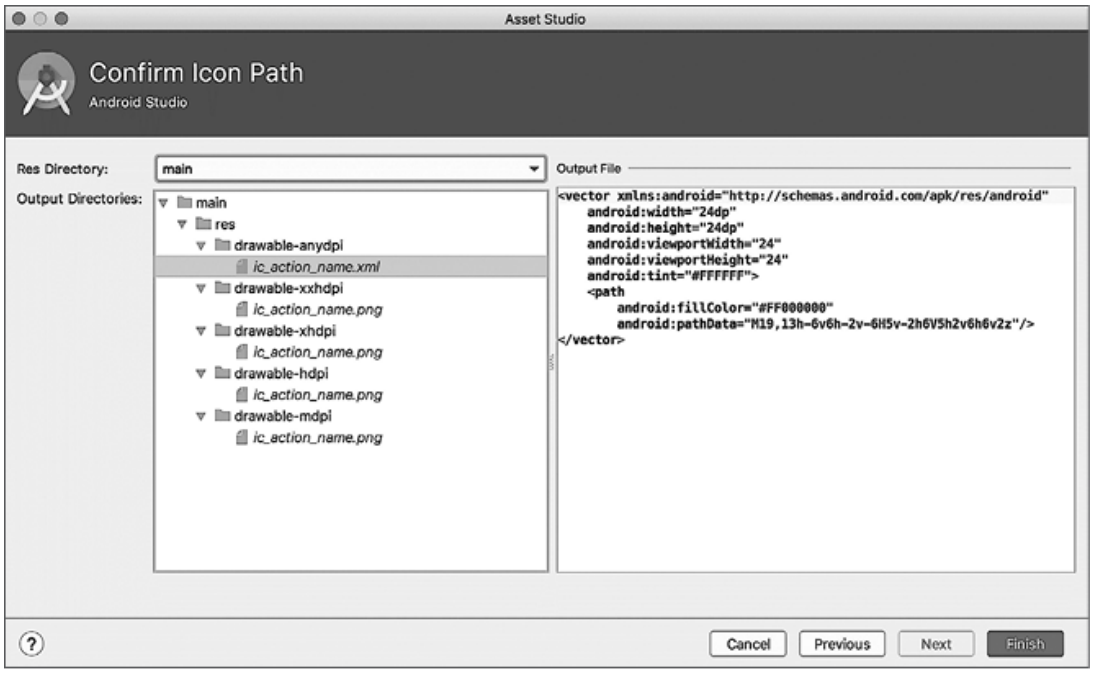


Рис. 14.12. Файлы, сгенерированные Asset Studio

Нажмите кнопку **Finish**, чтобы сгенерировать изображения. Затем в файле макета измените атрибут `icon` и включите в него ссылку на новый ресурс из вашего проекта.

Листинг 14.7. Ссылка на локальный ресурс (res/menu/fragment_crime_list.xml)

```
<item
    android:id="@+id/new_crime"
        ~~android:icon="@android:drawable/ic_menu_add"~~
    android:icon="@drawable/ic_menu_add"
```

```
        android:title="@string/new_crime"
        app:showAsAction="ifRoom|withText"/>
```

Запустите приложение и полюбуйтесь новеньким значком, который выглядит одинаково в любых версиях Android (рис. 14.13).

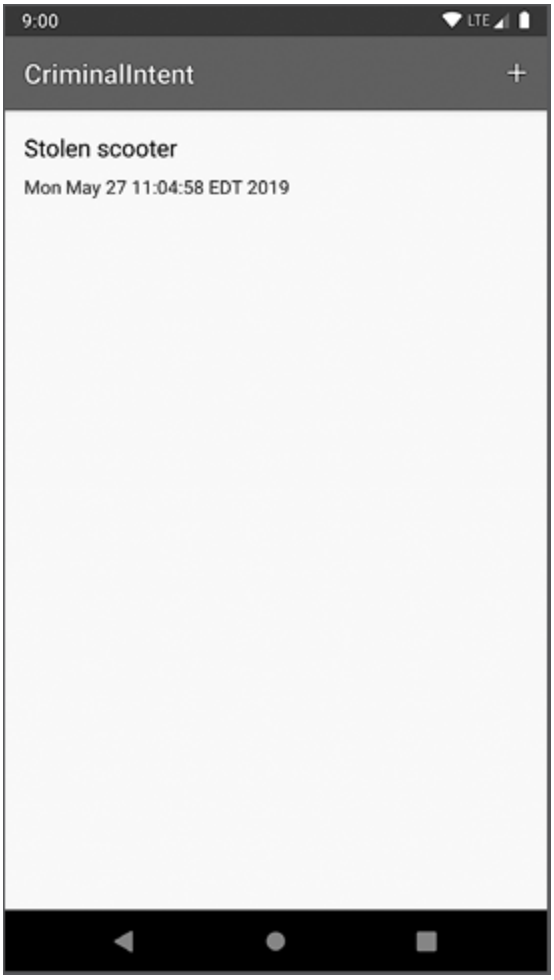


Рис. 14.13. Обновленный значок

Для любознательных: панель приложения/действий/инструментов

Можно часто слышать, как люди называют панель приложения панелью инструментов или панелью действий. В официальной документации по Android эти термины используются взаимозаменяемо. Но разве панель приложений, панель действий и панель инструментов — действительно одно и то же? Нет. Эти термины связаны, но не совсем эквивалентны.

Сам элемент дизайна пользовательского интерфейса называется панелью приложения. До Android 5.0 (Lollipop, API уровня 21) панель приложений реализовывалась с помощью класса `ActionBar`. Термины «панель действий» и «панель приложения» начали означать одно и то же. Начиная с Android 5.0 (Lollipop, API уровня 21) в качестве предпочтительного метода реализации панели приложения был представлен класс `Toolbar`.

На момент написания книги в AppCompat для реализации панели приложения используется Jetpack-виджет `Toolbar` (рис. 14.14).

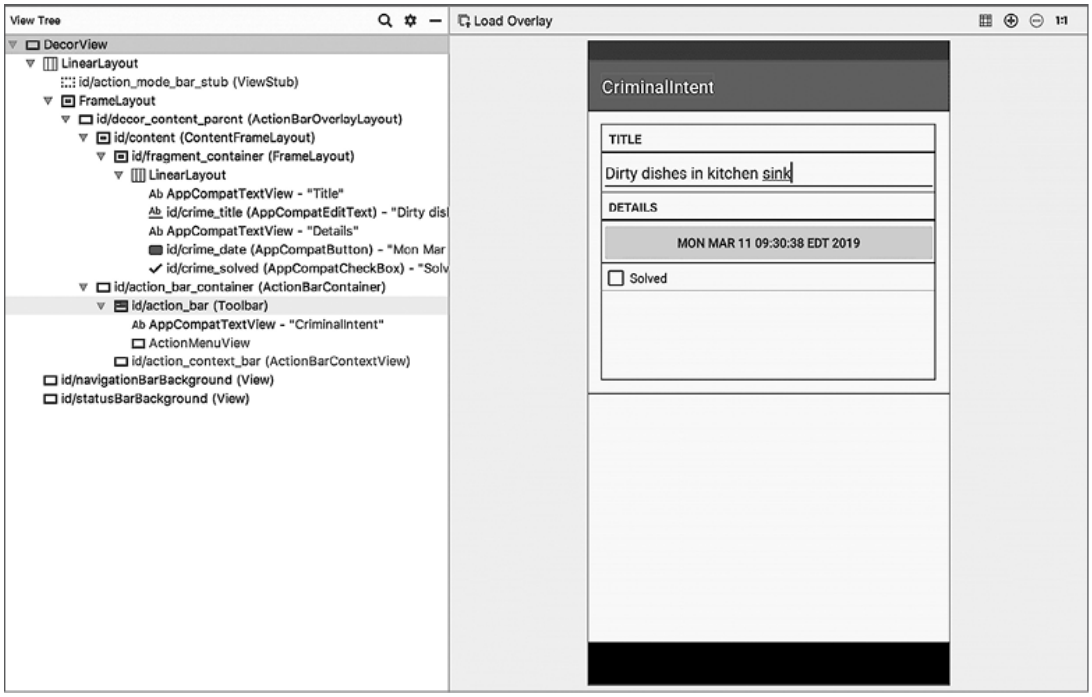


Рис. 14.14. Панель приложения в инспекторе макета

Классы `ActionBar` и `Toolbar` очень похожи компонентами. Панель инструментов строится поверх панели действий. Она имеет измененный пользовательский интерфейс и более гибкая в использовании.

Панель действий имеет множество ограничений. Она всегда отображается в верхней части экрана. Панель действий может быть только одна. Размер панели действий фиксирован и не должен меняться. Панель инструментов не имеет таких ограничений.

В этой главе мы использовали панель инструментов, позаимствованную из тем AppCompat. Также можно вручную включить панель инструментов как обычное представление в файл макета activity или фрагмента. Панель инструментов можно разместить где угодно, и на экране могут одновременно находиться несколько панелей инструментов. Эта гибкость открывает интересные возможности: например, представьте, что каждый фрагмент, используемый в вашем приложении, поддерживает собственную панель инструментов. При одновременном размещении на экране нескольких фрагментов каждый из них может отображать собственную панель инструментов, вместо того чтобы совместно использовать одну панель инструментов у верхнего края экрана.

Другое интересное дополнение к панели инструментов — возможность размещения нескольких `View` на панели инструментов и регулировки ее высоты. Все это значительно расширяет гибкость работы приложений.

Вооружившись новыми знаниями об истории API, связанной с панелью приложения, вы будете лучше ориентироваться в официальной документации разработчиков. Возможно, вы даже сможете передать эти знания будущим разработчикам Android, так как разобраться здесь ой как непросто.

Для любознательных: доступ к AppCompat панели приложения

В этой главе вы видели, что мы можем редактировать содержимое панели приложения, добавляя элементы меню. Кроме того, можно изменять другие атрибуты панели приложения во время выполнения, например отображаемый на ней заголовок.

Для доступа к панели приложения AppCompat требуется ссылаться на свойство `supportFragmentManager` у `AppCompatActivity`. В случае с `CrimeFragment` это будет выглядеть примерно так:

```
val appCompatActivity = activity as AppCompatActivity
val appBar = appCompatActivity.supportActionBar as Toolbar
```

Activity, в которой находится фрагмент, преобразуется в `AppCompatActivity`. Поскольку в приложении CriminalIntent используется библиотека AppCompat, ваша `MainActivity` стала подклассом `AppCompatActivity`, что позволяет получить доступ к панели приложения.

Преобразование `SupportActionBar` в `Toolbar` позволяет вызывать любые функции панели инструментов. (Помните, что в AppCompat для реализации панели приложения используется компонент `Toolbar`. Однако раньше для этого использовался компонент `ActionBar`, как мы поясняли выше, и поэтому с именами свойств есть некоторая путаница.)

Получив ссылку на панель приложения, вы можете вносить изменения, как показано ниже:

```
appBar.setTitle(R.string.some_cool_title)
```

Список других функций, которые можно использовать для изменения панели приложения (если ваша панель приложений является компонентом `Toolbar`), приведен на справочной API Toolbar по адресу **developer.android.com/reference/androidx/appcompat/widget/Toolbar**.

Заметим, что если вам нужно изменить содержимое меню панели приложения, пока активность отображается на экране, вы можете использовать обратный вызов функции `OnCreateOptionsMenu(Menu,MenuInflater)`, вызвав функцию `invalidateOptionsMenu()`. Вы можете программно изменить содержимое меню в окне обратного вызова `onCreateOptionsMenu`, и эти изменения будут приняты после завершения обратного вызова.

Упражнение. Пустое представление для списка

В настоящее время при запуске CriminalIntent отображает пустой виджет `RecyclerView` — большую белую пустоту. Мы должны предоставить пользователям что-то для взаимодействия при отсутствии элементов в списке.

Пусть в пустом представлении выводится сообщение (например, «Список пуст»). Добавьте в представление кнопку, которая будет инициировать создание нового преступления.

Для отображения и сокрытия нового представления-заполнителя используйте свойство `visibility`, доступное в каждом классе `View`.

15. Неявные интенты

В Android можно запустить activity из другого приложения на устройстве при помощи *неявного интента* (*implicit intent*). В *явном интенте* (*explicit intent*) задается класс запускаемой activity, а ОС запускает его. В неявном интенте вы описываете операцию, которую необходимо выполнить, а ОС запускает activity соответствующего приложения.

В приложении CriminalIntent мы будем использовать неявные интенты для выбора подозреваемых из адресной книги пользователя и отправки текстовых отчетов о преступлении. Пользователь выбирает подозреваемого в приложении адресной книги, установленном на устройстве, и получает список приложений для отправки отчета (рис. 15.1).

Использовать функциональность других приложений при помощи неявных интентов намного проще, чем писать собственные реализации стандартных задач. Пользователям также нравится работать с приложениями, которые им уже хорошо знакомы, в сочетании с вашим приложением.

Прежде чем создавать неявные интенты, необходимо выполнить с CriminalIntent ряд подготовительных действий:

- добавить в макеты `CrimeFragment` кнопки выбора подозреваемого (**CHOOSESUSPECT**) и отправки отчета (**SENDCRIMEREPORT**);
- добавить в класс `Crime` свойство `suspect`, в котором будет храниться имя подозреваемого;
- создать отчет о преступлении с использованием *форматных строк ресурсов*.

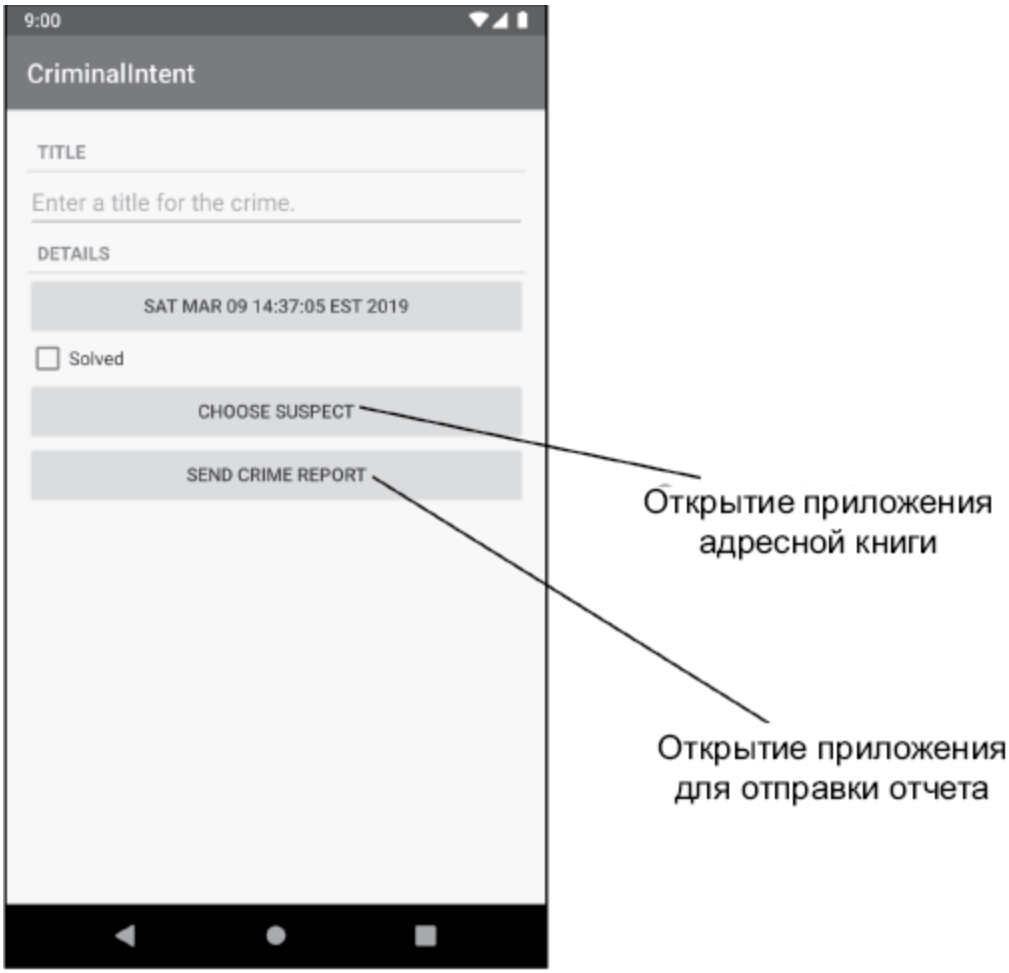


Рис. 15.1. Открытие приложений адресной книги и отправки отчетов

Добавление кнопок

Начнем с включения в макеты `CrimeFragment` новых кнопок. Прежде всего добавьте строки, которые будут отображаться на кнопках.

Листинг 15.1. Добавление строк для надписей на кнопках (strings.xml)

```
<resources>
    ...
```

```
    <string name="new_crime">New Crime</string>
      <string name="crime_suspect_text">Choose
Suspect</string>
    <string name="crime_report_text">Send Crime
Report</string>
</resources>
```

Добавьте в файл res/layout/fragment_crime.xml два виджета Button, как показано в листинге 15.2.

Листинг 15.2. Добавление кнопок CHOOSE SUSPECT и SEND CRIME REPORT (res/layout/fragment_crime.xml)

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/r
es/android"
                ... >
    ...
    <CheckBox
            android:id="@+id/crime_solved"
            android:layout_width="match_parent"
            android:layout_height="wrap_content
"
            android:text="@string/crime_solved_
label"/>

    <Button
            android:id="@+id/crime_suspect"
            android:layout_width="match_parent"
            android:layout_height="wrap_content
"
```

```
            android:text="@string/crime_suspect
_text"/>

    <Button
            android:id="@+id/crime_report"
            android:layout_width="match_parent"
            android:layout_height="wrap_content
"
            android:text="@string/crime_report_
text"/>

</LinearLayout>
```

На этой стадии вы можете проверить макеты в области предварительного просмотра или запустить приложение CriminalIntent, чтобы убедиться в правильности расположения новых кнопок.

Добавление подозреваемого в уровень модели

Теперь откройте файл `Crime.kt` и назначьте объекту `Crime` свойство, в котором будет храниться имя подозреваемого.

Листинг 15.3. Добавление свойства для подозреваемого (Crime.kt)

```
@Entity
data class Crime(@PrimaryKey val id: UUID =
UUID.randomUUID(),
                 var title: String = "",
                 var date: Date = Date(),
                 var isSolved: Boolean = false,
```

```
                    var suspect: String = "")
```

Теперь вам нужно добавить дополнительное свойство в базу данных преступлений. Для этого необходимо расширить класс базы данных `CrimeDatabase`, а также рассказать Room, как переносить базу данных между версиями.

Чтобы сообщить Room, как переводить базу данных с одной версии на другую, нужно добавить свойство `Migration`. Откройте файл `CrimeDatabase.kt`, добавьте это свойство и увеличьте номер версии.

Листинг 15.4. Добавление миграции базы данных (database/CrimeDatabase.kt)

```
@Database(entities = [ Crime::class ], version=1 version=2)
@TypeConverters(CrimeTypeConverters::class)
abstract class CrimeDatabase : RoomDatabase() {

    abstract fun crimeDao(): CrimeDao
}

val migration_1_2 = object : Migration(1, 2) {
            override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL(
                "ALTER TABLE Crime ADD COLUMN suspect TEXT NOT NULL DEFAULT ''"
        )
    }
}
```

Так как начальная версия базы данных равна 1, нужно увеличить ее до 2, а затем создать объект `Migration`, который содержит инструкции по обновлению базы данных.

Конструктор класса `Migration` принимает два параметра. Первый — это версия базы данных, из которой осуществляется миграция, а второй — версия, в которую осуществляется миграция. В данном случае вводятся номера версий 1 и 2.

Единственная функция, которую необходимо реализовать в объекте `Migration` — это `migrate(SupportSQLiteDatabase)`. Вы используете параметр базы данных для выполнения любых SQL-команд, необходимых для обновления таблиц. (В Room используется SQLite, как вы читали в главе 11.) Команда `ALTERTABLE`, которую вы написали, добавляет новый столбец в таблицу преступлений.

После создания свойства `Migration` необходимо передать ее в БД. Откройте файл `CrimeRepository.kt` и выполните миграцию в Room при создании экземпляра базы данных `CrimeDatabase`.

Листинг 15.5. Предоставление миграции в Room (CrimeRepository.kt)

```
class CrimeRepository private constructor(context: Context) {

    private val database : CrimeDatabase = Room.databaseBuilder(
        context.applicationContext,
        CrimeDatabase::class.java,
        DATABASE_NAME
    ).build()
```

```
        ).addMigrations(migration_1_2)
            .build()
        private val crimeDao = database.crimeDao()
        ...
    }
```

Чтобы настроить миграцию, перед вызовом функции `build()` вызывается функция `addMigrations(...)`. Функция `addMigrations()` принимает переменное количество объектов `Migration`, так что вы можете передать все свои миграции в момент объявления.

Когда ваше приложение запустится и Room соберет базу данных, оно сначала проверит версию существующей базы данных на устройстве. Если эта версия не соответствует той, которую вы определили в аннотации к базе данных `@Database`, Room будет искать соответствующие миграции, чтобы обновить базу данных до последней версии.

Миграция важна для преобразования вашей базы данных. Если вы не укажете миграцию, Room удалит старую версию базы данных и создаст новую версию. Это означает, что все данные будут потеряны, а это приведет к недовольству пользователей.

После того как вы реализовали миграцию, запустите CriminalIntent, чтобы удостовериться, что все построено правильно. Поведение приложения должно быть таким же, как и до того, как вы применили миграцию, и вы увидите добавленное преступление, как в главе 14. Вскоре мы воспользуемся добавленным столбцом.

Форматные строки

Последним подготовительным шагом станет создание шаблона отчета о преступлении, который заполняется информацией о конкретном преступлении. Так как подробная информация недоступна до стадии выполнения, необходимо использовать форматную строку с заполнителями, которые будут заменяться во время выполнения. Форматная строка будет выглядеть так:

```
%1$s! The crime was discovered on %2$s. %3$s,
and %4$s
```

Поля `%1$s`, `%2$s` и т.д. — заполнители для строковых аргументов. В коде вы вызываете функцию `getString(...)` и передаете форматную строку и еще четыре строки в том порядке, в каком они должны заменять заполнители.

Сначала добавьте в файл `strings.xml` строки из листинга 15.6.

Листинг 15.6. Добавление строковых ресурсов (res/values/strings.xml)

```
<resources>
    ...
      <string name="crime_suspect_text">Choose
Suspect</string>
    <string name="crime_report_text">Send Crime
Report</string>
    <string name="crime_report">%1$s!
       The crime was discovered on %2$s. %3$s,
and %4$s
    </string>
    <string name="crime_report_solved">The case
is solved</string>
```

```
    <string name="crime_report_unsolved">The
case is not solved</string>
                                    <string
name="crime_report_no_suspect">there is no
suspect.</string>
    <string name="crime_report_suspect">the
suspect is %s.</string>
                                    <string
name="crime_report_subject">CriminalIntent
Crime Report</string>
    <string name="send_report">Send crime
report via</string>
</resources>
```

В файле CrimeFragment.kt добавьте функцию, которая создает четыре строки, соединяет их и возвращает полный отчет.

Листинг 15.7. Добавление функции getCrimeReport() (CrimeFragment.kt)

```
private const val REQUEST_DATE = 0
private const val DATE_FORMAT = "EEE, MMM, dd"

class CrimeFragment : Fragment(),
DatePickerFragment.Callbacks {
    ...
    private fun updateUI() {
        ...
    }

    private fun getCrimeReport(): String {
```

```
            val solvedString = if (crime.isSolved)
{
                getString(R.string.crime_report_sol
ved)
            } else {
                getString(R.string.crime_report_uns
olved)
            }

                                val   dateString   =
DateFormat.format(DATE_FORMAT,
crime.date).toString()
                               var   suspect   =   if
(crime.suspect.isBlank()) {
                getString(R.string.crime_report_no_
suspect)
            } else {
                getString(R.string.crime_report_sus
pect, crime.suspect)
            }

            return getString(R.string.crime_report,
                            crime.title, dateString,
solvedString, suspect)
        }

        companion object {
            ...
        }
    }
```

(Обратите внимание: класс `DateFormat` существует в нескольких версиях. Используйте `android.text.format.DateFormat`.)

Приготовления завершены, теперь можно непосредственно заняться неявными интентами.

Использование неявных интентов

Объект `Intent` описывает для ОС некую операцию, которую вы хотите выполнить. Для явных интентов, использовавшихся до настоящего момента, разработчик явно указывает activity, которую должна запустить ОС:

```
val intent = Intent(this, CheatActivity::class.java)
startActivity(intent)
```

Для неявных интентов разработчик описывает выполняемую операцию, а ОС запускает activity, которая ранее сообщила о том, что она способна выполнять эту операцию. Если ОС находит несколько таких activity, пользователю предлагается выбрать нужную.

Строение неявного интента

Ниже перечислены важнейшие составляющие интента, используемые для определения выполняемой операции.

Выполняемое действие (action)

Обычно определяется константами из класса Intent. Так, для просмотра URL-адреса используется константа `Intent.ACTION_VIEW`, а для отправки данных — константа `Intent.ACTION_SEND`.

Местонахождение данных

Это может быть как ссылка на данные, находящиеся за пределами устройства (скажем, URL веб-страницы), так и URI файла или *URI контента*, ссылающийся на запись ContentProvider.

Тип данных, с которыми работает действие

Тип MIME (например, text/html или audio/mpeg3). Если в интент включено местонахождение данных, то тип обычно удается определить по этим данным.

Необязательные категории

Если действие указывает, *что* нужно сделать, категория обычно описывает, *где*, *когда* или *как* вы пытаетесь использовать операцию. Android использует категорию `android.intent.category.LAUNCHER` для обозначения activity, которые должны отображаться в лаунчере приложений верхнего уровня. С другой стороны, категория `android.intent.category.INFO` обозначает activity, которая выдает пользователю информацию о пакете, но не отображается в лаунчере.

Например, простой неявный интент для просмотра веб-сайта включает действие `Intent.ACTION_VIEW` и объект данных `Uri` с URL-адресом сайта.

На основании этой информации ОС запускает соответствующую activity соответствующего приложения. (Если ОС обнаруживает более одного кандидата, пользователю предлагается принять решение.)

Activity сообщает о себе как об исполнителе для `ACTION_VIEW` при помощи фильтра интентов в манифесте. Например, если вы пишете приложение-браузер, вы включаете следующий фильтр интентов в объявление activity, реагирующей на `ACTION_VIEW`.

```
<activity
    android:name=".BrowserActivity"
    android:label="@string/app_name" >
    <intent-filter>
                                        <action
android:name="android.intent.action.VIEW" />
                                       <category
android:name="android.intent.category.DEFAULT"
/>
                   <data  android:scheme="http"
android:host="www.bignerdranch.com" />
    </intent-filter>
</activity>
```

Категория DEFAULT должна явно задаваться в фильтрах интентов. Элемент action в фильтре интентов сообщает ОС, что activity способна выполнять операцию, а категория DEFAULT — что она желает рассматриваться среди кандидатов на выполнение операции. Категория DEFAULT неявно добавляется к почти любому неявному интенту. (Единственное исключение составляет категория LAUNCHER, с которой мы будем работать в главе 23.)

Неявные интенты, как и явные, также могут включать дополнения. Однако дополнения неявного интента не используются ОС для поиска соответствующей activity.

Также следует отметить, что компоненты действия и данных интента могут использоваться в сочетании с явными интентами. Результат равнозначен тому, как если бы вы приказали конкретной activity выполнить конкретную операцию.

Отправка отчета

Чтобы увидеть на практике, как работает эта схема, мы создадим неявный интент для отправки отчета о преступлении в приложении CriminalIntent. Операция, которую нужно выполнить, — отправка простого текста; отчет представляет собой строку. Таким образом, действие неявного интента будет представлено константой `ACTION_SEND`. Интент не содержит ссылок на данные и не имеет категорий, но определяет тип `text/plain`.

В функции `onCreateView(...)` в `CrimeFragment` получите ссылку на кнопку SEND CRIME REPORT и назначьте для нее слушателя. В реализации слушателя создайте неявный интент и передайте его `startActivity(Intent)`.

Листинг 15.8. Отправка отчета о преступлении (CrimeFragment.kt)

```
class CrimeFragment : Fragment(), DatePickerFragment.Callbacks {
    ...
        private lateinit var solvedCheckBox: CheckBox
    private lateinit var reportButton: Button
    ...
    override fun onCreateView(
        ...
    ): View? {
        ...
                                dateButton = view.findViewById(R.id.crime_date) as Button
                            solvedCheckBox = view.findViewById(R.id.crime_solved) as CheckBox
```

```
                                    reportButton    =
view.findViewById(R.id.crime_report) as Button

        return view
    }
    ...
    override fun onStart() {
        ...
        dateButton.setOnClickListener {
            ...
        }

        reportButton.setOnClickListener {
            Intent(Intent.ACTION_SEND).apply {
                type = "text/plain"
                    putExtra(Intent.EXTRA_TEXT,
getCrimeReport())
                putExtra(
                    Intent.EXTRA_SUBJECT,
                    getString(R.string.crime_re
port_subject))
            }.also { intent ->
                startActivity(intent)
            }
        }
    }
    ...
}
```

Здесь мы используем конструктор `Intent`, который получает строку с константой, описывающей действие. Также

существуют другие конструкторы, которые могут использоваться в зависимости от вида создаваемого неявного интента. Информацию о них можно найти в справочной документации `Intent`. Конструктора, получающего тип, не существует, поэтому мы задаем его явно.

Текст отчета и строка темы включаются в дополнения. Обратите внимание на использование в них констант, определенных в классе `Intent`. Любая activity, реагирующая на интент, знает эти константы и то, что следует делать с ассоциированными значениями.

Запуск activity из фрагмента работает почти так же, как и запуск activity из другой activity. Вы вызываете функцию `startActivity(Intent)` фрагмента, которая вызывает соответствующую функцию `Activity`.

Запустите приложение CriminalIntent и нажмите кнопку `SEND CRIME REPORT`. Так как этот интент с большой вероятностью совпадет со многими activity на устройстве, скорее всего, на экране появится список activity (рис. 15.2).

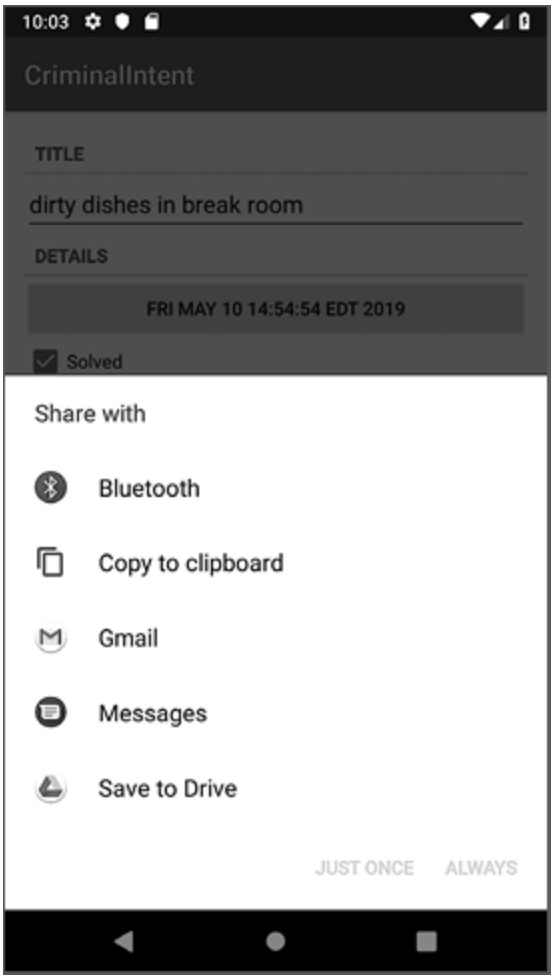


Рис. 15.2. Activity, готовые отправить ваш отчет

Если на экране появился список, выберите нужный вариант. Вы увидите, что отчет о преступлении загружается в выбранном вами приложении. Вам остается лишь ввести адрес и отправить его.

Обратите внимание, что приложения вроде Gmail и Google Drive требуют входа в аккаунт Google. Проще выбрать приложение Messages, для которого не требуется вход в систему. Нажмите кнопку **Newmessage** в диалоговом окне **Selectconversation**, введите любой номер телефона в поле **To** и нажмите на появившуюся надпись **Sendto<номертелефона>** (рис. 15.3). В теле сообщения появится отчет о преступлении.

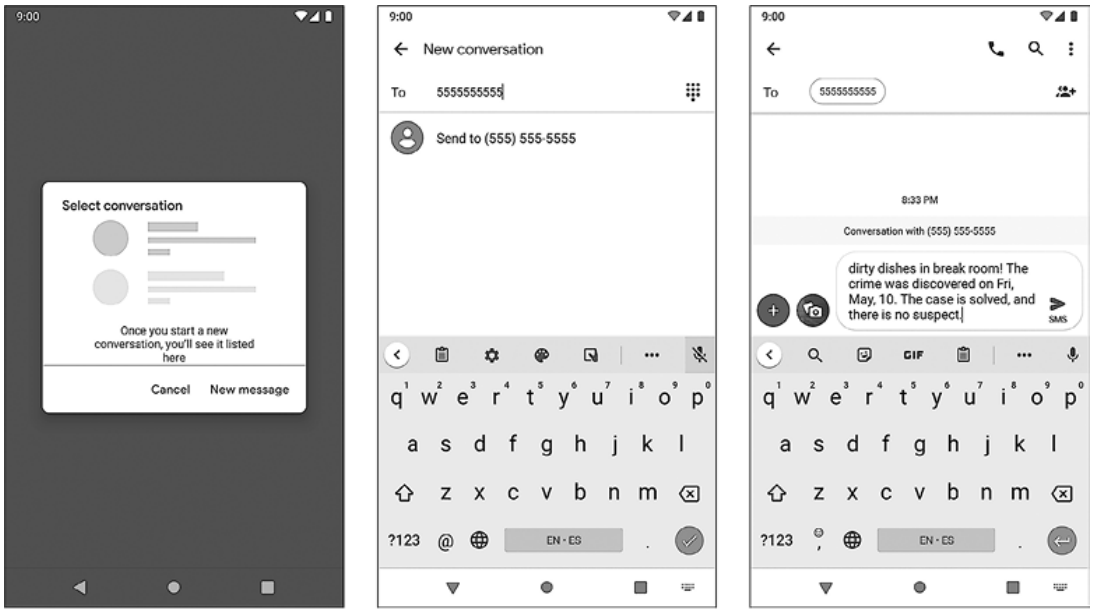


Рис. 15.3. Отправка отчета о преступлении с помощью приложения Messages

Если список не появился, это может означать одно из двух: либо вы уже назначили приложение по умолчанию для идентичного неявного интента, либо на вашем устройстве имеется всего одна activity, способная реагировать на этот интент.

Часто лучшим вариантом оказывается использование приложения по умолчанию, выбранного пользователем для действия `ACTION_SEND`. Впрочем, в приложении CriminalIntent лучше всегда предоставлять пользователю выбор: сегодня пользователь предпочтет не поднимать шум и отправит отчет по электронной почте, а завтра выставит нарушителя на общественное осуждение в «Твиттере».

Вы можете создать список, который будет отображаться каждый раз при использовании неявного интента для запуска activity. После создания неявного интента способом, показанным ранее, вы вызываете функцию `Intent.createChooser(Intent,String)` и передаете ей неявный интент и строку с выбранным заголовком.

Затем вы передаете интент, возвращенный функцией `createChooser(...)`, в `startActivity(...)`.

В файле `CrimeFragment.kt` создайте список выбора для отображения activity, реагирующих на неявный интент.

Листинг 15.9. Использование списка выбора (CrimeFragment.kt)

```
reportButton.setOnClickListener {
    Intent(Intent.ACTION_SEND).apply {
        type = "text/plain"
                    putExtra(Intent.EXTRA_TEXT,
getCrimeReport())
        putExtra(
            Intent.EXTRA_SUBJECT,
            getString(R.string.crime_report_sub
ject))
    }.also { intent ->
            ~~startActivity(intent)~~
        val chooserIntent =
                    Intent.createChooser(intent,
getString(R.string.send_report))
        startActivity(chooserIntent)
    }
}
```

Запустите приложение CriminalIntent и нажмите кнопку `SEND CRIME REPORT`. Если в системе имеется несколько activity, способных обработать ваш интент, на экране появляется список для выбора (рис. 15.4).

Запрос контакта у Android

Теперь мы создадим другой неявный интент, который предлагает пользователю выбрать подозреваемого из адресной книги. Для этого неявного интента будет определено действие и местонахождение соответствующих данных. Действие задается константой `Intent.ACTION_PICK`, а местонахождение данных — `ContactsContract.Contacts.CONTENT_URI`. Короче говоря, вы просите Android помочь с выбором записи из базы данных контактов.

Запущенная activity должна вернуть результат, поэтому мы передаем интент через `startActivityForResult(...)` вместе с кодом запроса. Добавьте в файл `CrimeFragment.kt` константу для кода запроса и поле для кнопки **CHOOSE SUSPECT**.

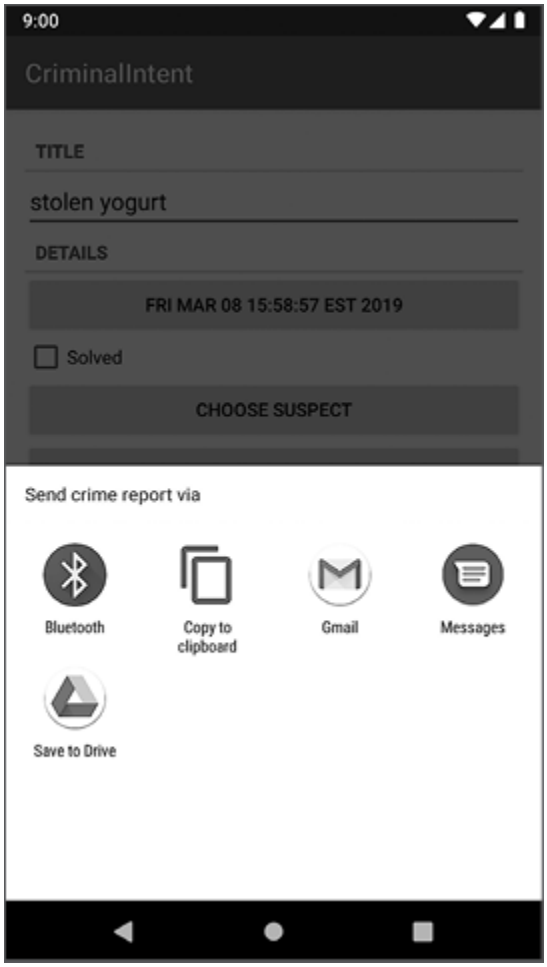


Рис. 15.4. Отправка текста с выбором activity

Листинг 15.10. Добавление поля для кнопки CHOOSE SUSPECT (CrimeFragment.kt)

```
private const val REQUEST_DATE = 0
private const val REQUEST_CONTACT = 1
private const val DATE_FORMAT = "EEE, MMM, dd"

class CrimeFragment : Fragment(),
DatePickerFragment.Callbacks {
    ...
    private lateinit var reportButton: Button
    private lateinit var suspectButton: Button
    ...
}
```

В конце `onCreateView(...)` получите ссылку на кнопку. Затем в функции `onStart()` назначьте кнопке слушателя. В реализации слушателя передайте неявный интент функции `startActivityForResult(...)`. Создайте неявный интент для запроса контакта. Также выведите на кнопке имя подозреваемого.

Листинг 15.11. Отправка неявного интента (CrimeFragment.kt)

```
class CrimeFragment : Fragment(),
DatePickerFragment.Callbacks {
    ...
    override fun onCreateView(
        ...
    ): View? {
        ...
```

```
                                    reportButton    =
view.findViewById(R.id.crime_report) as Button
                                    suspectButton    =
view.findViewById(R.id.crime_suspect) as Button

        return view
    }
    ...
    override fun onStart() {
        ...
        reportButton.setOnClickListener {
            ...
        }

        suspectButton.apply {
            val pickContactIntent =
                    Intent(Intent.ACTION_PICK,
ContactsContract.Contacts.CONTENT_URI)

            setOnClickListener {
                startActivityForResult(pickCont
actIntent, REQUEST_CONTACT)
            }
        }
    }
    ...
}
```

Мы еще воспользуемся интентом `pickContact`, поэтому он размещается за пределами слушателя `OnClickListener`.

Изменим функцию `updateUI()`, чтобы установить текст на кнопке **CHOOSE SUSPECT**, если у преступления есть подозреваемый.

Листинг 15.12. Настройка текста кнопки CHOOSE SUSPECT (CrimeFragment.kt)

```
private fun updateUI() {
    titleField.setText(crime.title)
    dateButton.text = crime.date.toString()
    solvedCheckBox.apply {
        isChecked = crime.isSolved
        jumpDrawablesToCurrentState()
    }
    if (crime.suspect.isNotEmpty()) {
        suspectButton.text = crime.suspect
    }
}
```

Запустите приложение CriminalIntent и нажмите кнопку **CHOOSE SUSPECT**. На экране отобразится список контактов из адресной книги (рис. 15.5).

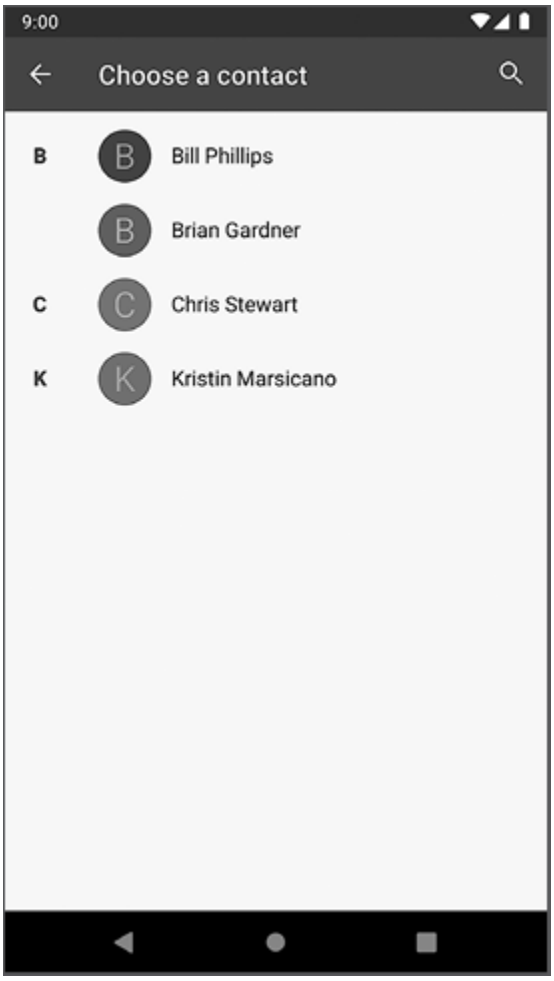


Рис. 15.5. Список подозреваемых

Если у вас установлено другое приложение адресной книги, экран будет выглядеть иначе. Это еще одно преимущество неявных интентов: вам не нужно знать название приложения адресной книги, чтобы использовать его из нашего проекта. Соответственно, пользователь может установить то приложение, которое считает нужным, а ОС найдет и запустит его.

Получение данных из адресной книги

Теперь необходимо получить результат от приложения адресной книги. Контактная информация совместно

используется многими приложениями, поэтому Android предоставляет расширенный API для работы с контактами через `ContentProvider`. Экземпляры этого класса инкапсулируют базы данных и предоставляют доступ к ним другим приложениям. Обращение к `ContentProvider` осуществляется через `ContentResolver`. (Разговор о базе данных контактов — сама по себе обширная тема, и мы не будем обсуждать ее здесь. Если вы хотите узнать больше, ознакомьтесь с руководством по API Content Provider по адресу **developer.android.com/guide/topics/providers/content-provider-basics**.)

Так как activity запускалась с возвращением результата с использованием `ACTION_PICK`, вы можете получить интент вызовом `onActivityResult(...)`. Интент включает URI данных — ссылку на конкретный контакт, выбранный пользователем.

В файле `CrimeFragment.kt` добавьте код в реализацию `onActivityResult(...)` из `CrimeFragment`, запрашивающий имя контакта из адресной книги. Этот фрагмент кода большой, поэтому мы разберем его подробно.

Листинг 15.13. Получение имени контакта (CrimeFragment.kt)

```
class CrimeFragment : Fragment(),
DatePickerFragment.Callbacks {
    ...
    private fun updateUI() {
        ...
    }

    override fun onActivityResult(requestCode:
Int, resultCode: Int, data: Intent?) {
```

```
        when {
            resultCode != Activity.RESULT_OK ->
return

             requestCode == REQUEST_CONTACT &&
data != null -> {
                        val contactUri: Uri? =
data.data
                // Указать, для каких полей ваш
запрос должен возвращать значения.
                            val queryFields =
arrayOf(ContactsContract.Contacts.DISPLAY_NAME)
                 // Выполняемый здесь запрос —
contactUri похож на предложение "where"
                                val cursor =
requireActivity().contentResolver
                                .query(contactUri,
queryFields, null, null, null)
                cursor?.use {
                      // Verify cursor contains
at least one result
                    if (it.count == 0) {
                        return
                    }

                       // Первый столбец первой
строки данных —
                              // это имя вашего
подозреваемого.
                    it.moveToFirst()
                                val suspect =
it.getString(0)
```

```
                        crime.suspect = suspect
                        crimeDetailViewModel.saveCr
ime(crime)
                                suspectButton.text =
suspect
                }
            }
        }
    }
    ...
}
```

В листинге 15.13 создается запрос всех отображаемых имен контактов в возвращенных данных. Затем вы запрашиваете базу данных контактов и получаете объект `Cursor`, с которым мы работаем. После проверки того, что возвращенный курсор содержит хотя бы одну строку, вы вызываете функцию `Cursor.moveToFirst()` для перемещения курсора в первую строку. Наконец, вы вызываете функцию `Cursor.getString(Int)` для перемещения содержимого первого столбца в виде строки. Эта строка будет именем подозреваемого, и вы используете ее для установки подозреваемого в преступлении и текста для кнопки **CHOOSE SUSPECT**.

Вы также сохраняете информацию о преступлении в базу данных, как только получаете данные о подозреваемом. Это нужно сделать потому, что при возобновлении `CrimeFragment` вызывается функция `onViewCreated(...)`, и вы запросите преступление из базы данных. Но функция `onActivityResult(...)` вызывается перед `onViewCreated(...)`, так что при получении названия преступления из базы данных оно запишет в преступление

информацию о подозреваемом. Чтобы не потерять данные о подозреваемом, вы записываете преступление вместе с подозреваемым.

Теперь информация будет сохранена в базе данных. `CrimeFragment` все равно сначала получит старую информацию о преступлении, но `LiveData` будет сразу уведомляться о новом преступлении, как только обновление будет завершено.

Через мгновение мы запустим приложение. Обязательно запускайте его на устройстве, где есть приложение адресной книги, — используйте эмулятор, если на вашем Android-устройстве его нет. Если вы используете эмулятор, добавьте несколько контактов с помощью своего приложения адресной книги, прежде чем запускать CriminalIntent. Затем запустите приложение.

Выберите подозреваемого. Имя подозреваемого, которого вы выбрали, должно появиться на кнопке **CHOOSE SUSPECT**. Затем отправьте отчет о преступлении. Имя подозреваемого должно появиться в отчете о преступлении (рис. 15.6).

Разрешение контактов

Как получить разрешение на чтение из базы данных контактов? Приложение адресной книги распространяет свои разрешения на вас. Оно обладает полными разрешениями на обращение к базе данных. Когда приложение адресной книги возвращает родительской activity URI данных в `Intent`, оно также добавляет флаг `Intent.FLAG_GRANT_READ_URI_PERMISSION`. Этот флаг сообщает Android, что родительской activity в CriminalIntent следует разрешить однократное использование этих данных.

Такой подход работает хорошо, потому что фактически нам нужен доступ не ко всей базе данных контактов, а к одному.

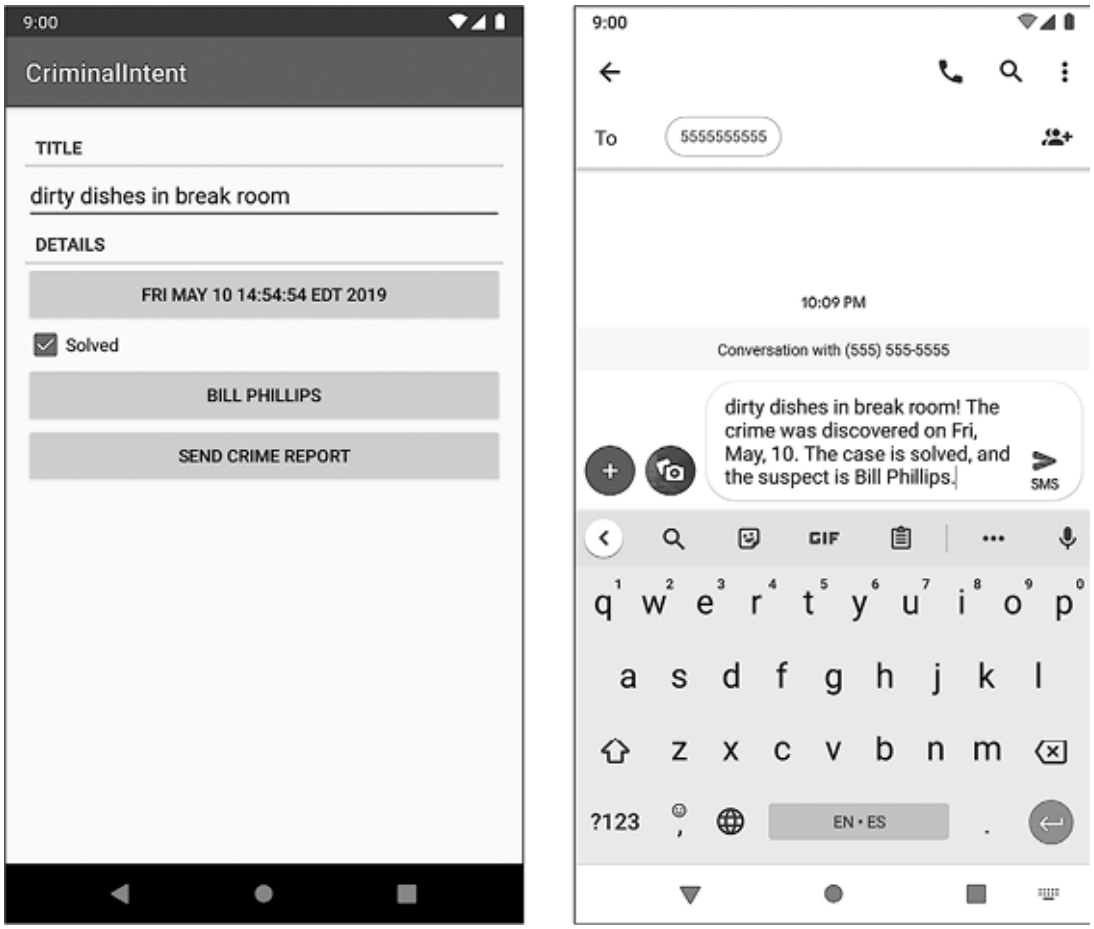


Рис. 15.6. Имя подозреваемого на кнопке и в отчете о преступлении

Проверка реагирующих activity

На первый неявный интент, созданный в этой главе, кто-то гарантированно отреагирует: даже если способа отправки отчета не существует, окно выбора все равно будет отображено. Со вторым интентом дело обстоит иначе: на некоторых устройствах (или у некоторых пользователей) может не оказаться приложения адресной книги. Если ОС не найдет подходящую activity, в приложении происходит сбой.

Проблема решается предварительной проверкой того, от какой части ОС поступил вызов PackageManager. Это удобно сделать в функции onStart().

Листинг 15.14. Защита от отсутствия приложений адресной книги (CrimeFragment.kt)

```
override fun onStart() {
    ...
    suspectButton.apply {
        val pickContactIntent =
                    Intent(Intent.ACTION_PICK,
ContactsContract.Contacts.CONTENT_URI)

        setOnClickListener {
            startActivityForResult(pickContactI
ntent, REQUEST_CONTACT)
    }

          val packageManager: PackageManager =
requireActivity().packageManager
        val resolvedActivity: ResolveInfo? =
            packageManager.resolveActivity(pick
ContactIntent,
                PackageManager.MATCH_DEFAULT_ONL
Y)
        if (resolvedActivity == null) {
            isEnabled = false
        }
    }
}
```

PackageManager известно все о компонентах, установленных на устройстве Android, включая все его activity. (Другие компоненты встретятся вам позднее в этой книге.) Вызывая resolveActivity(Intent,int), вы приказываете найти activity, соответствующую переданному интенту. Флаг MATCH_DEFAULT_ONLY ограничивает поиск activity с флагом CATEGORY_DEFAULT (по аналогии с startActivity(Intent)).

Если поиск прошел успешно, возвращается экземпляр ResolveInfo, который сообщает полную информацию о найденной activity. С другой стороны, если поиск вернул null, все кончено — приложения адресной книги нет, поэтому бесполезная кнопка просто блокируется.

Если вы хотите убедиться в том, что фильтр работает, но не располагаете устройством без приложения адресной книги, временно добавьте в интент дополнительную категорию. Эта категория предотвращает возможные совпадения приложений адресной книги с вашим интентом.

Листинг 15.15. Фиктивный код для проверки фильтра (CrimeFragment.kt)

```
override fun onStart() {
    ...
    suspectButton.apply {
        ...
        pickContactIntent.addCategory(Intent.CATEGORY_HOME)
          val packageManager: PackageManager = requireActivity().packageManager
        ...
    }
}
```

На этот раз кнопка выбора подозреваемого недоступна (рис. 15.7).

После завершения проверки удалите фиктивный код.

Листинг 15.16. Удаление фиктивного кода (CrimeFragment.kt)

```
override fun onStart() {
    ...
    suspectButton.apply {
        ...
            pickContactIntent.addCategory(Intent.CATEGORY_HOME)
          val packageManager: PackageManager =
requireActivity().packageManager
        ...
    }
}
```

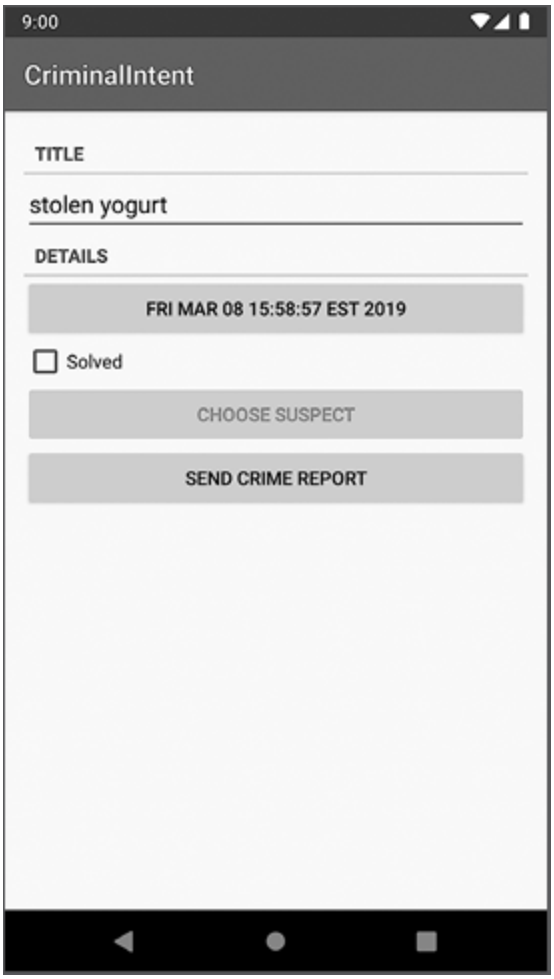


Рис. 15.7. Заблокированная кнопка выбора подозреваемого

Упражнение. Другой неявный интент

Возможно, вместо отправки отчета разгневанный пользователь предпочтет разобраться с подозреваемым по телефону. Добавьте новую кнопку для звонка подозреваемому.

Вам понадобится извлечь номер телефона из базы данных контактов. Для этого необходимо обратиться с запросом к другой таблице базы данных, `ContactsContract`, которая называется `CommonDataKinds.Phone`. За дополнительными сведениями о том, как получить эту информацию, обращайтесь

к документации ContactsContract и ContactsContract.CommonDataKinds.Phone.

Пара подсказок: для запроса дополнительных данных можно воспользоваться разрешением android.permission.READ_CONTACTS. Это *разрешение во время выполнения* (*runtime permission*), поэтому вам нужно явно запросить у пользователя разрешение на доступ к его адресной книге. Если вы хотите узнать больше, посетите сайт **developer.android.com/training/permissions/requesting**.

С этим разрешением вы сможете прочитать ContactsContract.Contacts._ID для получения идентификатора контакта из исходного запроса. Затем полученный идентификатор используется для получения данных из таблицы CommonDataKinds.Phone.

После получения телефонного номера можно создать неявный интент с URI телефона:

```
Uri number = Uri.parse("tel:5551234");
```

При этом может использоваться действие Intent.ACTION_DIAL или Intent.ACTION_CALL. ACTION_CALL запускает телефонное приложение и немедленно осуществляет звонок по номеру, отправленному в интент; ACTION_DIAL только вводит номер и ждет, пока пользователь инициирует звонок.

Мы рекомендуем использовать ACTION_DIAL. Режим ACTION_CALL может быть ограничен, и для него определенно потребуются разрешения. Кроме того, у пользователя будет возможность немного остыть перед нажатием кнопки вызова.

16. Интенты при работе с камерой

Итак, вы научились работать с неявными интентами, и теперь преступления будут документироваться еще точнее. Располагая снимком места преступления, вы сможете поделиться жуткими подробностями со всеми желающими. Для создания снимков вам понадобится пара новых инструментов, которые используются в сочетании с уже знакомыми вам неявными интентами.

Неявный интент используется при запуске любимого приложения для работы с камерой и получения от него нового снимка. Неявный интент может создать снимок, но где его разместить? И как вывести на экран созданную фотографию? В этой главе даны ответы на оба вопроса.

Размещение фотографий

Прежде всего следует обустроить место, в котором будет «жить» ваша фотография. Для этого понадобятся два новых объекта `View`: `ImageView` для отображения фотографии на экране и `Button` для создания снимка (рис. 16.1).

Если выделить отдельную строку под миниатюру и кнопку, приложение будет выглядеть убого и непрофессионально. Чтобы этого не произошло, необходимо аккуратно разместить все компоненты на экране.

Добавьте новые представления создания снимка в `res/layout/fragment_crime.xml`, чтобы сформировать новую область. Начните с построения левой стороны: добавьте `ImageView` для изображения и `ImageButton` для размещения фото.

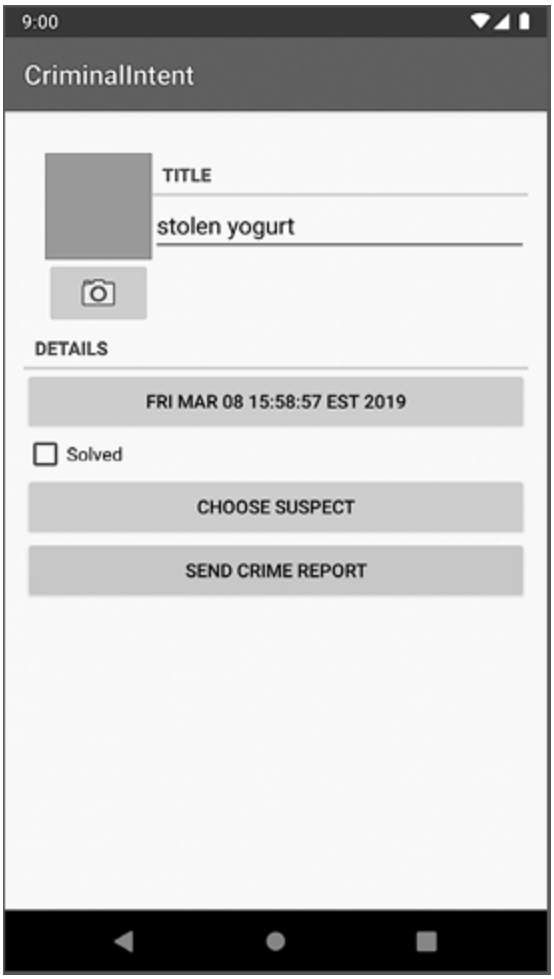


Рис. 16.1. Новый интерфейс

Листинг 16.1. Добавление в макет кнопок изображения и камеры (res/layout/fragment_crime.xml)

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
                ... >
    <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content
"
```

```
            android:orientation="horizontal"
            android:layout_marginStart="16dp"
            android:layout_marginTop="16dp">

        <LinearLayout
                android:layout_width="wrap_cont
ent"
                android:layout_height="wrap_con
tent"
                android:orientation="vertical">

            <ImageView
                    android:id="@+id/crime_phot
o"
                    android:layout_width="80dp"
                    android:layout_height="80dp
"
                    android:scaleType="centerIn
side"
                    android:cropToPadding="true
"
                    android:background="@androi
d:color/darker_gray"/>

            <ImageButton
                    android:id="@+id/crime_came
ra"
                    android:layout_width="match
_parent"
                    android:layout_height="wrap
_content"
```

```
                    android:src="@android:drawa
ble/ic_menu_camera"/>
        </LinearLayout>
    </LinearLayout>

    <TextView
                                        style="?
android:listSeparatorTextViewStyle"
            android:layout_width="match_parent"
            android:layout_height="wrap_content
"
            android:text="@string/crime_title_l
abel"/>
    ...
</LinearLayout>
```

Затем создайте правую сторону: переместите заголовочный виджет `TextView` и `EditText` в нового потомка `LinearLayout` виджета `LinearLayout`.

Листинг 16.2. Обновление макета заголовка (res/layout/fragment_crime.xml)

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/r
es/android"
... >
    <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content
"
            android:orientation="horizontal"
```

```
            android:layout_marginStart="16dp"
            android:layout_marginTop="16dp">
        <LinearLayout
                android:layout_width="wrap_cont
ent"
                android:layout_height="wrap_con
tent"
                android:orientation="vertical">
                ...
        </LinearLayout>
        </LinearLayout>

        <LinearLayout
                android:orientation="vertical"
                android:layout_width="0dp"
                android:layout_height="wrap_con
tent"
                android:layout_weight="1">

            <TextView
                                        style="?
android:listSeparatorTextViewStyle"
                    android:layout_width="match
_parent"
                    android:layout_height="wrap
_content"
                    android:text="@string/crime
_title_label"/>

            <EditText
                    android:id="@+id/crime_titl
e"
```

```
                    android:layout_width="match
_parent"
                    android:layout_height="wrap
_content"
                    android:hint="@string/crime
_title_hint"/>
        </LinearLayout>
    </LinearLayout>
    ...
</LinearLayout>
```

Запустите приложение CriminalIntent; новый пользовательский интерфейс должен выглядеть так, как показано на рис. 16.1.

Выглядит замечательно, но для реагирования на нажатия `ImageButton` и управления содержимым `ImageView` нам понадобятся переменные экземпляров со ссылками на оба виджета. Как обычно, вызовите функцию `findViewById(int)` для заполняемого макета `fragment_crime.xml`, чтобы найти представления и подключить их.

Листинг 16.3. Добавление переменных экземпляров (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {
    ...
    private lateinit var suspectButton: Button
        private lateinit var photoButton:
ImageButton
    private lateinit var photoView: ImageView
        private val crimeDetailViewModel:
CrimeDetailViewModel by lazy {
```

```
        ViewModelProviders.of(this).get(CrimeDe
tailViewModel::class.java)
    }
    ...
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        ...
                                suspectButton    =
view.findViewById(R.id.crime_suspect) as Button
                                  photoButton    =
view.findViewById(R.id.crime_camera)          as
ImageButton
                                    photoView    =
view.findViewById(R.id.crime_photo)           as
ImageView

        return view
    }
    ...
}
```

На этом ненадолго оставим пользовательский интерфейс (через несколько страниц будет дано объяснение, как эти кнопки подключить к логике приложения).

Хранилище файлов

Одного лишь места на экране вашим фотографиям недостаточно. Полноразмерная фотография слишком велика

для хранения в базе данных SQLite, не говоря уже об интенте. Ей необходимо место для хранения в файловой системе устройства.

Обычно такие данные размещаются в закрытом (приватном) хранилище. Вспомните, что именно в закрытой области хранится наша база данных SQLite. Такие функции, как `Context.getFileStreamPath(String)` и `Context.getFilesDir()`, позволяют хранить в этой же области и обычные файлы (в папке по соседству с папкой `databases`, в которой размещается база данных SQLite).

Основные функции для работы с внешними файлами и каталогами в классе `Context`:

`getFilesDir():File`

возвращает дескриптор каталога для приватных файлов приложения.

`openFileInput(name:String):FileInputStream`

открывает существующий файл для ввода (относительно каталога файлов).

`openFileOutput(name:String,mode:Int):FileOutputStream`

открывает существующий файл для вывода, возможно, создает его (относительно каталога файлов).

`getDir(name:String,mode:Int):File`

получает (и, возможно, создает) подкаталог в каталоге файлов.

`fileList(...):Array<String>`

получает список имен файлов в главном каталоге файлов (например, для использования с `openFileInput(String)`).

`getCacheDir():File`

возвращает дескриптор каталога, используемого для хранения кэш-файлов. Будьте внимательны, поддерживайте порядок в этом каталоге и старайтесь использовать как можно меньше пространства.

Тут есть подвох. Поскольку эти файлы объявлены приватными, взаимодействовать с ними может *только ваше собственное приложение*. Если доступ к этим файлам не понадобится другому приложению, можно обойтись и этими функциями.

Но их окажется недостаточно, если какому-то другому приложению потребуется выполнить запись в ваши файлы. Это как раз случай CriminalIntent, потому что приложению камеры нужно будет сохранить файл снимка в вашем приложении.

С другой стороны, если запись в файлы должна осуществляться другим приложением, вам не повезло: хотя существует флаг `Context.MODE_WORLD_READABLE`, который можно передать при вызове `openFileOutput(String,int)`, он официально считается устаревшим, а на новых устройствах его надежность не гарантирована. Когда-то существовала возможность передачи файлов в общедоступное внешнее хранилище, но в последних версиях она была заблокирована по соображениям безопасности.

Если вы сохраняете файлы, которые должны использоваться другими приложениями, или получаете файлы от других приложений (как, например, сохраненные фотографии), к файлам необходимо организовать доступ через `ContentProvider`. `ContentProvider` позволяет открыть доступ к URI контента другим приложениям. Тогда эти приложения смогут загружать или записывать данные по этим URI. В любом случае ситуация находится под вашим контролем, и вы всегда можете запретить чтение или запись по своему желанию.

Использование FileProvider

Если ваши потребности ограничиваются получением файла из другого приложения, полноценная реализация `ContentProvider` — это перебор. К счастью, Google предоставляет вспомогательный класс `FileProvider`, который берет на себя выполнение всех задач, кроме настройки конфигурации.

Прежде всего следует объявить `FileProvider` экземпляром `ContentProvider`, который связан с *конкретным хранилищем* (*authority*). Для этого объявление `ContentProvider` включается в манифест.

Листинг 16.4. Добавление объявления FileProvider (manifests/AndroidManifest.xml)

```
<activity android:name=".MainActivity">
    ...
</activity>
<provider
        android:name="androidx.core.content.FileProvider"
        android:authorities="com.bignerdranch.android.criminalintent.fileprovider"
        android:exported="false"
        android:grantUriPermissions="true">
</provider>
```

Хранилищем называется место, в котором будут сохраняться файлы. Строка, которую вы выбираете для `android:authorities`, должна быть уникальной по всей системе. Чтобы гарантировать это, принимается соглашение о подготовке строки полномочий с именем вашего пакета. (Мы

показываем имя пакета com.bignerdranch.android.criminalintent выше. Если имя пакета вашего приложения другое, используйте вместо него имя вашего пакета).

Связывая FileProvider с хранилищем, вы предоставляете другим приложениям цель для запросов. Добавляя атрибут exported="false", вы запрещаете использование провайдера всеми сторонами, которым вы не предоставите разрешение. Добавляя атрибут grantUriPermissions, вы добавляете возможность предоставлять другим приложениям право записи по URI для этого хранилища, передаваемым в интентах. (Об этом чуть позднее.)

Теперь, когда вы сообщили Android, где находится FileProvider, также необходимо сообщить FileProvider, какие файлы предоставляются. Эта конфигурация определяется в дополнительном ресурсном файле в формате XML. Щелкните правой кнопкой мыши по папке app/res на панели **Project** и выберите команду **New⇒Androidresourcefile** в контекстном меню. Выберите тип ресурса XML и введите имена файлов.

Откройте файл res/xml/files.xml и добавьте туда путь к файлу:

Листинг 16.5. Заполнение описания путей (res/xml/files.xml)

```
    <PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">

    </PreferenceScreen>
<paths>
    <files-path name="crime_photos" path="."/>
</paths>
```

Этот файл XML по сути гласит: «Назначить crime_photos корневым путем моего приватного хранилища». Вы не будете использовать имя crime_photos — FileProvider делает это в своей внутренней реализации.

Теперь свяжите files.xml с FileProvider, добавив тег meta-data в файл AndroidManifest.xml.

Листинг 16.6. Связывание с описанием путей (manifests/AndroidManifest.xml)

```
<provider
        android:name="androidx.core.content.Fil
eProvider"
        android:authorities="com.bignerdranch.a
ndroid.criminalintent.fileprovider"
        android:exported="false"
        android:grantUriPermissions="true">
    <meta-data
            android:name="android.support.FILE_
PROVIDER_PATHS"
            android:resource="@xml/files"/>
</provider>
```

Выбор локации фотографии

Пора выделить фотографиям место, где они будут размещаться. Сначала добавьте в Crime вычисляемое свойство для получения подходящего имени файла.

Листинг 16.7. Добавление свойства для получения имени файла (Crime.kt)

```
@Entity
data class Crime(@PrimaryKey val id: UUID =
UUID.randomUUID(),
                 var title: String = "",
                 var date: Date = Date(),
                 var isSolved: Boolean = false,
                 var suspect: String = "") {

    val photoFileName
        get() = "IMG_$id.jpg"
}
```

Функция `photoFileName()` не знает, в какой папке будет храниться фотография. Однако имя файла будет уникальным, поскольку оно строится на основании идентификатора `Crime`.

Затем следует найти место, где расположить фотографии. Класс `CrimeRepository` отвечает за все, что относится к долгосрочному хранению данных в CriminalIntent, поэтому он становится наиболее естественным кандидатом. Добавьте в `CrimeRepository` функцию `getPhotoFile(Crime)`, которая будет возвращать эту информацию.

Листинг 16.8. Определение местонахождения файла фотографии (CrimeRepository.kt)

```
class CrimeRepository private
constructor(context: Context) {
    ...
                private val executor =
Executors.newSingleThreadExecutor()
                private val filesDir =
context.applicationContext.filesDir
```

```
    fun addCrime(crime: Crime) {
        ...
    }

    fun getPhotoFile(crime: Crime): File =
File(filesDir, crime.photoFileName)
...
}
```

Этот код не создает файлов в файловой системе. Он лишь возвращает объекты `File`, указывающие в нужные места. Позднее мы используем класс `FileProvider` для предоставления доступа к этим местам в виде URI-адресов.

Наконец, добавим в `CrimeDetailViewModel` функцию, выдающую информацию о файле для `CrimeFragment`.

Листинг 16.9. Выдача файла через CrimeDetailViewModel (CrimeDetailViewModel.kt)

```
class CrimeDetailViewModel : ViewModel() {
    ...
    fun saveCrime(crime: Crime) {
        crimeRepository.updateCrime(crime)
    }

    fun getPhotoFile(crime: Crime): File {
                                        return
crimeRepository.getPhotoFile(crime)
    }
}
```

Использование интента камеры

Следующий шаг — непосредственное создание снимка. Здесь все просто: необходимо снова воспользоваться неявным интентом.

Начнем с сохранения местонахождения файла фотографии. Эта информация будет использоваться еще в нескольких местах, поэтому сохранение избавит от лишней работы.

Листинг 16.10. Сохранение местонахождения файла фотографии (CrimeFragment.kt)

```
class CrimeFragment : Fragment(), DatePickerFragment.Callbacks {

    private lateinit var crime: Crime
    private lateinit var photoFile: File
    ...
       override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
            ...
            crimeDetailViewModel.crimeLiveData.observe(
                viewLifecycleOwner,
                Observer { crime ->
                    crime?.let {
                        this.crime = crime
                        photoFile = crimeDetailViewModel.getPhotoFile(crime)
                        updateUI()
                    }
                })
```

```
        }
        ...
    }
```

На следующем шаге мы подключим кнопку, которая будет непосредственно создавать снимок. Интент камеры определяется в `MediaStore` — верховном повелителе всего, что относится к аудиовизуальной информации в Android. Интент отправляется действием `MediaStore.ACTION_IMAGE_CAPTURE`, по которому Android запускает activity камеры и делает снимок за вас.

Но не стоит торопить события.

Отправка интента

Теперь все готово к отправке интента камеры. Нужное действие `ACTION_CAPTURE_IMAGE` определяется в классе `MediaStore`. Этот класс определяет открытые интерфейсы, используемые в Android при работе с основными аудиовизуальными материалами — изображениями, видео и музыкой. К этой категории относится и интент, запускающий камеру.

По умолчанию `ACTION_CAPTURE_IMAGE` послушно запускает приложение камеры и делает снимок, но результат не является фотографией в полном разрешении. Вместо нее создается миниатюра с малым разрешением, которая упаковывается в объект `Intent`, возвращаемый в `onActivityResult(...)`.

Чтобы получить выходное изображение в высоком разрешении, необходимо сообщить, где должно храниться изображение в файловой системе. Эта задача решается передачей URI для локации, в которой должен сохраняться

файл, в MediaStore.EXTRA_OUTPUT. URI будет указывать на место, предоставленное FileProvider.

Сначала создайте новое свойство для URI фотографии и инициализируйте его после того, как появится ссылка на photoFile.

Листинг 16.11. Добавление свойства URI фотографии (CrimeFragment.kt)

```
class CrimeFragment : Fragment(), DatePickerFragment.Callbacks {

    private lateinit var crime: Crime
    private lateinit var photoFile: File
    private lateinit var photoUri: Uri
    ...
       override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        ...
        crimeDetailViewModel.crimeLiveData.observe(
            viewLifecycleOwner,
            Observer { crime ->
                crime?.let {
                    this.crime = crime
                                    photoFile = crimeDetailViewModel.getPhotoFile(crime)
                                    photoUri = FileProvider.getUriForFile(requireActivity(),
                        "com.bignerdranch.android.criminalintent.fileprovider",
                        photoFile)
```

```
                    updateUI()
                }
            })
    }
    ...
}
```

Вызов `FileProvider.getUriForFile(...)` преобразует локальный путь к файлу в `Uri`, который видит приложение камеры. Функция принимает на вход activity, провайдера и файл фотографии для создания URI, который указывает на файл. Строка авторизации, которую вы передаете в `FileProvider.getUriForFile(...)`, должна соответствовать строке авторизации, которую вы определили в манифесте (листинг 16.4).

Напишите неявный интент для сохранения фотографии в месте, определяемом `photoUri` (листинг 16.12). Добавьте код, блокирующий кнопку при отсутствии приложения камеры или недоступности места, в котором должна сохраняться фотография. (Чтобы проверить доступность приложения камеры, запросите у `PackageManager` activity, реагирующие на неявный интент камеры. Об этом более подробно рассказано в разделе «Проверка реагирующих activity» главы 15.)

Листинг 16.12. Отправка интента камеры (CrimeFragment.kt)

```
private const val REQUEST_CONTACT = 1
private const val REQUEST_PHOTO = 2
private const val DATE_FORMAT = "EEE, MMM, dd"

class CrimeFragment : Fragment(),
DatePickerFragment.Callbacks {
```

```
    ...
    override fun onStart() {
        ...
        suspectButton.apply {
            ...
        }

        photoButton.apply {
             val packageManager: PackageManager
= requireActivity().packageManager

                          val captureImage =
Intent(MediaStore.ACTION_IMAGE_CAPTURE)
             val resolvedActivity: ResolveInfo?
=
                packageManager.resolveActivity(
captureImage,
                        PackageManager.MATCH_DE
FAULT_ONLY)
            if (resolvedActivity == null) {
                isEnabled = false
            }

            setOnClickListener {
                captureImage.putExtra(MediaStor
e.EXTRA_OUTPUT, photoUri)

                          val cameraActivities:
List<ResolveInfo> =
                    packageManager.queryIntentA
ctivities(captureImage,
```

```
                                PackageManager.MATC
H_DEFAULT_ONLY)

                            for (cameraActivity in
cameraActivities) {
                        requireActivity().grantUriP
ermission(
                            cameraActivity.activity
Info.packageName,
                            photoUri,
                            Intent.FLAG_GRANT_WRITE
_URI_PERMISSION)
                    }

                    startActivityForResult(captureI
mage, REQUEST_PHOTO)
                }
            }
            return view
        }
        ...
}
```

Чтобы записывать `photoUri`, вы должны предоставить разрешение приложению камеры. Для этого вы устанавливаете флаг `Intent.FLAG_GRANT_WRITE_URI_PERMISSION` для каждой activity, которую может обрабатывать интент `cameraImage`. Так мы предоставляем им всем разрешение на запись специально для этого `Uri`. Добавление атрибута `android:grantUriPermissions` в объявление провайдера было необходимо для раскрытия этой функциональности.

Позже вы отзовете это разрешение, чтобы закрыть потенциальную уязвимость.

Запустите CriminalIntent и нажмите кнопку запуска приложения камеры (рис. 16.2).

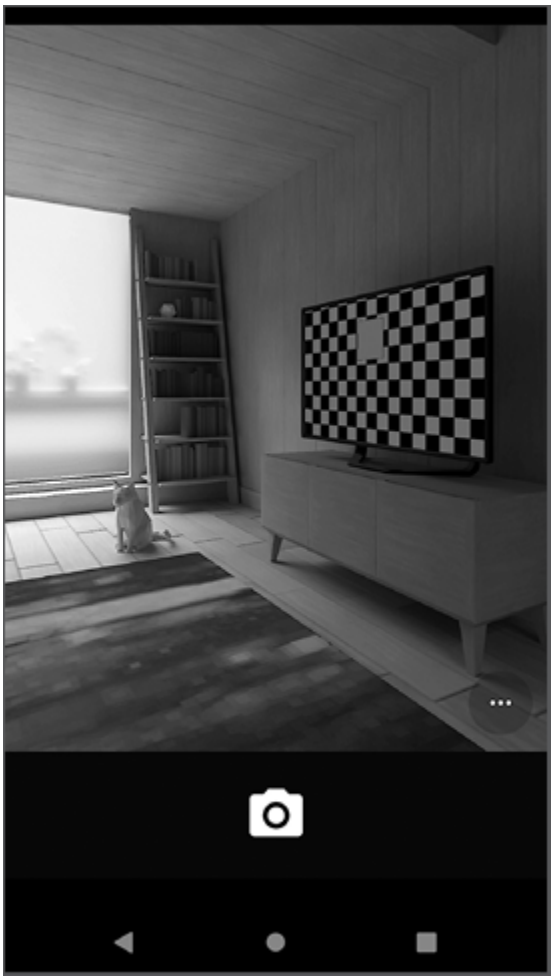


Рис. 16.2. Приложение для работы с камерой

Масштабирование и отображение растровых изображений

После всего что было сделано, приложение успешно делает снимки, которые сохраняются в файловой системе для дальнейшего использования.

Следующий шаг — поиск файла с изображением, его загрузка и отображение для пользователя. Для этого необходимо загрузить данные изображения в объект `Bitmap` достаточного размера. Чтобы построить объект `Bitmap` на базе файла, достаточно воспользоваться классом `BitmapFactory`:

```
val bitmap = BitmapFactory.decodeFile(photoFile.getPath())
```

Но ведь должна же быть какая-то загвоздка, верно? Иначе мы бы просто напечатали эту строку жирным шрифтом, вы бы ввели ее — и на этом все закончилось.

Да, загвоздка существует: «достаточный размер» следует понимать буквально. `Bitmap` — простой объект для хранения необработанных данных пикселов. Таким образом, даже если исходный файл был сжат, в объекте `Bitmap` никакого сжатия не будет. Итак, 24-битовое изображение с камеры на 16 мегапикселов, которое может занимать всего 5 Мб в формате JPG, в объекте `Bitmap` разрастается до 48(!) Мб.

Найти обходное решение возможно, но оно означает, что изображение придется масштабировать вручную. Для этого можно сначала просканировать файл и определить его размер, затем вычислить, насколько его нужно масштабировать для того, чтобы он поместился в заданную область, и, наконец, заново прочитать файл для создания уменьшенного объекта `Bitmap`.

Создайте файл `PictureUtils.kt` и добавьте в него функцию уровня файловой системы с именем `getScaledBitmap(String,int,int)` (листинг 16.13).

Листинг 16.13. Создание функции getScaledBitmap(...) (PictureUtils.kt)

```
fun getScaledBitmap(path: String, destWidth: Int, destHeight: Int): Bitmap {
    // Чтение размеров изображения на диске
    var options = BitmapFactory.Options()
    options.inJustDecodeBounds = true
    BitmapFactory.decodeFile(path, options)

    val srcWidth = options.outWidth.toFloat()
    val srcHeight = options.outHeight.toFloat()

    // Выясняем, на сколько нужно уменьшить
    var inSampleSize = 1
     if (srcHeight > destHeight || srcWidth > destWidth) {
            val heightScale = srcHeight / destHeight
        val widthScale = srcWidth / destWidth

          val sampleScale = if (heightScale > widthScale) {
            heightScale
        } else {
            widthScale
        }
        inSampleSize = Math.round(sampleScale)
    }

    options = BitmapFactory.Options()
    options.inSampleSize = inSampleSize
```

```
        // Чтение и создание окончательного растрового изображения
        return BitmapFactory.decodeFile(path, options)
    }
```

Ключевой параметр `inSampleSize` определяет величину «образца» для каждого пиксела исходного изображения: образец с размером 1 содержит один горизонтальный пиксел для каждого горизонтального пиксела исходного файла, а образец с размером 2 содержит один горизонтальный пиксел для каждых двух горизонтальных пикселов исходного файла. Таким образом, если значение `inSampleSize` равно 2, количество пикселов в изображении составляет четверть от количества пикселов оригинала.

И последняя неприятная новость: при запуске фрагмента вы еще не знаете величину `PhotoView`. До обработки макета никаких экранных размеров не существует. Первый проход этой обработки происходит после выполнения `onCreate(...)`, `onStart()` и `onResume()`, поэтому `PhotoView` и не знает своих размеров.

У проблемы есть два решения: либо подождать, пока будет сделан первый проход, либо воспользоваться консервативной оценкой. Второй способ менее эффективен, но более прямолинеен. Напишите еще одну функцию уровня файловой системы `getScaledBitmap(String,Activity)` для масштабирования `Bitmap` под размер конкретной `Activity`.

Листинг 16.14. Функция масштабирования с консервативной оценкой (PictureUtils.kt)

```
fun getScaledBitmap(path: String, activity: Activity): Bitmap {
```

```
    val size = Point()
    activity.windowManager.defaultDisplay.getSi
ze(size)

        return getScaledBitmap(path, size.x,
size.y)
}

fun getScaledBitmap(path: String, destWidth:
Int, destHeight: Int): Bitmap {
    ...
}
```

Функция проверяет размер экрана и уменьшает изображение до этого размера. Виджет `ImageView`, в который загружается изображение, всегда меньше размера экрана, так что эта оценка весьма консервативна.

Чтобы загрузить объект `Bitmap` в `ImageView`, добавьте в `CrimeFragment` функцию для обновления `photoView`.

Листинг 16.15. Обновление photoView (CrimeFragment.kt)

```
class CrimeFragment : Fragment(),
DatePickerFragment.Callbacks {
    ...
    private fun updateUI() {
        ...
    }

    private fun updatePhotoView() {
        if (photoFile.exists()) {
```

```
                                    val bitmap =
getScaledBitmap(photoFile.path,
requireActivity())
                photoView.setImageBitmap(bitmap)
            } else {
                photoView.setImageDrawable(null)
            }
        }

         override fun onActivityResult(requestCode:
Int, resultCode: Int, data: Intent?) {
            ...
        }
        ...
}
```

Затем вызовите эту функцию из updateUI() и onActivityResult(...).

Листинг 16.16. Вызов updatePhotoView() (CrimeFragment.kt)

```
class CrimeFragment : Fragment(),
DatePickerFragment.Callbacks {
    ...
    private fun updateUI() {
        ...
        if (crime.suspect.isNotEmpty()) {
            suspectButton.text = crime.suspect
        }
        updatePhotoView()
    }
    ...
```

```
     override fun onActivityResult(requestCode:
Int, resultCode: Int, data: Intent?) {
        when {
            resultCode != Activity.RESULT_OK ->
return

              requestCode == REQUEST_CONTACT &&
data != null -> {
                ...
            }

            requestCode == REQUEST_PHOTO -> {
                updatePhotoView()
            }
        }
    }
    ...
}
```

Теперь, когда приложение камеры завершило запись в файл, можно отозвать разрешение и снова перекрыть доступ к файлу. Выполните это с помощью функции `onActivityResult(...)`, чтобы отозвать разрешение в случае действительного результата, а также с помощью `onDetach()`, чтобы скрыть вероятность неверного отклика.

Листинг 16.17. Отзыв разрешений URI (CrimeFragment.kt)

```
class CrimeFragment : Fragment(),
DatePickerFragment.Callbacks {
    ...
    override fun onStop() {
```

```
            ...
        }

        override fun onDetach() {
            super.onDetach()
            requireActivity().revokeUriPermission(p
hotoUri,
                Intent.FLAG_GRANT_WRITE_URI_PERMISS
ION)
        }

         override fun onActivityResult(requestCode:
Int, resultCode: Int, data: Intent?) {
            when {
                ...

                requestCode == REQUEST_PHOTO -> {
                    requireActivity().revokeUriPerm
ission(photoUri,
                        Intent.FLAG_GRANT_WRITE_URI
_PERMISSION)
                    updatePhotoView()
                }
            }
        }
        ...
    }
```

Снова запустите приложение CriminalIntent. Откройте экран с подробностями преступления и с помощью кнопки камеры сделайте снимок. Вы должны увидеть свое изображение на панели эскизов (рис. 16.3).

Объявление функциональности

Наша реализация камеры сейчас отлично работает. Остается решить еще одну задачу: сообщить о ней потенциальным пользователям. Когда приложение использует некоторое оборудование (например, камеру или NFC) или любой другой элемент, который может отличаться от устройства к устройству, настоятельно рекомендуется сообщить о нем Android. Это позволит другим приложениям (например, магазину Google Play) заблокировать установку приложения, если в нем используются возможности, не поддерживаемые вашим устройством.

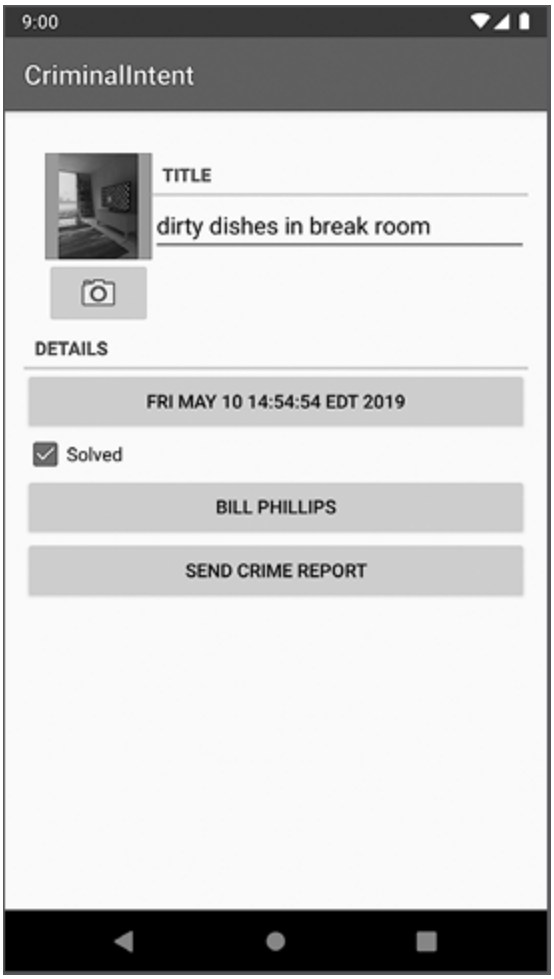


Рис. 16.3. Эскиз в окне подробностей преступления

Чтобы объявить, что в приложении используется камера, включите тег <uses-feature> в файл AndroidManifest.xml (листинг 16.18).

Листинг 16.18. Добавление тега <uses-feature> (manifests/AndroidManifest.xml)

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.criminalintent" >

    <uses-feature
android:name="android.hardware.camera"
                  android:required="false"/>
    ...
</manifest>
```

В этом примере в тег добавляется необязательный атрибут android:required. Почему? По умолчанию объявление об использовании некоторой возможности означает, что без нее приложение может работать некорректно. К CriminalIntent это не относится. Мы вызываем resolveActivity(...), чтобы проверить наличие приложения камеры, после чего корректно блокируем кнопку, если приложение не найдено.

Атрибут android:required="false" корректно обрабатывает эту ситуацию. Мы сообщаем Android, что приложение может нормально работать без камеры, но некоторые части приложения окажутся недоступными.

Упражнение. Вывод увеличенного изображения

Конечно, вы видите уменьшенное изображение, но вряд ли вам удастся рассмотреть его во всех подробностях. Создайте новый фрагмент `DialogFragment`, в котором отображается увеличенная версия фотографии места преступления. Когда пользователь нажимает на миниатюру, на экране должен появиться `DialogFragment` с увеличенным изображением.

Упражнение. Эффективная загрузка миниатюры

В этой главе нам пришлось использовать довольно грубую оценку размера, до которого должно быть уменьшено изображение. Такое решение не идеально, но оно работает и быстро реализуется.

С существующими API можно использовать `ViewTreeObserver` — объект, который можно получить от любого представления в иерархии `Activity`:

```
val observer = imageView.viewTreeObserver
```

Вы можете зарегистрировать для `ViewTreeObserver` разнообразных слушателей, включая `OnGlobalLayoutListener`. Этот слушатель инициирует событие каждый раз, когда происходит проход обработки макета.

Измените свой код так, чтобы он использовал размеры `photoView`, когда они действительны, и ожидал прохода обработки макета перед первым вызовом `updatePhotoView()`.

17. Локализация

Предвидя бешеную популярность приложения CriminalIntent, вы решаете сделать его доступным для большей аудитории. Первым шагом должна стать локализация всего текста, видимого пользователю, чтобы пользователь мог общаться с приложением как на английском, так и на испанском языке.

Локализацией называется процесс формирования ресурсов приложения в зависимости от языка, выбранного пользователем. В этой главе мы создадим испаноязычную версию файла `res/values/strings.xml`. Когда на устройстве выбирается испанский язык, Android на стадии выполнения автоматически находит и использует испанские строки (рис. 17.1).

Рис. 17.1. IntentoCriminal

Локализация ресурсов

Языковые настройки являются частью конфигурации устройства (см. раздел «Конфигурации устройств и жизненный цикл ресурса» в главе 3). Android предоставляет квалификаторы для разных языков по аналогии с ориентацией, размером экрана и другими аспектами конфигурации. Процесс локализации становится достаточно тривиальным: вы создаете подкаталоги ресурсов, снабжаете их квалификаторами нужных языков и размещаете в них альтернативные ресурсы. Система ресурсов Android сделает все остальное.

В проекте CriminalIntent создайте файл ресурсов со значениями. На панели **Project** щелкните правой кнопкой мыши по папке `res/values/` и выберите команду **New⇒Valuesresourcefile** в контекстном меню. В поле **Filename** введите имя **strings**. Оставьте в поле **Sourceset** значение **main**. Убедитесь в том, что в поле **Directoryname** выбрано значение **values**.

Выберите в списке **Availablequalifiers** вариант **Locale** и нажмите кнопку **>>**, чтобы переместить **Locale** в группу **Chosenqualiifiers**. Выберите в списке **Language** значение **es:Spanish**. В списке **SpecificRegionOnly** автоматически выбирается вариант **Anyregion** — именно это вам и нужно, оставьте этот вариант.

Окно **NewResourceFile** показано на рис. 17.2.

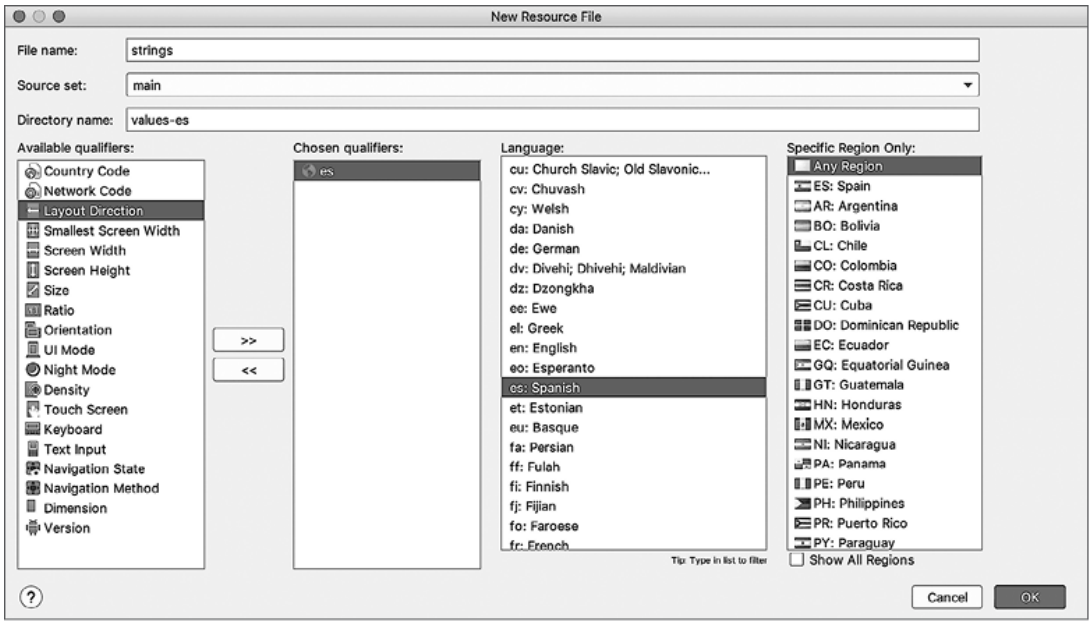


Рис. 17.2. Добавление файла строковых ресурсов с квалификаторами

Android Studio автоматически заносит в поле **Directoryname** значение **values-es**. Квалификаторы языков берутся из стандарта ISO 639-1 и состоят из двух символов. Для испанского языка используется квалификатор `-es`.

Нажмите кнопку **OK**. Новый файл `strings.xml` появится в папке `res/values`, за его именем следует суффикс `(es)`. Строковые файлы группируются в режиме **Android** панели **Project** (рис. 17.3).

Рис. 17.3. Новый файл strings.xml в режиме Android

Но если вы просмотрите структуру каталогов, вы увидите, что проект теперь содержит дополнительный каталог `res/values-es`. Сгенерированный файл `strings.xml` находится в этом новом каталоге (рис. 17.4).

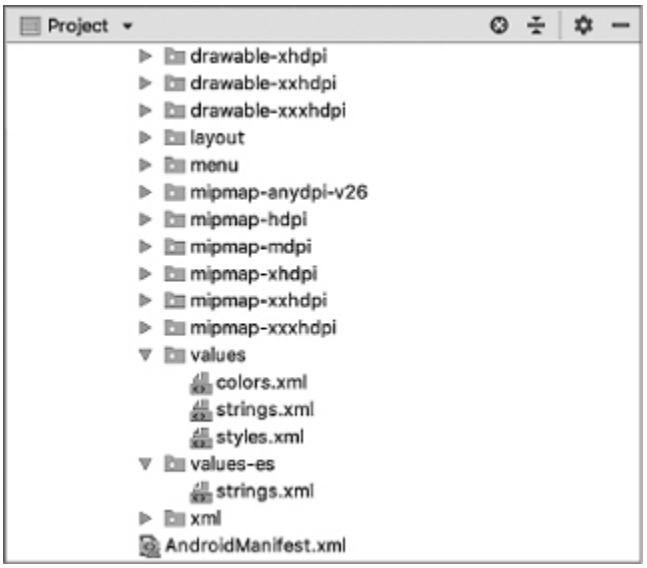


Рис. 17.4. Новый файл strings.xml в режиме Project

А теперь пора заняться тем, чтобы английский текст как по волшебству заменялся испанским. Добавьте испанские версии всех ваших строк в файл `res/values-es/strings.xml`. (Если вы не хотите вводить строки вручную, скопируйте содержимое из файла решения, который можно скачать по ссылке **www.bignerdranch.com/solutions/AndroidProgramming4e.zip**.)

Листинг 17.1. Добавление строковых ресурсов для испанского языка (res/values-es/strings.xml)

```
<resources>
                                        <string
name="app_name">IntentoCriminal</string>
     <string name="crime_title_hint">Introduzca
un título para el crimen.</string>
                                        <string
name="crime_title_label">Título</string>
                                        <string
name="crime_details_label">Detalles</string>
                                        <string
name="crime_solved_label">Solucionado</string>
              <string   name="new_crime">Crimen
Nuevo</string>
      <string name="crime_suspect_text">Elegir
Sospechoso</string>
     <string name="crime_report_text">Enviar el
Informe del Crimen</string>
    <string name="crime_report">%1$s!
           El crimen fue descubierto el %2$s.
%3$s, y %4$s
    </string>
```

```
    <string name="crime_report_solved">El caso
está resuelto</string>
      <string name="crime_report_unsolved">El
caso no está resuelto</string>
     <string name="crime_report_no_suspect">no
hay sospechoso.</string>
     <string name="crime_report_suspect">el/la
sospechoso/a es %s.</string>
                                        <string
name="crime_report_subject">IntentoCriminal
Informe del Crimen</string>
       <string name="send_report">Enviar el
informe del crimen a través de</string>
</resources>
```

Вот и все, что необходимо, чтобы предоставить приложению локализованные строковые ресурсы. Чтобы убедиться в этом, выберите на своем устройстве испанский язык. (Откройте приложение **Settings** и найдите раздел выбора языка. В зависимости от версии Android этот раздел может называться **Languageandinput**, **LanguageandKeyboard** или как-нибудь в этом роде.)

Когда на экране появится список языковых настроек, выберите вариант **Español**. Выбор региона (**España** или **EstadosUnidos**) роли не играет, потому что квалификатор `-es` подходит для обоих вариантов. (Обратите внимание: в новых версиях Android пользователь может выбрать несколько языков и назначить приоритеты. Если вы используете новое устройство, убедитесь в том, что вариант **Español** стоит на первом месте в списке языковых настроек.)

Запустите CriminalIntent и насладитесь видом локализованного приложения. Потом снова включите на устройстве английский язык. Найдите в лаунчере приложение

настроек — **Ajustes** или **Configuración** — и раздел, в имени которого присутствует слово **Idioma** (язык).

Ресурсы по умолчанию

Для английского языка используется квалификатор `-en`. В пылу локализационного рвения вам, возможно, захочется переименовать существующий каталог значений в `values-en`. Делать так не стоит, но представьте на минуту, что это было сделано: это гипотетическое обновление означает, что ресурсы вашего приложения для английского языка хранятся в файле `strings.xml` в каталоге `values-en`, а ресурсы для испанского языка — в файле `strings.xml` в каталоге `values-es`.

Только что обновленное приложение нормально собирается. Оно также нормально работает на устройствах, на которых выбран испанский или английский язык. Но что произойдет, если на устройстве будет выбран итальянский язык? Ничего хорошего. Совсем ничего. Во время выполнения Android не найдет строковые ресурсы для текущей конфигурации. Приложение вылетит с исключением `Resources.NotFoundException`.

У Android Studio есть способ спасти вас от этой участи. Инструмент Android Asset Packaging Tool (AAPT) выполняет множество проверок во время упаковки ваших ресурсов. Если AAPT обнаружит, что вы используете ресурсы, не включенные в файлы ресурсов по умолчанию, он выдаст ошибку во время компиляции:

```
Android resource linking failed

warn: removing resource
com.bignerdranch.android.criminalintent:string/
crime_title_label
```

```
without required default value.

AAPT: error: resource string/crime_title_label
(aka
com.bignerdranch.android.criminalintent:string/
crime_title_label)
not found.

error: failed linking file resources.
```

Мораль: предоставляйте *версию по умолчанию* для каждого из своих ресурсов. Ресурсы, находящиеся в каталогах ресурсов без квалификаторов, используются по умолчанию, если ресурс, соответствующий текущей конфигурации устройства, не найден. Если Android ищет ресурс, но не может найти версию, соответствующую текущей конфигурации устройства, работа приложения будет нарушена.

Отличия в плотности пикселов

В правилах назначения ресурсов по умолчанию существует исключение: экранная плотность пикселов. Как вы уже видели, каталоги `drawable` проекта обычно уточняются квалификаторами плотности экрана `-mdpi`, `-xxhdpi` и т.д. Однако при выборе используемого ресурса графического объекта решение Android не сводится к простому подбору экранной плотности устройства или использованию каталога без квалификатора при отсутствии совпадения.

Выбор определяется сочетанием размера экрана и плотности пикселов, и Android может выбрать графический ресурс из каталога с квалификатором более низкой или высокой плотности с последующим масштабированием. Более подробная информация приведена в документации по адресу

developer.android.com/guide/practices/screens_support.html, а пока просто учтите, что размещать ресурсы графических объектов по умолчанию в `res/drawable/` не обязательно.

Проверка покрытия локализации в Translations Editor

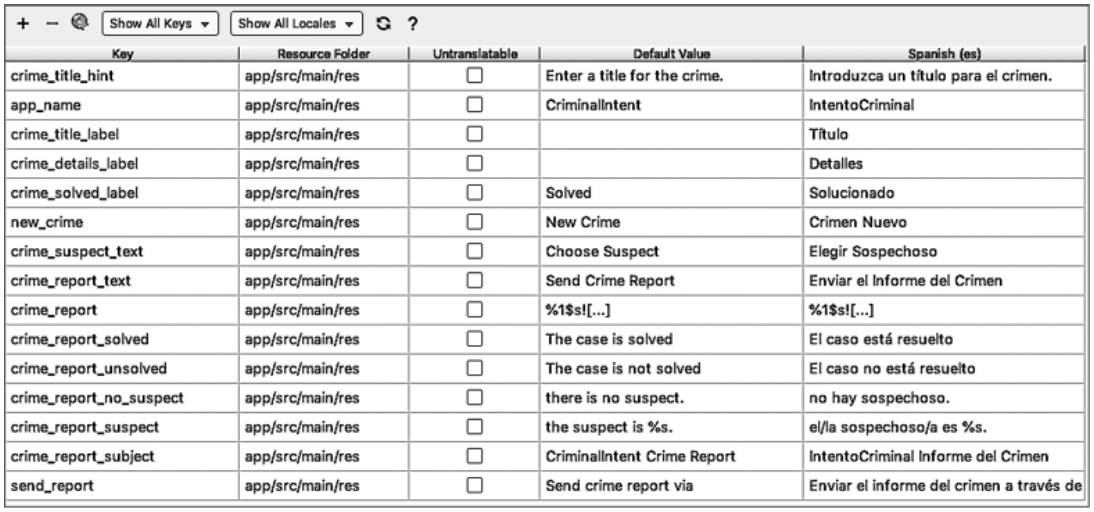
С ростом числа поддерживаемых языков становится все труднее следить за тем, чтобы в локализации были представлены переводы всех строк для всех языков. К счастью, в Android Studio существует удобная программа Translations Editor для централизованной работы со всеми локализациями. Перед началом работы откройте файл `strings.xml` и закомментируйте строки `crime_title_label` и `crime_details_label` (листинг 17.2).

Листинг 17.2. Закомментированные строки (res/values/strings.xml)

```
<resources>
                                        <string
name="app_name">CriminalIntent</string>
       <string name="crime_title_hint">Enter a
title for the crime.</string>
                                    <!--<string
name="crime_title_label">Title</string>-->
                                    <!--<string
name="crime_details_label">Details</string>-->
                                        <string
name="crime_solved_label">Solved</string>
    ...
</resources>
```

Чтобы запустить Translations Editor, щелкните правой кнопкой мыши по одному из файлов `strings.xml` на панели **Project** и выберите команду **OpenTranslationsEditor**. Translations Editor выводит все строки приложения и статус перевода для каждого языка, для которого ваше приложение в настоящее время предоставляет строковые значения с квалификаторами. Так как поля `crime_title_label` и `crime_report_subject` закомментированы, их имена выделяются красным цветом (рис. 17.5).

Вы получили простой список ресурсов для добавления в ваш проект. Найдите недостающие ресурсы в любой конфигурации и добавьте их в соответствующий строковый файл.



Key	Resource Folder	Untranslatable	Default Value	Spanish (es)
crime_title_hint	app/src/main/res	☐	Enter a title for the crime.	Introduzca un título para el crimen.
app_name	app/src/main/res	☐	CriminalIntent	IntentoCriminal
crime_title_label	app/src/main/res	☐		Título
crime_details_label	app/src/main/res	☐		Detalles
crime_solved_label	app/src/main/res	☐	Solved	Solucionado
new_crime	app/src/main/res	☐	New Crime	Crimen Nuevo
crime_suspect_text	app/src/main/res	☐	Choose Suspect	Elegir Sospechoso
crime_report_text	app/src/main/res	☐	Send Crime Report	Enviar el Informe del Crimen
crime_report	app/src/main/res	☐	%1$s![...]	%1$s![...]
crime_report_solved	app/src/main/res	☐	The case is solved	El caso está resuelto
crime_report_unsolved	app/src/main/res	☐	The case is not solved	El caso no está resuelto
crime_report_no_suspect	app/src/main/res	☐	there is no suspect.	no hay sospechoso.
crime_report_suspect	app/src/main/res	☐	the suspect is %s.	el/la sospechoso/a es %s.
crime_report_subject	app/src/main/res	☐	CriminalIntent Crime Report	IntentoCriminal Informe del Crimen
send_report	app/src/main/res	☐	Send crime report via	Enviar el informe del crimen a través de

Рис. 17.5. Проверка покрытия локализации в Translation Editor

Вы можете добавлять строки прямо в Translation Editor, в вашем случае вам нужно только раскомментировать строки `crime_title` и `crime_details_label`. Сделайте это, перед тем как двигаться дальше.

Региональная локализация

Каталогу ресурсов также можно назначить квалификатор «язык-регион», который позволяет еще точнее определять ресурсы. Например, квалификатор для испанского языка и имеет вид `-es-rES`, где `r` — признак квалификатора региона, а `ES` — код Испании согласно стандарту ISO 3166-1-alpha-2. Конфигурационные квалификаторы не учитывают регистр символов, но в них рекомендуется соблюдать схему, принятую в Android: код языка записывается в нижнем регистре, а код региона записывается символами верхнего регистра с префиксом r (в нижнем регистре).

Учтите, что квалификатор «язык-регион» (такой как `-es-rES`) может выглядеть как объединение двух разных конфигурационных квалификаторов, но в действительности квалификатор только один. Регион сам по себе не является действительным квалификатором.

Ресурс с квалификаторами по локализации и по региону получает две возможности подстройки под локализацию пользователя. Точное совпадение происходит тогда, когда и язык, и регион совпадают с локалью пользователя. Если точное совпадение не найдено, система удалит классификатор региона и будет искать точное совпадение только для языка.

Выбор ресурсов при отсутствии точного совпадения может выполяться по-разному, в зависимости от версии устройства для Android. На рис. 17.6 показана стратегия локального разрешения ресурсов до и после Android Nougat.

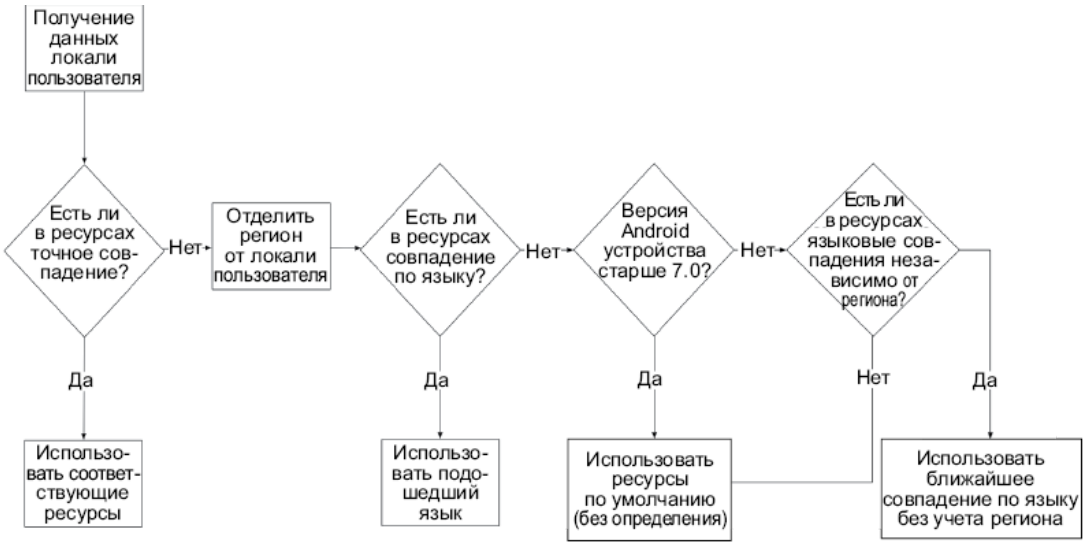


Рис. 17.6. Правила разрешения локализованных ресурсов

На устройствах с версиями Android, предшествующими Nougat, при отсутствии совпадения для языка используются ресурсы по умолчанию (без квалификаторов).

В Nougat поддержка локальных контекстов была усовершенствована: их количество увеличилось, а пользователь получил возможность выбрать несколько локальных контекстов в конфигурации устройства. Система также использует более разумную стратегию разрешения ресурсов, чтобы правильный язык использовался как можно чаще, даже если на устройстве не совпадает регион или язык без квалификатора. Если на устройстве с Nougat точное совпадение не найдено, и не найдено совпадение только для языка, система ищет ресурс с тем же языком, но по другим регионам, и использует лучшее совпадение для ресурсов, удовлетворяющих этим критериям.

Рассмотрим пример. Допустим, на устройстве выбран испанский язык и регион Чили (рис. 17.7). Приложение на устройстве содержит файлы `strings.xml` с ресурсами на испанском языке для Испании и Мексики (в каталогах `values-`

es-rES и values-es-rMX). Каталог values для ресурсов по умолчанию содержит английский файл strings.xml.

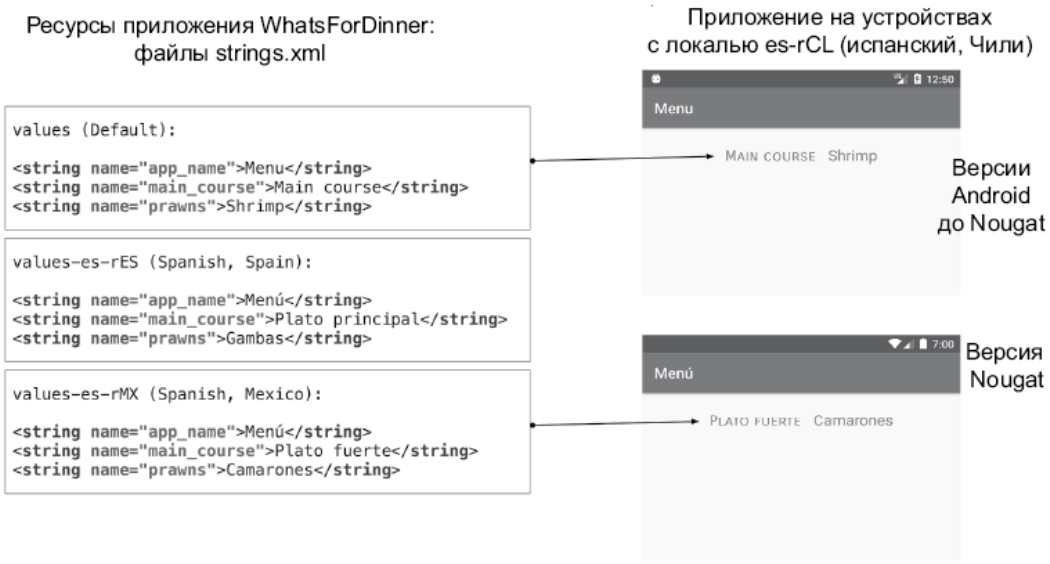


Рис. 17.7. Пример разрешения локализованных ресурсов (до и после Nougat)

Если на вашем устройстве работает Android версии до Nougat, вы увидите англоязычный текст из каталога values. Но если на устройстве установлена версия Nougat, результат выглядит более логично: вы увидите ресурсы из файла values-es-rMX/strings.xml — то есть испанский текст, хотя и не адаптированный для Чили.

Пример немного искусственный, но он подчеркивает один важный момент: строки следует предоставлять по возможности в общем контексте, используя каталоги с квалификаторами языка и региона только по необходимости. Вместо того чтобы сопровождать все испаноязычные строки в трех каталогах, уточненных квалификаторами регионов, в этом приложении лучше было бы хранить строки в каталоге с квалификатором языка values-es и определять строки с квалификаторами регионов только для слов и выражений, различающихся в региональных диалектах.

Такое решение не только упрощает сопровождение строк для программиста, но и упрощает разрешение ресурсов на устройствах с разными версиями Android — как с версией Nougat, так и до нее. Обратите внимание, что эта рекомендация распространяется на все типы альтернативных ресурсов в каталогах `values`: общие ресурсы должны находиться в более общих каталогах, и это должны быть только те ресурсы, которые адаптированы в каталогах с более конкретной квалификацией.

Конфигурационные квалификаторы

Вы уже видели и использовали конфигурационные квалификаторы для предоставления альтернативных ресурсов: язык (например, `values-es`), ориентацию экрана (например, `layout-land`), плотность пикселов (например, `drawable-mdpi`).

Android поддерживает конфигурационные квалификаторы для выбора ресурсов в зависимости от следующих аспектов конфигурации устройства:

1. Код страны для мобильной связи (MCC), за которым может следовать код сети мобильной связи (MNC).
2. Код языка, за которым может следовать код региона.
3. Направление макета.
4. Минимальная ширина.
5. Доступная ширина.
6. Доступная высота.

7. Размер экрана.

8. Пропорции экрана.

9. Закругленный экран (API уровня 23 и выше).

10. Ориентация экрана.

11. Широкая цветовая гамма.

12. Высокий динамический диапазон.

13. Режим пользовательского интерфейса.

14. Ночной режим.

15. Плотность экрана (dpi).

16. Тип сенсорного экрана.

17. Доступность клавиатуры.

18. Основной метод ввода текста.

19. Доступность клавиш перемещения.

20. Основной несенсорный метод перемещения.

21. Уровень API.

Описания этих характеристик и примеры конкретных конфигурационных квалификаторов можно найти по адресу **developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources**.

Не все квалификаторы поддерживаются ранними версиями Android. К счастью, система неявно добавляет квалификатор версии платформы к квалификаторам, добавленным после Android 1.0. Таким образом, если вы, например, используете квалификатор `round`, Android автоматически добавляет квалификатор `v23`, потому что квалификатор круглого экрана появился в API уровня 23. Это означает, что при добавлении ресурсов с квалификаторами новых устройств вам не придется беспокоиться о возможных проблемах со старыми устройствами.

Приоритеты альтернативных ресурсов

При таком количестве конфигурационных квалификаторов для определения ресурсов могут возникнуть ситуации, при которых к конфигурации устройства могут подходить сразу несколько альтернативных ресурсов. В таких ситуациях квалификаторам назначаются приоритеты в соответствии с их порядком в приведенном выше списке.

Чтобы понять, как работает система приоритетов, добавим в CriminalIntent альтернативный ресурс — более длинную англоязычную версию строкового ресурса `crime_title_hint`, которая должна отображаться в том случае, если ширина экрана в текущей конфигурации не менее 600dp. Ресурс `crime_title_hint` отображается в текстовом поле описания до того, как пользователь введет какой-либо текст. Если приложение CriminalIntent выполняется на экране с шириной не менее 600dp (например, на планшете или в альбомном режиме на устройстве с меньшим экраном), с таким изменением в поле описания будет выводиться более содержательная и осмысленная подсказка.

Создайте новый файл ресурсов со строковыми значениями, назвав его `strings`. Следуйте инструкциям из раздела «Локализация ресурсов» из этой главы, чтобы создать файл ресурсов, но в этот раз выберите пункт **ScreenWidth** в списке **Availablequalifiers** и нажмите кнопку **>>**, чтобы переместить **ScreenWidth** в раздел **ChosenQualifiers**. В появившемся поле **ScreenWidth** введите **600**. Будет автоматически выбрано имя папки `values-w600dp`. Значение `-w600dp` подойдет любому устройству, у которого текущая доступная ширина экрана составляет `600dp` или более, что означает, что устройство подойдет под этот параметр в альбомном режиме, но не в книжном. (Чтобы узнать больше о классификаторах размера экрана, прочтите раздел «Для любознательных: подробнее об определении размеров экрана устройства» в конце этой главы.) Диалоговое окно должно выглядеть так, как показано на рис. 17.8.

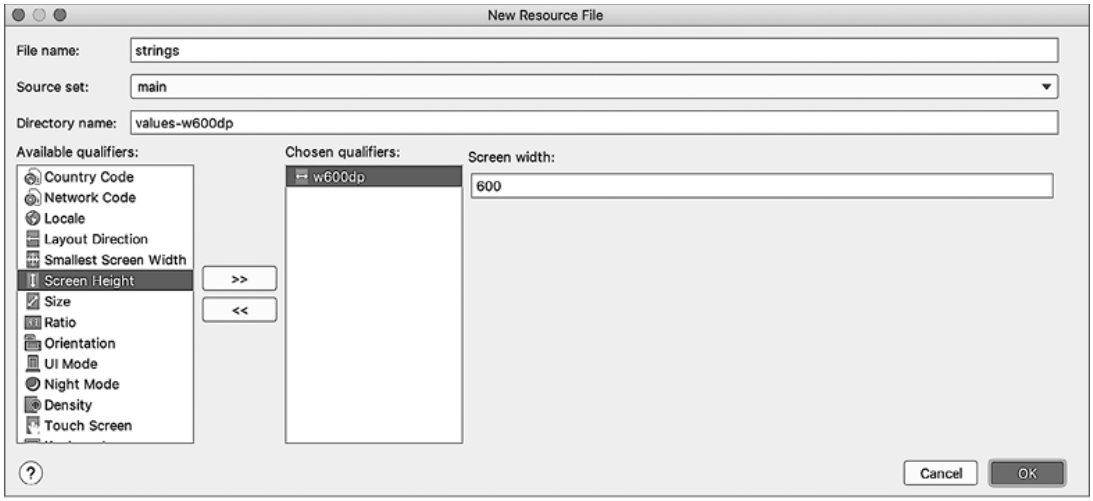


Рис. 17.8. Добавление строк для экрана большей ширины

Добавьте длинную строку `crime_title_hint` в файл `res/values-w600dp/strings.xml`.

Листинг 17.3. Создание альтернативных строковых ресурсов для широкого экрана (res/values-w600dp/strings.xml)

```
<resources>
    <string name="crime_title_hint">
        Enter a meaningful, memorable title for
the crime.
    </string>
</resources>
```

Единственный строковый ресурс, который используется только на широких экранах, — `crime_title_hint`. Это единственная строка, хранящаяся в каталоге `values-w600dp`. Альтернативы для строковых ресурсов (и других ресурсов-значений) определяются на уровне отдельных строк, поэтому дублировать одинаковые строки *не нужно*. Дублирование только создаст проблемы с будущим сопровождением проекта.

Итак, теперь у вас имеются три версии `crime_title_hint`: версия по умолчанию в файле `res/values/strings.xml`, испанская альтернативная версия в файле `res/values-es/strings.xml`, и альтернативная версия для широкого экрана в файле `res/values-w600dp/strings.xml`.

Пока на устройстве остается выбранным испанский язык, запустите приложение CriminalIntent и поверните устройство в альбомную ориентацию (рис. 17.9). Альтернативная версия для испанского языка обладает более высоким приоритетом, поэтому вместо версии из `res/values-w600dp/strings.xml` вы увидите строку из `res/values-es/strings.xml`.

Рис. 17.9. В Android язык обладает более высоким приоритетом, чем доступная ширина экрана

Верните на устройстве английский язык. Снова запустите приложение и убедитесь в том, что на экране, как и ожидалось, выводится альтернативная строка для широкого экрана.

Множественные квалификаторы

Вероятно, вы заметили, что диалоговое окно **NewResourceFile** содержит много возможных квалификаторов. Каталогу ресурсов можно назначить более одного квалификатора. В таком случае квалификаторы перечисляются в порядке приоритетов. Таким образом, `values-es-w600dp` является действительным именем каталога, а `values-w600dp-es` — нет. (Если вы используете диалоговое окно **NewResourceFile**, оно автоматически настраивает имя каталога.)

Создайте каталог для строки на испанском языке, предназначенной для широкого экрана. Каталогом должен называться `values-es-w600dp`, и в нем должен присутствовать файл `strings.xml`. Добавьте строковый ресурс

`crime_title_hint` в файл `values-es-w600dp/strings.xml` (листинг 17.4).

Листинг 17.4. Создание строкового ресурса для испанского языка на широком экране (res/values-es-w600dp/strings.xml)

```
<resources>
    <string name="crime_title_hint">
        Introduzca un título significativo y
memorable para el crimen.
    </string>
</resources>
```

Выберите испанский язык, запустите приложение CriminalIntent и убедитесь в том, что новый альтернативный ресурс отображается в положенном месте (рис. 17.10).

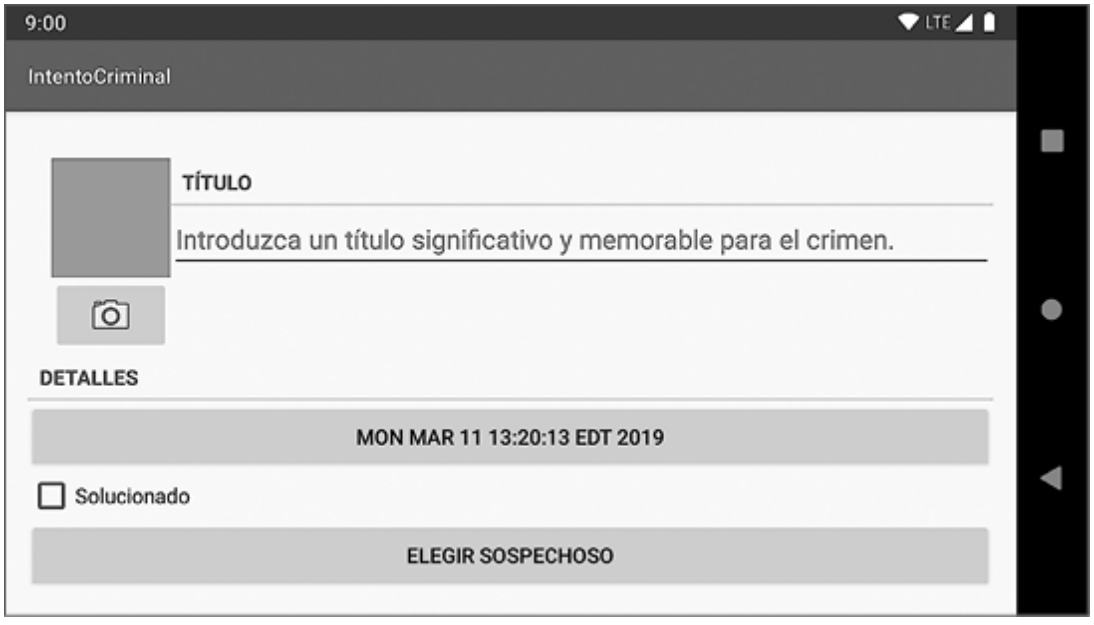


Рис. 17.10. Испаноязычный строковый ресурс для широкого экрана

Поиск наиболее подходящих ресурсов

Каким образом Android определяет, какая версия `crime_title_hint` должна отображаться при каждом запуске? Возьмем четыре альтернативных варианта строкового ресурса `crime_title_hint` на устройстве Pixel 2 с испанским языком и доступной шириной экрана в альбомной ориентации более 600dp:

Конфигурация устройства	Каталог для crime_title_hint
• Язык: `es` (испанский)	• `values`
• Доступная высота: `411dp`	• `values-es`
• Доступная ширина: `731dp`	• `values-es-w600dp`
• и т.д.	• `values-w600dp`

Исключение несовместимых каталогов

Поиск наиболее подходящего ресурса Android начинает с исключения всех каталогов ресурсов, несовместимых с текущей конфигурацией.

Все четыре варианта совместимы с текущей конфигурацией. (Если повернуть устройство в книжную ориентацию, то доступная ширина станет равной 411dp; в этом случае каталоги `values-w600dp/` и `values-es-w600dp/` станут несовместимыми и будут исключены.)

Перебор по таблице приоритетов

После исключения несовместимых каталогов ресурсов Android начинает перебирать таблицу приоритетов, приведенную в разделе «Конфигурационные квалификаторы» ранее в этой главе, начиная с самого приоритетного квалификатора: MCC. Если существует каталог ресурсов с квалификатором MCC, то все каталоги ресурсов, не имеющие квалификатора MCC, будут исключены. Если после этого осталось более одного

подходящего каталога, то Android рассматривает следующий по приоритету квалификатор. Перебор продолжается до тех пор, пока не останется только один каталог.

В нашем примере каталогов с квалификатором MCC нет, поэтому ни один каталог не исключается, и Android переходит по списку к квалификатору языка. Квалификаторы языка присутствуют в двух каталогах (`values-es` и `values-es-w600dp`). В одном каталоге (`values-w600dp`) этого квалификатора нет, поэтому он исключается.

(Однако, как мы уже упоминали ранее в этой главе, каталог без квалификатора `values` служит ресурсом по умолчанию, или откатом. Так что, хотя это пока исключено из-за отсутствия квалификатора языка, `values` все равно могут оказаться лучшим совпадением, если другие каталоги дают несоответствие в одном или нескольких квалификаторах нижнего порядка.)

Конфигурация устройства	Каталог для crime_title_hint
• Язык: `es` (испанский)	• values (без привязки к языку)
• Доступная высота: `411dp`	• `values-es`
• Доступная ширина: `731dp`	• `values-es-w600dp`
• и т.д.	• values-w600dp (без привязки к языку)

Кандидатов все еще слишком много, поэтому Android продолжает двигаться по списку квалификаторов. При достижении квалификатора доступной ширины Android находит один каталог с квалификатором доступной ширины и один каталог без него. Каталог `values-es` исключается, остается только `values-es-w600dp`:

Конфигурация устройства	Каталог для crime_title_hint
• Язык: `es` (испанский)	• values (без привязки к ширине или языку)
• Доступная высота: `411dp`	• values-es (без привязки к ширине)

• Доступная ширина: 731dp	• values-es-w600dp (лучшее совпадение)
• и т.д.	• values-w600dp (без привязки к языку)

Итак, Android использует ресурс из каталога values-es-w600dp.

Тестирование альтернативных ресурсов

Приложение следует протестировать на разных конфигурациях устройств, чтобы увидеть, как ваши макеты и другие ресурсы смотрятся в этих конфигурациях. Для тестирования могут использоваться как физические, так и виртуальные устройства. Также можно воспользоваться графическим конструктором.

В графическом конструкторе предусмотрено несколько способов предварительного просмотра макета в разных конфигурациях: для разных размеров экрана, типов устройств, уровней API, языков и т.д.

Чтобы ознакомиться с вариантами предварительного просмотра, откройте файл res/layout/fragment_crime.xml в графическом конструкторе и опробуйте элементы управления, изображенные на рис. 17.11.

Рис. 17.11. Предварительный просмотр в графическом конструкторе

Чтобы убедиться в том, что вы включили все необходимые ресурсы по умолчанию, включите в конфигурации устройства язык, для которого не определены локализованные ресурсы. Запустите приложение и поработайте с ним. Откройте все представления, поверните устройство.

Прежде чем переходить к следующей главе, не забудьте вернуть на устройстве английский язык.

Поздравляем! Теперь приложение CriminalIntent стало доступным как для англоязычных, так и для испаноязычных пользователей. Преступления регистрируются. Дела раскрываются. И все это с максимумом удобств — пользователь общается с приложением на родном языке. Для поддержки новых языков достаточно добавить новые файлы со строками в каталог с соответствующими квалификаторами.

Для любознательных: подробнее об определении размеров экрана устройства

У Android есть три классификатора, которые позволяют проверить размеры устройства. Эти новые классификаторы приведены в таблице 17.1.

Допустим, вы хотели задать макет, который будет использоваться только в том случае, если дисплей будет иметь ширину не менее 300dp. В этом случае вы можете использовать доступный квалификатор ширины и поместить ваш файл макета в `res/layout-w300dp` (буква `w` означает width — ширина). То же самое можно сделать и для высоты, используя `h` (height — высота).

Таблица 17.1. Классификаторы размеров дискретного экрана

Формат классификатора	Описание
wXXXdp	Доступная ширина: больше и равная XXX dp
hXXXdp	Доступная высота: больше и равная XXX dp
swXXXdp	Наименьшая ширина: ширина и высота (смотря что из этого меньше) больше или равна XXX dp

Однако высота и ширина могут меняться в зависимости от ориентации устройства. Для определения размера экрана

можно использовать sw, что означает *smallest width — наименьшая ширина*. Так вы сможете определить наименьший размер экрана. В зависимости от ориентации устройства это может быть как ширина, так и высота. Если размер экрана 1024·800, то sw равняется 800. Если размер экрана 800·1024, то sw все еще равняется 800.

Упражнение. Локализация дат

Возможно, вы заметили, что независимо от выбранного на устройстве локального контекста даты в CriminalIntent всегда выводятся в формате, принятом в США: месяц перед днем. Продолжите процесс локализации и обеспечьте форматирование дат в соответствии с конфигурацией локального контекста. Это проще, чем может показаться на первый взгляд.

Обратитесь к документации разработчика — а именно к описанию класса DateFormat во фреймворке Android. Класс DateFormat предоставляет функции форматирования даты/времени по правилам текущего локального контекста. Для управления выводом используются конфигурационные константы, встроенные в DateFormat.

18. Специальные возможности

В этой главе мы займемся специальными возможностями приложения CriminalIntent. Приложение со специальными возможностями может использоваться кем угодно, независимо от ограничений по зрению, слуху или двигательным функциям. Ограничения могут быть как постоянными, так и временными или ситуационными: например, расширенные зрачки после обследования у офтальмолога могут затруднить чтение. Жирные руки во время приготовления пищи отбивают желание прикасаться к экрану. А если вы находитесь на концерте, громкая музыка заглушает все звуки, производимые вашим устройством. Чем лучше реализованы специальные возможности приложения, тем приятнее с ним работать.

Обеспечение полного спектра специальных возможностей приложения — слишком грандиозная задача, но это не повод даже не пытаться ее решить. В этой главе мы постараемся сделать приложение CriminalIntent более доступным для пользователей с дефектами зрения. Эта область станет хорошей отправной точкой для изучения проблем специальных возможностей и проектирования приложений с учетом специальных возможностей.

Изменения этой главы не затронут внешнего вида приложения. Мы поступим иначе: упростим работу с контентом использованием *TalkBack*.

Приложение TalkBack

TalkBack — экранный диктор для Android, разработанный компанией Google. Он произносит вслух описание

содержимого экрана, которое зависит от того, что делает пользователь.

TalkBack представляет собой *сервис специальных возможностей* — специальный компонент, который может читать информацию с экрана независимо от того, в каком приложении вы работаете. Любой желающий может написать собственный сервис специальных возможностей, но TalkBack является самым популярным решением.

Чтобы использовать TalkBack, установите приложение Android Accessibility Suite из магазина Play Store на эмуляторе или устройстве (рис. 18.1). Если вы решили использовать эмулятор, нужно использовать образ эмулятора, на котором установлено приложение Play Store.

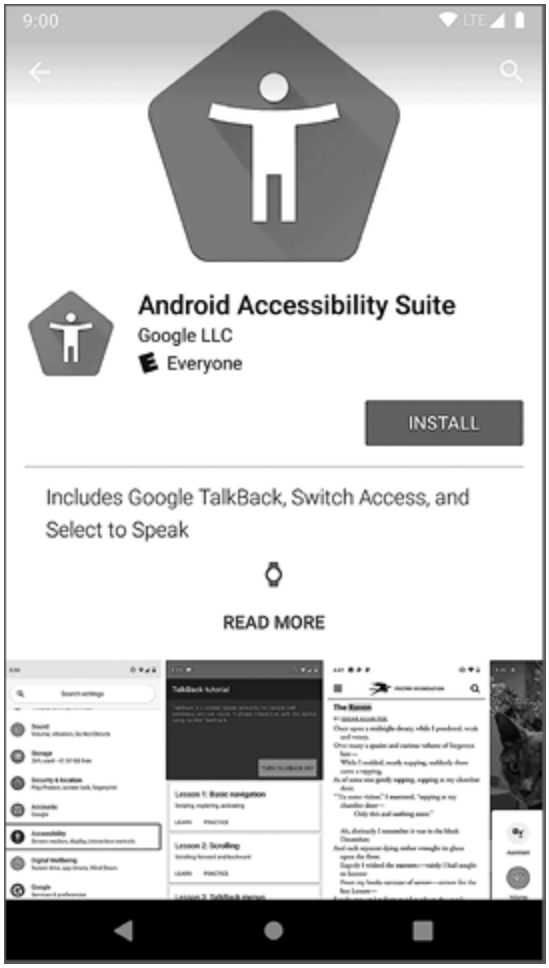


Рис. 18.1. Android Accessibility Suite

Убедитесь в том, что на устройстве не отключен звук, но, возможно, вам стоит надеть наушники, потому что с TalkBack устройство становится чрезвычайно «разговорчивым».

Чтобы включить TalkBack, запустите приложение **Settings** и откройте категорию **Accessibility**. Выберите раздел **TalkBack** под заголовком **Services**. Переведите переключатель **Useservice** в верхней части экрана в правое положение, чтобы включить TalkBack (рис. 18.2).

Android отображает диалоговое окно, в котором система запрашивает разрешение на доступ к определенной информации, например к отслеживанию действий пользователя и изменению некоторых настроек, скажем, включению функции **ExplorebyTouch** (рис. 18.3). Нажмите кнопку **OK**.

Если вы впервые используете TalkBack на устройстве, откроется обучающее руководство. Выйдите из меню при помощи кнопки «Вверх» на панели инструментов.

Изменения в работе устройства видны практически сразу. У кнопки «Вверх» появится зеленый контур (рис. 18.4), а устройство подскажет: «Перейти вверх, кнопка. Коснитесь дважды, чтобы активировать».

Хотя на устройствах Android обычно применяется термин «нажатие» (press), TalkBack использует термин «касание» (tap). Кроме того, TalkBack использует двойные касания, нечасто встречающиеся в Android.

Зеленый контур показывает, какой элемент пользовательского интерфейса *сфокусирован*. В любой момент времени *фокусспециальных возможностей* может быть только у одного элемента. При получении элементом фокуса специальных возможностей TalkBack предоставляет информацию об этом элементе.

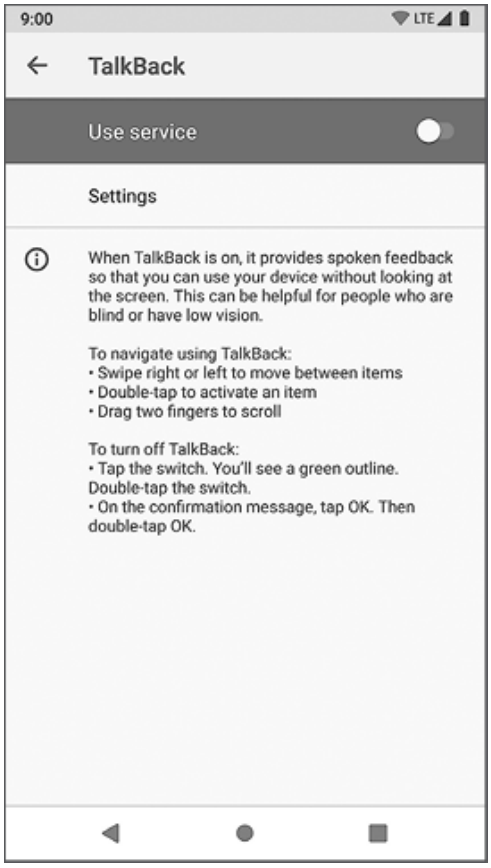


Рис. 18.2. Экран настроек TalkBack

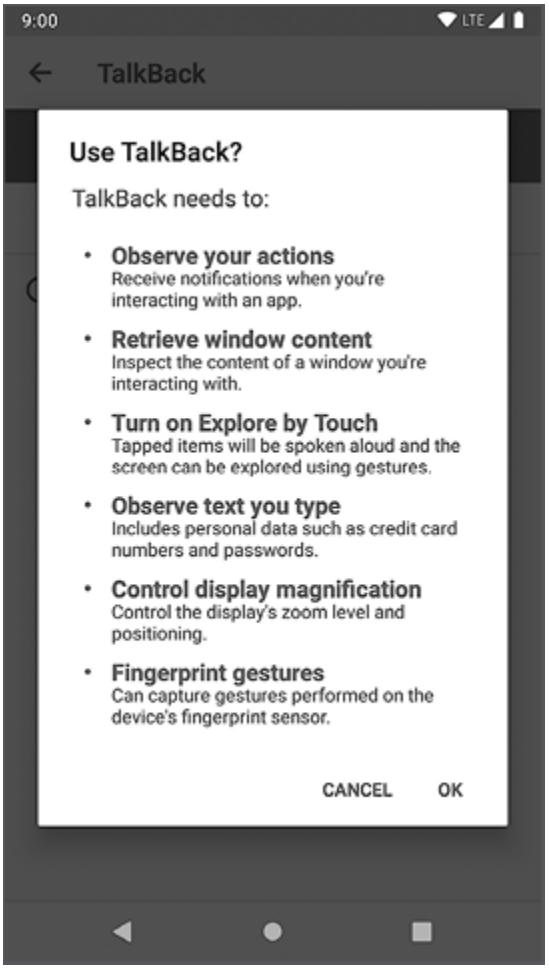


Рис. 18.3. Предоставление разрешений TalkBack

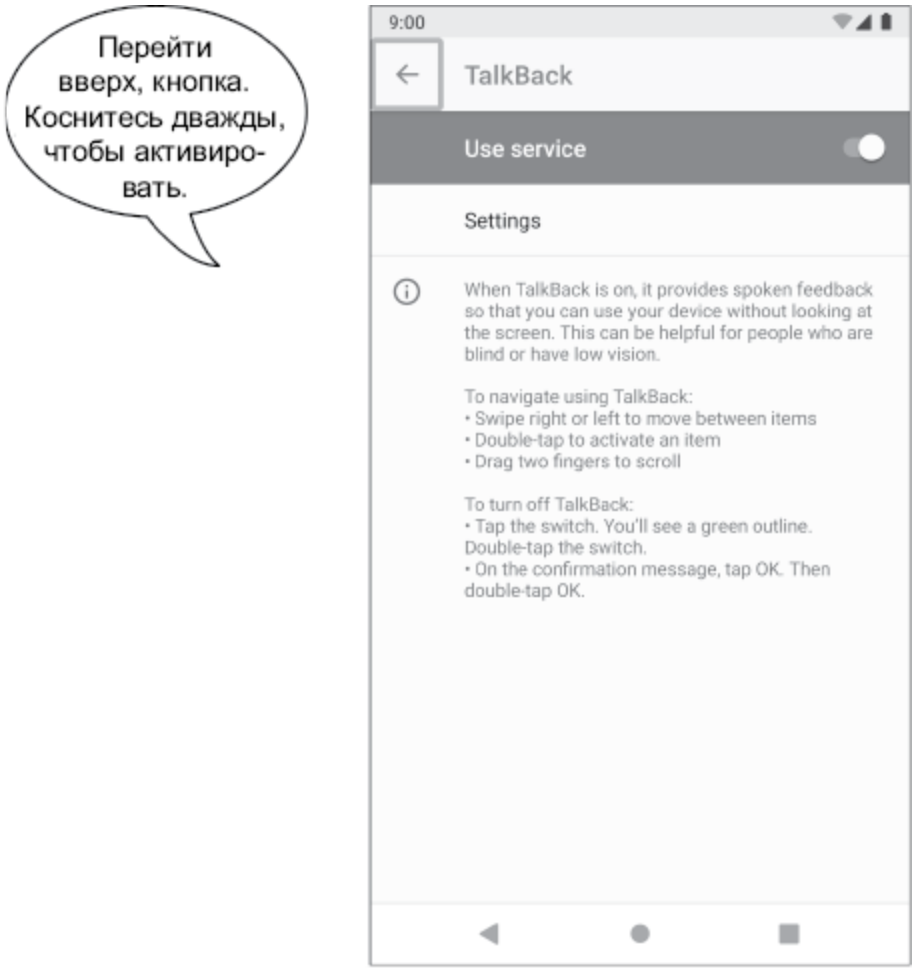


Рис. 18.4. Устройство с включенной функцией TalkBack

При включенной функции TalkBack одно нажатие (или «касание») передает элементу фокус специальных возможностей. Двойное касание в любом месте экрана активизирует элемент. Таким образом, когда фокус находится у кнопки «Вверх», двойное касание активизирует кнопку «Вверх»; когда фокус находится у флажка — изменяет его состояние и т.д. (Кроме того, если ваше устройство заблокировано, вы сможете снять блокировку, коснувшись изображения замка и выполнив двойное касание в любой точке экрана.)

Функция Explore by Touch

Включив TalkBack, вы также включили функцию **ExplorebyTouch** в TalkBack. Это означает, что устройство будет проговаривать информацию об элементе сразу же после нажатия. (Предполагается, что для нажатого элемента задана информация, которую TalkBack может зачитать, — вскоре мы расскажем об этом подробнее.)

Оставьте у кнопки «Вверх» выделение и фокус специальных возможностей. Выполните двойное касание в любой точке экрана. Устройство возвращается к меню специальных возможностей, а TalkBack произносит его название.

Для виджетов инфраструктуры Android — `Toolbar`, `RecyclerView` и `Button` — реализована встроенная поддержка TalkBack. Старайтесь по возможности использовать виджеты инфраструктуры, чтобы пользоваться результатами уже выполненной работы в области специальных возможностей. Также возможно обеспечить правильную реакцию на события специальных возможностей для нестандартных виджетов, но эта тема выходит за рамки книги.

Чтобы прокрутить список на физическом устройстве, перетаскивайте его вверх или вниз сразу двумя пальцами на экране. Чтобы прокрутить список на эмуляторе, удерживайте кнопку ⌘ (**Ctrl**) на клавиатуре, щелкните по одному из двух больших полупрозрачных кругов, которые появляются, и перетащите вверх или вниз с помощью мыши или трекпада (рис. 18.5).

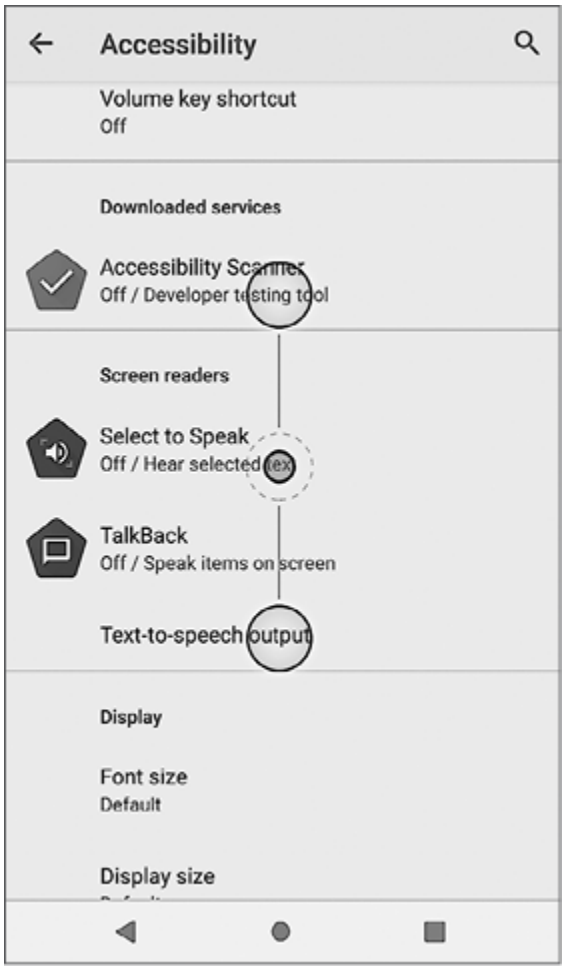


Рис. 18.5. Прокрутка на эмуляторе

В зависимости от длины списка при прокрутке вы будете слышать различные тональные сигналы. Эти звуки, называемые *earcon*, используются для передачи метаданных о взаимодействии.

Линейная навигация смахиванием

Представьте, что это такое: впервые исследовать приложение «на ощупь». Вы еще не знаете, что где находится. Что, если единственный способ ознакомиться со структурой экрана — нажимать все подряд, пока вы не попадете на элемент, который TalkBack сможет прочитать вслух? Возможно, какие-то

элементы будут нажаты повторно, или, что еще хуже, какие-то элементы будут пропущены.

К счастью, существует способ линейного изучения пользовательского интерфейса. Собственно, это более распространенный вариант использования TalkBack: смахивание направо переводит фокус специальных возможностей к следующему элементу на экране, а смахивание налево — к предыдущему элементу. Пользователь последовательно перебирает элементы, вместо того чтобы наугад тыкать в экран в надежде наткнуться на что-то осмысленное.

Попробуйте сами. Запустите приложение CriminalIntent и перейдите к экрану со списком преступлений. По умолчанию в фокусе будет находиться элемент **+** на панели приложения. (Если нет, нажмите на **+**, чтобы сфокусироваться на нем.) Устройство скажет: «CriminalIntent. Добавить новое преступление. Коснитесь дважды, чтобы активировать» (рис. 18.6).

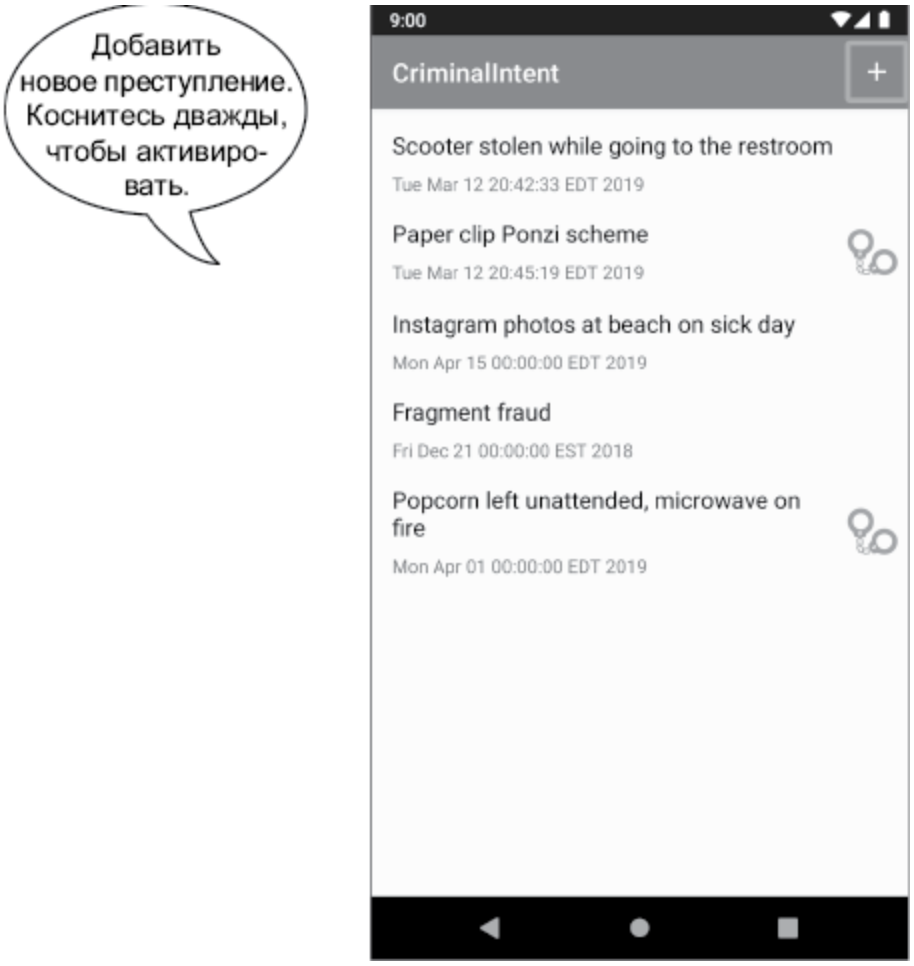


Рис. 18.6. Добавление нового преступления

Для виджетов фреймворка, например элементов меню и кнопок, TalkBack будет по умолчанию считывать видимый текстовый контент, отображаемый на виджете. Но элемент **NewCrime** меню является просто значком и не содержит видимого текста. В этом случае TalkBack ищет другую информацию в виджетах. Вы указали свойство `title` в XML-коде меню, и TalkBack передает эту информацию пользователю. TalkBack также предоставляет подробную информацию о действиях, которые пользователь может выполнять с виджетами, а иногда и информацию о том, что это за виджет.

Теперь проведите пальцем влево. Фокус перемещается на заголовок **CriminalIntent** на панели приложения. TalkBack

говорит: «CriminalIntent» (рис. 18.7).

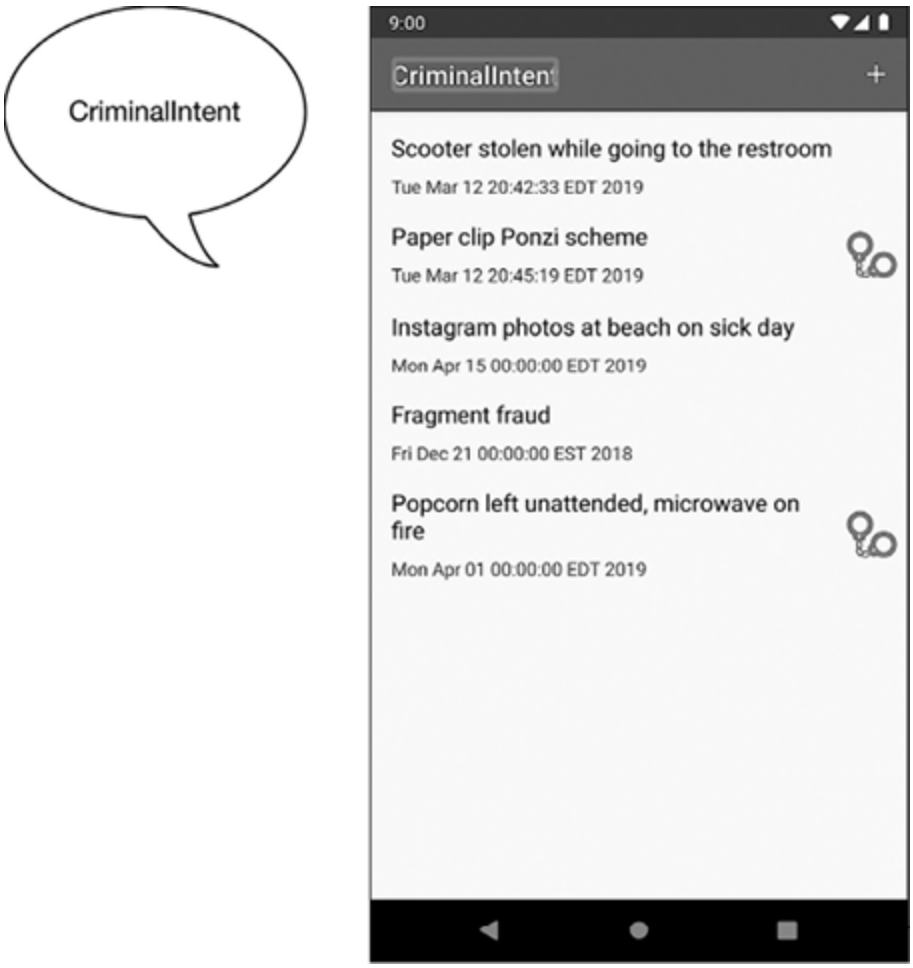


Рис. 18.7. Выбран заголовок панели приложения

Проведите пальцем вправо, и TalkBack снова прочитает информацию о кнопке меню **+**. Второй раз проведите пальцем вправо, после чего фокус переместится на первое преступление в списке. Проведите пальцем влево, и фокус переместится обратно на кнопку меню +. Android старается перемещать фокус осмысленным и понятным образом.

Чтение нетекстовых элементов

Когда выбрана кнопка создания преступления, выполните двойное касание в любой точке экрана, чтобы открыть экран с подробной информацией о преступлении.

Добавление описаний к контенту

На экране с подробной информацией о преступлении нажмите кнопку создания фотографии, чтобы передать ей фокус специальных возможностей (рис. 18.8). TalkBack объявляет: «Кнопка без подписи. Дважды коснитесь, чтобы активировать». (Возможно, на вашем устройстве текст будет немного другим, это зависит от используемой версии Android.)

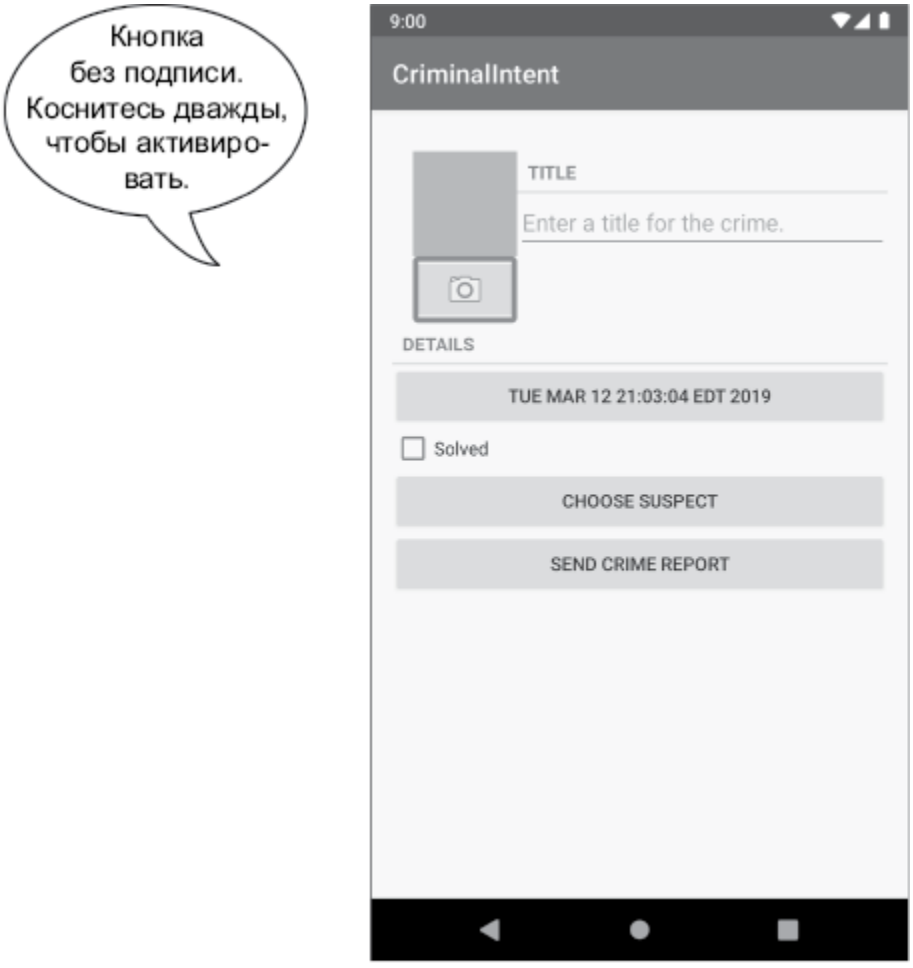


Рис. 18.8. Кнопка создания фотографии

На кнопке управления камерой текста нет, поэтому TalkBack описывает ее, как может. Но несмотря на все старания, эта информация не поможет пользователю с ослабленным зрением.

К счастью, проблема легко решается. Вы сами указываете информацию, которую должен зачитывать диктор TalkBack; для этого следует добавить *описание контента* к виджету `ImageButton`. Описание контента — текст, который описывает виджет и зачитывается TalkBack. (Заодно мы добавим описание содержимого для виджета `ImageView`, в котором выводится изображение.)

Описание контента можно задать в XML-файле макета — необходимое значение присваивается атрибуту `android:contentDescription`. Собственно, именно это мы сейчас и сделаем. Также описание контента может быть задано в коде инициализации вызовами `вашеПредставление.setContentDescription(вашаСтрока)`; этот способ будет применен позднее в этой главе.

Текст описания должен быть осмысленным, но не слишком длинным. Помните: пользователи TalkBack слушают звук, а время прослушивания пропорционально длине текста. Скорость воспроизведения в TalkBack можно повысить, но и в этом случае лучше не включать избыточную информацию и не тратить время пользователя. Например, если вы задаете описание для встроенного виджета, не стоит включать информацию о типе виджета (например, «кнопка»), потому что TalkBack уже знает и произносит эту информацию.

Сначала небольшая подготовительная работа. Добавьте следующие строки в файл `res/values/strings.xml` без квалификаторов.

Листинг 18.1. Добавление строк с описанием контента (res/values/strings.xml)

```
<resources>
    ...
                                        <string
name="crime_details_label">Details</string>
                                        <string
name="crime_solved_label">Solved</string>
                                        <string
name="crime_photo_button_description">Take
photo of crime scene</string>
                                        <string
name="crime_photo_no_image_description">
        Crime scene photo (not set)
    </string>
                                        <string
name="crime_photo_image_description">Crime
scene photo (set)</string>
    ...
</resources>
```

Android Studio подчеркивает вновь добавленные строки красным цветом, предупреждая вас, что вы еще не задали испанскую версию этих новых строк. Чтобы исправить это, добавьте их в файл `res/values-es/strings.xml`.

Листинг 18.2. Добавление строк с описанием испанского контента (res/values-es/strings.xml)

```
<resources>
    ...
```

```
                                                <string
name="crime_details_label">Detalles</string>
                                                <string
name="crime_solved_label">Solucionado</string>
                                                <string
name="crime_photo_button_description">
        Tomar foto de la escena del crimen
    </string>
                                                <string
name="crime_photo_no_image_description">
            Foto de la escena del crimen (no
establecida)
    </string>
                                                <string
name="crime_photo_image_description">
              Foto de la escena del crimen
(establecida)
    </string>

</resources>
```

Теперь откройте файл `res/layout/fragment_crime.xml` и задайте описание контента для `ImageButton`.

Листинг 18.3. Назначение описания контента для ImageButton (res/layout/fragment_crime.xml)

```
<ImageButton
    android:id="@+id/crime_camera"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
```

```
        android:src="@android:drawable/ic_menu_came
    ra"
        android:contentDescription="@string/crime_p
    hoto_button_description"/>
```

Запустите приложение CriminalIntent и нажмите кнопку управления камерой. TalkBack подсказывает: «Сделать фотографию места преступления. Дважды коснитесь, чтобы активировать». Эта информация намного полезнее, чем «Кнопка без подписи».

Теперь нажмите на поле для хранения фотографии (в нашем случае оно содержит серый прямоугольник). Возможно, вы ожидаете, что фокус специальных возможностей переместится к `ImageView`, но зеленый контур появится вокруг всего фрагмента, а TalkBack выдаст общую информацию о фрагменте вместо `ImageView`. В чем дело?

Фокусировка на виджетах

Проблема в том, что виджет `ImageView` не зарегистрирован для получения фокуса. Некоторые виджеты (такие как `Button`) могут получать фокус по умолчанию. Другие, такие как `ImageView`, фокус не получают. Также можно сделать элемент доступным для фокуса, добавив `android:contentDescription`.

Разрешите получение фокуса специальных возможностей виджетом `ImageView` с фотографией преступления, добавив описание контента.

Листинг 18.4. Разрешение получения фокуса виджетом ImageView (res/layout/fragment_crime.xml)

```
<ImageView
```

```
    android:id="@+id/crime_photo"
    ...
    android:background="@android:color/darker_g
ray"
    android:contentDescription="@string/crime_p
hoto_no_image_description" />
```

Снова запустите CriminalIntent и нажмите на фотографию. Виджет `ImageView` получает фокус, и TalkBack объявляет: «Фотография места преступления (не задана)» (рис. 18.9).

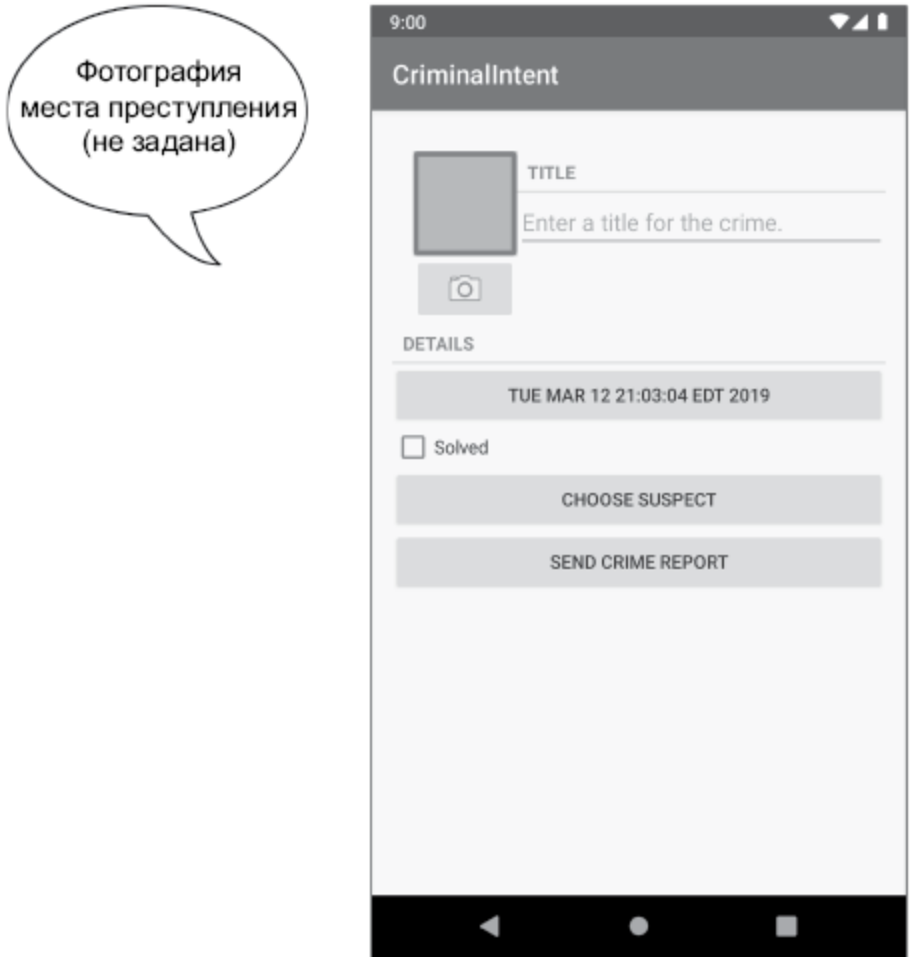


Рис. 18.9. Получение фокуса виджетом ImageView

Создание сопоставимого опыта взаимодействия

Описание контента должно задаваться для любого UI-виджета, который предоставляет информацию пользователю, но не в текстовом виде (например, в форме графики). Если виджет декоративный, прикажите TalkBack игнорировать его, задав в качестве описания контента `null`.

Возможно, вы думаете: «Если пользователь все равно не видит, зачем ему знать, есть изображение или нет?» Однако вы не должны ничего предполагать относительно пользователей. Что еще важнее, вы должны проследить за тем, чтобы пользователь с ослабленным зрением получал столько же информации и функциональности, что и пользователь с нормальным зрением. Общий опыт взаимодействия и последовательность операций могут быть другими, но все пользователи должны иметь доступ к единой функциональности приложения.

Качественно спроектированное доступное приложение не обязано зачитывать описание каждого объекта на экране. Вместо этого оно должно стараться обеспечить сопоставимый опыт взаимодействия. Какая информация и контекст действительно важны?

Непосредственно сейчас опыт взаимодействия, связанный с фотографией преступления, ограничен. TalkBack всегда сообщает, что изображение не задано, даже если оно будет задано. Чтобы убедиться в этом, нажмите на кнопке камеры, а потом выполните двойное касание на экране, чтобы активизировать ее. Запускается приложение камеры, и TalkBack объявляет: «Камера». Сохраните фотографию: нажмите кнопку и дважды коснитесь в любой точке экрана.

Подтвердите получение фотографии. (Конкретная последовательность действий зависит от того, какое приложение камеры вы используете, но помните: чтобы

активизировать ее, следует выделить кнопку и выполнить двойное касание в любой точке.) Открывается экран с подробной информацией о преступлении и обновленной фотографией. Нажмите на снимок, чтобы передать ему фокус специальных возможностей. TalkBack объявляет: «Фотография места преступления (не задана)».

Чтобы предоставить более актуальную информацию для пользователей TalkBack, динамически назначьте описание контента `ImageView` в `updatePhotoView()`.

Листинг 18.5. Динамическое назначение описания контента (CrimeFragment.kt)

```
class CrimeFragment : Fragment() {
    ...
    private fun updatePhotoView() {
        if (photoFile.exists()) {
                                val bitmap =
getScaledBitmap(photoFile.path,
requireActivity())
            photoView.setImageBitmap(bitmap)
            photoView.contentDescription =
                getString(R.string.crime_photo_
image_description)
        } else {
            photoView.setImageDrawable(null)
            photoView.contentDescription =
                getString(R.string.crime_photo_
no_image_description)
        }
    }
    ...
```

```
}
```

Теперь при обновлении представления с фотографией функцией `updatePhotoView()` будет обновляться и описание контента. Если поле `photoFile` пусто, то назначается описание контента, сообщающее об отсутствии фотографии. В противном случае назначается описание, сообщающее о том, что фотография есть.

Запустите приложение CriminalIntent. Просмотрите экран с подробным описанием преступления, в который была добавлена фотография. Нажмите на фотографии места преступления (рис. 18.10). TalkBack объявляет: «Фотография места преступления (задана)».

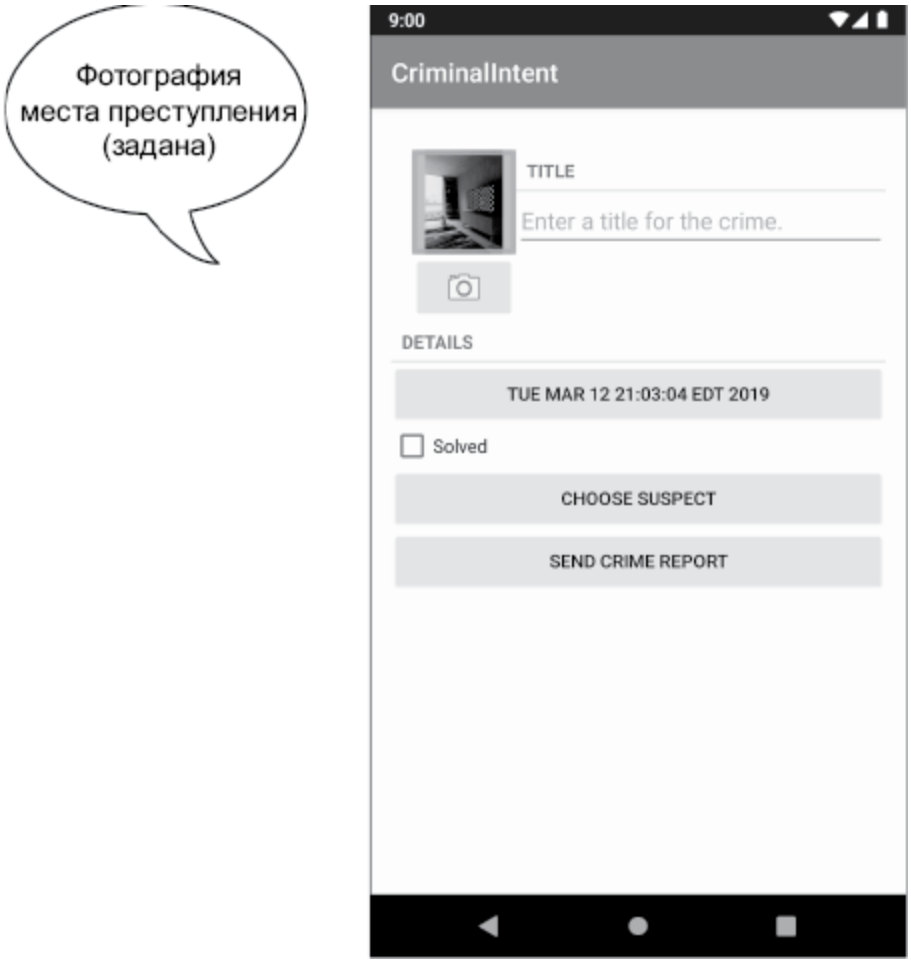


Рис. 18.10. Виджет ImageView с возможностью получения фокуса и динамическим описанием

Поздравляем: приложение стало более доступным людям с ограничениями! Объясняя отсутствие поддержки специальных возможностей в своих приложениях, разработчики обычно отговариваются тем, что не знали, что это нужно. Теперь вы это знаете и видите, как легко сделать приложение доступным для пользователей TalkBack. Кроме того, улучшение поддержки TalkBack в вашем приложении означает, что оно с большей вероятностью будет поддерживать другие средства специальных возможностей, такие как BrailleBack.

Проектирование и реализация специальных возможностей приложения — достаточно серьезное дело. Разработчики делают на этом серьезную карьеру. Но не стоит полностью отказываться от поддержки специальных возможностей только потому, что вы опасаетесь сделать что-то неправильно; лучше начать с азов — проследите за тем, чтобы каждый осмысленный фрагмент контента мог воспроизводиться TalkBack. Убедитесь в том, что пользователи TalkBack получают достаточно контекста, чтобы понять, что происходит в вашем приложении, и при этом они не тратят время на прослушивание лишней информации. А самое важное — прислушивайтесь к мнению пользователей!

На этом наше знакомство с CriminalIntent подходит к концу. За 11 глав мы создали сложное приложение, которое использует фрагменты, взаимодействует с другими приложениями, делает снимки, сохраняет данные и даже говорит на испанском языке. Почему бы не отпраздновать окончание работы куском торта?

Только не забудьте убрать за собой крошки, чтобы ваш проступок не попал в CriminalIntent.

Для любознательных: Accessibility Scanner

В этой главе мы работали над тем, чтобы приложение стало более доступным для людей, использующих TalkBack. Но это еще не все — ослабленное зрение составляет всего лишь одну из подкатегорий специальных возможностей.

В тестировании приложения для обеспечения доступности следует задействовать пользователей, которые регулярно пользуются средствами специальных возможностей. Вы должны постараться улучшить специальные возможности своего приложения.

Компания Google разработала программу Accessibility Scanner, которая анализирует приложения на предмет обеспечения специальных возможностей. Программа предоставляет рекомендации на основании своих результатов. Опробуйте ее на приложении CriminalIntent.

Для начала установите Accessibility Scanner на своем устройстве (рис. 18.11).

После установки и запуска Accessibility Scanner вы увидите синюю галочку, и после этого начнется самое интересное. Запустите CriminalIntent из лаунчера или сводного экрана, не трогая эту галочку. Как только появится CriminalIntent, убедитесь, что на экране отображается подробная информация о преступлении (рис. 18.12).

Нажмите на галочку, и Accessibility Scanner заработает. (Сначала он, возможно, запросит определенные разрешения — предоставьте ему их.) Во время проведения анализа вы увидите индикатор прогресса. После завершения анализа появится окно с результатами (рис. 18.13).

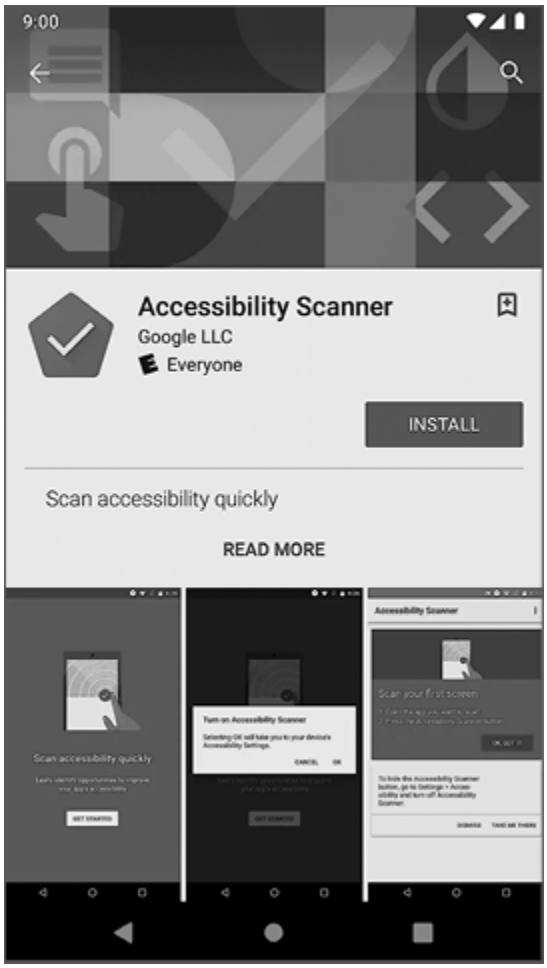


Рис. 18.11. Установка Accessibility Scanner

Рис. 18.12. Запуск CriminalIntent для анализа

Рис. 18.13. Сводка результатов Accessibility Scanner

Виджеты `EditText` и `CheckBox` окружены рамками. Это означает, что программа обнаружила потенциальные проблемы с доступностью специальных возможностей этих виджетов. Нажмите на `CheckBox`, чтобы просмотреть рекомендации по виджету. Нажмите стрелку вниз, чтобы просмотреть подробности (рис. 18.14).

Сканер Accessibility Scanner предлагает увеличить размер поля `CheckBox`. Рекомендуемый минимальный размер для всех сенсорных объектов — 48dp. Размер поля `CheckBox` меньше, что можно легко исправить, настроив атрибут `android:minHeight` для виджета.

Вы можете узнать больше о рекомендациях Accessibility Scanner, нажав кнопку **LEARNMORE**.

Чтобы отключить Accessibility Scanner, перейдите в раздел **Settings**. Выберите пункт **Accessibility**, затем нажмите кнопку **AccessibilityScanner**. Установите переключатель в неактивное положение, чтобы выключить сканер (рис. 18.15).

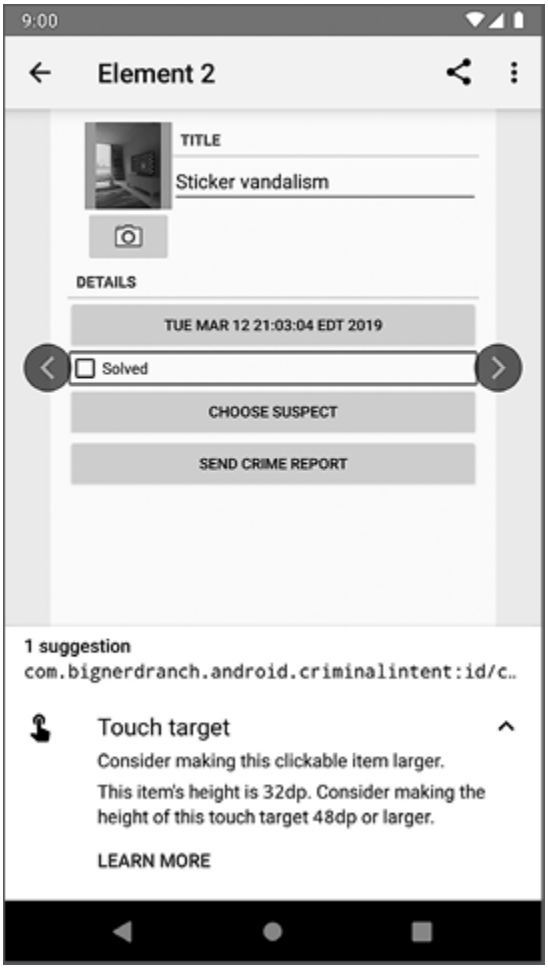


Рис. 18.14. Рекомендации по специальным возможностям CheckBox

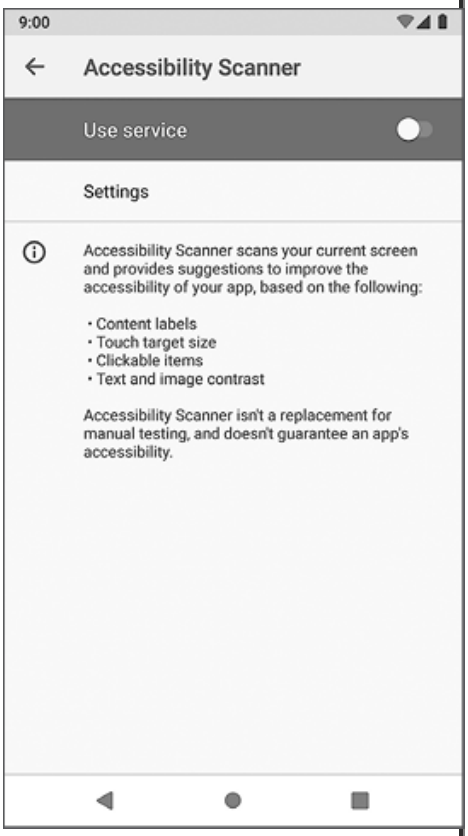


Рис. 18.15. Выключение сканера Accessibility Scanner

Упражнение. Улучшение списка

На экране со списком преступлений TalkBack читает описание и дату каждого элемента. Однако диктор не сообщает, было

преступление раскрыто или нет. Чтобы устранить этот недостаток, назначьте описание контента значку с изображением наручников.

Сводка получается довольно длинной (с учетом формата даты), а статус раскрытия преступления зачитывается в самом конце или вообще не зачитывается, если преступление не раскрыто. Чтобы задание стало еще более интересным, вместо считывания обоих элементов `TextView` и описания содержимого значка (если он присутствует) через TalkBack, добавьте динамическое описание в каждую строку утилизатора (recycler view).

Описание должно содержать сводку данных, выводимых в одной строке.

Упражнение. Предоставление контекста для ввода данных

Кнопкам даты и выбора подозреваемого присущ тот же недостаток, что и исходному текстовому полю с описанием: в приложении нет никаких явных признаков того, для чего нужна кнопка с датой (независимо от того, используется функция TalkBack или нет). Аналогичным образом после выбора контакта в качестве подозреваемого пользователь не получает никакой информации о том, что представляет собой кнопка. Возможно, пользователь и так догадается о предназначении кнопок и текста на этих кнопках, но стоит ли на это надеяться?

В этом проявляется одна из тонкостей проектирования пользовательских интерфейсов. Вы (или ваша дизайн-группа) определяете, какие решения будут наиболее логичными для вашего приложения, чтобы выдержать баланс между простотой интерфейса и простотой взаимодействий.

Измените реализацию экрана с подробной информацией, чтобы пользователь не терял смысловой контекст выбранных им данных. Проблема может решаться простым добавлением надписей ко всем полям. Для этого можно было бы добавить метку TextView к каждой кнопке. Затем мы инструктируем TalkBack, что TextView — это метка для кнопки Button, используя атрибут android:labelFor.

```
<TextView
    android:id="@+id/crime_date_label"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Date"
    android:labelFor="@+id/crime_date"/>
<Button
    android:id="@+id/crime_date"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:text="Wed Nov 14 11:56 EST 2018"/>
```

Атрибут android:labelFor сообщает TalkBack, что TextView служит меткой для виджета, указанного в идентификаторе. labelFor определяется в классе View, поэтому вы можете сделать любой элемент меткой для любого другого. Обратите внимание, что здесь вы должны использовать синтаксис@+id, потому что вы ссылаетесь на идентификатор, который не был определен в тот момент в файле. Теперь вы можете удалить + из строки android:id="@+id/crime_title" в определении EditText, но в этом нет необходимости.

Упражнение. Оповещения о событиях

Динамические описания контента для виджетов `ImageView` с фотографиями места преступления упрощают работу с фотографиями. С другой стороны, пользователь TalkBack должен прикоснуться к виджету `ImageView`, чтобы проверить его статус. Пользователю с нормальным зрением проще: он видит, что изображение изменилось (или не изменилось) при возвращении из приложения камеры.

Аналогичную возможность можно реализовать и средствами TalkBack: вы сообщаете пользователю о произошедшем в результате закрытия приложения камеры. Прочитайте описание функции `View.announceForAccessibility(...)` в документации и используйте его в CriminalIntent в подходящий момент.

Возможно, вы решите выдать оповещение в `onActivityResult(...)`. В таком случае могут возникнуть проблемы с временем выдачи оповещения, связанные с жизненным циклом activity. Проблему можно решить небольшой задержкой оповещения за счет отправки `Runnable` (эта тема более подробно рассматривается в главе 25). Реализация выглядит примерно так:

```
someView.postDelayed(Runnable {
    // место для кода для объявления
}, SOME_DURATION_IN_MILLIS)
```

Также можно обойтись без `Runnable` и воспользоваться другим механизмом для определения момента оповещения об изменениях. Например, оповещение можно выдать в `onResume()`, хотя в этом случае нужно будет проверять, вернулся ли пользователь из приложения камеры или нет.

19. Привязка данных и MVVM

В этой главе также начнется работа над новым приложением BeatBox (рис. 19.1), предназначенным для воспроизведения всевозможных угрожающих звуков. Оно поможет вам победить соперников по боксу. Однако приложение не поможет в тренировках, чтобы вы стали быстрее и сильнее своего противника. Помогать оно будет в трудной части: издавать крики, пугающие соперника, чтобы он сдался.

Рис. 19.1. Вид приложения BeatBox к концу этой главы

В этом проекте вы узнаете, как использовать библиотеку компонентов архитектуры Jetpack под названием *data binding* (связывание/привязка данных). Вы будете использовать привязку данных для реализации архитектуры *Model-View-ViewModel*, *MVVM* (Модель-Представление-МодельПредставления). Кроме того, вы узнаете, как использовать систему активов для хранения звуковых файлов.

Другие архитектуры: зачем они нужны?

Во всех приложениях, написанных вами до сих пор, используется простая версия MVC. И до сих пор — если мы хорошо справились со своим делом — все приложения выглядели вполне логично. Зачем что-то менять? В чем проблема?

Архитектура MVC в том виде, в каком она применяется в книге, хорошо подходит для небольших простых приложений. В нее легко добавляются новые функции, и она позволяет легко представить взаимодействие между частями приложения. MVC закладывает надежную основу для разработки, позволяет быстро построить работоспособное приложение и эффективно работает на ранних стадиях работы над проектом.

Проблемы начинаются тогда, когда размеры вашей программы выходят за рамки примеров, приведенных в книге, как это происходит практически во всех программах. Большие фрагменты и activity трудно расширять, и в логике их работы трудно разобраться. На реализацию новых возможностей и исправление ошибок уходит больше времени. На какой-то стадии контроллеры необходимо разделить на меньшие, более удобные части.

Как это сделать? Определите различные операции, выполняемые большим классом контроллера, и выделите

каждую операцию в отдельный класс. Вместо одного большого класса появятся экземпляры нескольких классов, которые будут совместно выполнять общую работу.

Как определить эти разные задания? Ответ на этот вопрос лежит в определении архитектуры. Такие описания, как «Модель — Представление — Контроллер» (MVC) и «Модель — Представление — МодельПредставления» (MVVM), используются в качестве ответов на высоком уровне. Однако в конечном итоге на этот вопрос всегда отвечаете только вы, и итоговая архитектура определяется только вами.

Приложение BeatBox построено на базе архитектуры MVVM. Нам архитектура MVVM очень нравится, потому что она отлично справляется с вынесением большого объема рутинного кода контроллера в файл макета, где вы сразу видите, какие части интерфейса являются динамическими. В то же время содержательный динамический код контроллера выносится в *класс модели представления*, что упрощает его тестирование и проверку работоспособности.

Величина модели представления всегда должна определяться здравым смыслом. Если ваша модель представления будет слишком большой, ее можно разбить на несколько частей. Ваша архитектура принадлежит только вам, и никому другому.

Сравнение моделей представления MVVM и Jetpack-класса ViewModel

Перед началом работы над новым проектом ознакомьтесь с терминологией: модель представления, входящая в MVVM, — это не то же самое, что Jetpack-класс `ViewModel`, о котором вы узнали в главах 4 и 9. Чтобы избежать путаницы, мы всегда

будем оформлять имя Jetpack-класса как `ViewModel`, а концепцию MVVM как «модель представления».

Напомним, что `ViewModel` — это специальный Jetpack-класс для обновления данных во фрагментах и activity с учетом непостоянства их жизненных циклов. Модель представления в MVVM — более концептуальная часть архитектуры. Модели представлений могут быть реализованы с помощью Jetpack-классов `ViewModel`, но, как вы увидите из этой главы, они также могут быть реализованы без использования класса `ViewModel`.

Создание приложения BeatBox

Пора браться за дело. Начнем с создания приложения BeatBox.

Выполните в Android Studio команду **File⇒New⇒NewProject** для создания нового проекта. Присвойте ему имя **BeatBox** и введите имя пакета **com.bignerdranch.android.beatbox**. Выберите опцию **UseAndroidXartifacts**. Оставьте остальные параметры настроенными по умолчанию.

В приложении снова будет использоваться виджет `RecyclerView`, поэтому откройте настройки проекта и добавьте зависимость `androidx.recyclerview:recyclerview:1.0.0` в файл `app/build.gradle` (не забудьте выполнить синхронизацию). Замените автоматически сгенерированное содержимое файла `res/layout/activity_main.xml` одним `RecyclerView`.

Листинг 19.1. Изменение основного файла макета для MainActivity (res/layout/activity_main.xml)

```
<androidx.constraintlayout.widget.ConstraintLayout
```

```
            ...
            tools:context=".MainActivity">
            ...
    </androidx.constraintlayout.widget.ConstraintLayout>
    <androidx.recyclerview.widget.RecyclerView
            xmlns:android="http://schemas.android.com/apk/res/android"
            android:id="@+id/recycler_view"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
```

Запустите приложение. Вы должны увидеть пустой экран, что говорит о том, что вы подключили все правильно. Поздравьте себя и двигайтесь дальше.

Простая привязка данных

Следующая задача — подключение `RecyclerView`. Вы уже делали это прежде, но на этот раз мы используем привязку данных для ускорения работы.

Привязка данных дает ряд преимуществ, которые облегчают жизнь при работе с макетами. В простых случаях, наподобие того что вы увидите в этом разделе, это позволяет получить доступ к представлениям без необходимости вызывать `findViewById(...)`. Позже вы увидите более продвинутое использование привязки данных, в том числе и то, как она помогает вам реализовать MVVM.

Начнем с включения привязки данных и применения плагина `kotlin-kapt` в файле `build.gradle` вашего приложения.

Листинг 19.2. Включение привязки данных (app/build.gradle)

```
apply plugin: 'kotlin-kapt'

android {
    ...
    buildTypes {
        ...
    }
    dataBinding {
        enabled = true
    }
}
```

Обратите внимание, что применение плагина `kotlin-kapt` позволяет выполнять привязку данных для обработки аннотаций на Kotlin. Важность этого действия объясняется позже в этой главе.

Чтобы использовать привязку данных в файле макета, необходимо создать файл макета привязки данных. Для этого весь XML-файл оборачивается в тег `<layout>`. Сделайте это изменение в файле `activity_main.xml`, как показано в листинге 19.3.

Листинг 19.3. Заключение разметки в тег <layout> (res/layout/activity_main.xml)

```
<layout
xmlns:android="http://schemas.android.com/apk/res/android">
    <androidx.recyclerview.widget.RecyclerView
                    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
            android:id="@+id/recycler_view"
            android:layout_width="match_parent"
            android:layout_height="match_parent
"/>
</layout>
```

Тег `<layout>` сигнализирует средствам привязки данных, что они должны обработать файл макета. После завершения обработки будет сгенерирован *класс привязки*. По умолчанию имя класса выбирается в соответствии с именем файла макета, но вместо стандартной схемы регистра (змеиного_регистра) используется ГорбатыйРегистр.

Таким образом, библиотека привязки данных должна уже сгенерировать класс привязки `ActivityMainBinding` для вашего файла `activity_main.xml`. Этот класс будет использоваться для привязки данных: вместо заполнения иерархии представлений с использованием `setContentView(int)` мы заполним экземпляр `ActivityMainBinding`. `ActivityMainBinding` сохраняет иерархию представлений в свойстве `root`. Кроме того, сохраняются именованные ссылки для всех представлений, помеченных в файле макета атрибутом `android:id`.

Таким образом, класс `ActivityMainBinding` сохраняет две ссылки: `root` для всего макета и `recyclerView` для `RecyclerView` (рис. 19.2).

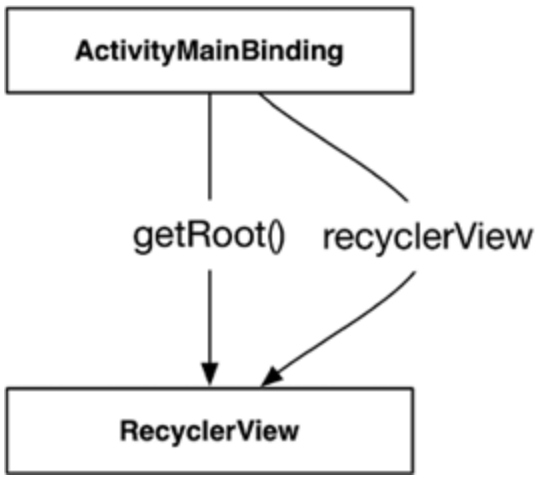


Рис. 19.2. Класс привязки

Конечно, в нашем макете только одно представление, поэтому обе ссылки указывают на одно представление: `RecyclerView`.

Перейдем к использованию класса привязки. Переопределите `onCreate(Bundle?)` в `MainActivity` и используйте `DataBindingUtil` для заполнения экземпляра `ActivityMainBinding` (листинг 19.4).

Класс `ActivityMainBinding` должен импортироваться, как и любой другой класс. Если Android Studio не может найти `ActivityMainBinding`, это означает, что класс не был автоматически сгенерирован по какой-либо причине. Прикажите Android Studio сгенерировать класс командой **Build⇒RebuildProject**. Если класс не был сгенерирован после повторной сборки проекта, перезапустите Android Studio.

Листинг 19.4. Заполнение класса привязки (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState:
Bundle?) {
```

```
        super.onCreate(savedInstanceState)
            ~~setContentView(R.layout.activity_ma~~
~~in)~~

        val binding: ActivityMainBinding =
            DataBindingUtil.setContentView(this
, R.layout.activity_main)
    }
}
```

После создания привязки можно заняться виджетом `RecyclerView` и его настройкой.

Листинг 19.5. Настройка RecyclerView (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

        val binding: ActivityMainBinding =
            DataBindingUtil.setContentView(this
, R.layout.activity_main)

        binding.recyclerView.apply {
                                layoutManager  =
GridLayoutManager(context, 3)
        }
    }
}
```

Теперь создайте макет для кнопок, res/layout/list_item_sound.xml. Здесь также будет использоваться привязка данных, поэтому заключите файл макета в тег <layout>.

Листинг 19.6. Создание макета для кнопок (res/layout/list_item_sound.xml)

```
<layout
xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools">
    <Button
        android:layout_width="match_parent"
        android:layout_height="120dp"
        tools:text="Sound name"/>
</layout>
```

Затем создайте объект SoundHolder, связанный с list_item_sound.xml.

Листинг 19.7. Создание объекта SoundHolder (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

     override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    private inner class SoundHolder(private val binding: ListItemSoundBinding) :
```

```
            RecyclerView.ViewHolder(binding.roo
t) {
    }
}
```

Затем создайте Adapter, связанный с SoundHolder.

Листинг 19.8. Создание класса SoundAdapter (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
    private inner class SoundHolder(private val
binding: ListItemSoundBinding) :
            RecyclerView.ViewHolder(binding.roo
t) {
    }

    private inner class SoundAdapter() :
        RecyclerView.Adapter<SoundHolder>() {

        override fun onCreateViewHolder(parent:
ViewGroup, viewType: Int):
                SoundHolder {
                            val   binding   =
DataBindingUtil.inflate<ListItemSoundBinding>(
                layoutInflater,
                R.layout.list_item_sound,
                parent,
                false
            )
            return SoundHolder(binding)
        }
```

```
        override fun onBindViewHolder(holder:
SoundHolder, position: Int) {
        }

        override fun getItemCount() = 0
    }
}
```

Подключите SoundAdapter в функции onCreate(Bundle?).

Листинг 19.9. Подключение SoundAdapter (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    super.onCreate(savedInstanceState)

    val binding: ActivityMainBinding =
            DataBindingUtil.setContentView(this,
R.layout.activity_main)

    binding.recyclerView.apply {
                                layoutManager =
GridLayoutManager(context, 3)
        adapter = SoundAdapter()
    }
}
```

Мы использовали привязку данных для настройки recycler view. К сожалению, выводить на экран пока нечего. Исправим это, передав программе парочку звуковых файлов.

Импорт активов

Пришло время добавить в проект звуковые файлы и загрузить их на стадии выполнения. Вместо системы ресурсов будут использоваться *активы* (*assets*). Активы можно рассматривать как усеченные аналоги ресурсов: они упаковываются в APK, как и ресурсы, но без конфигурационного инструментария, дополняющего ресурсы.

В некоторых отношениях это хорошо. Из-за отсутствия системы конфигурации активам можно присваивать любые имена и упорядочивать их в структуру папок. Впрочем, есть и оборотная сторона: без системы конфигурации ваше приложение не сможет автоматически реагировать на изменение плотности пикселов, языка или ориентации или же автоматически использовать активы в файлах макетов или других ресурсах.

Обычно предпочтение отдается ресурсам. Однако в тех случаях, когда вы обращаетесь к файлам только на программном уровне, активы выходят на первый план. Например, многие игры используют активы для хранения графики и звука — и такое же решение будет использовано в BeatBox.

Используемые активы следует импортировать в проект. Создайте в проекте папку для активов: щелкните правой кнопкой мыши по модулю `app` и выберите команду **New⇒Folder⇒AssetsFolder** в контекстном меню (рис. 19.3). Оставьте флажок **ChangeFolderLocation** сброшенным, а в списке **TargetSourceSet** выберите вариант **main**.

Рис. 19.3. Создание папки assets

Нажмите кнопку **Finish**, чтобы создать папку для активов.

Щелкните правой кнопкой мыши по папке `assets` и выберите команду **New⇒Directory** в контекстном меню. Введите имя каталога **sample_sounds** (рис. 19.4).

Рис. 19.4. Создание папки sample_sounds

Все содержимое папки `assets` интегрируется в приложение. Для удобства и порядка мы создали вложенную папку `sample_sounds`. Впрочем, в отличие от ресурсов, делать это было необязательно.

Где найти звуки? Мы воспользуемся подборкой, распространяемой на условиях лицензии Creative Commons, которую мы впервые увидели у пользователя plagasul (**www.freesound.org/people/plagasul/packs/3/**). Мы поместили все звуки в один zip-архив по адресу **www.bignerdranch.com/solutions/sample_sounds.zip**.

Загрузите архив и распакуйте его содержимое в папку `assets/sample_sounds` (рис. 19.5).

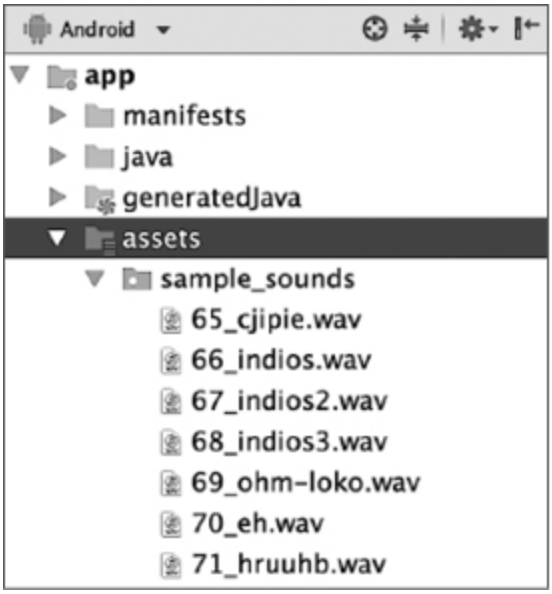


Рис. 19.5. Импортированные активы

(Кстати, проследите за тем, чтобы там находились файлы `.wav`, а не файл `.zip`, из которого они были извлечены.)

Соберите приложение заново. Следующим шагом станет получение списка активов и вывод его для пользователя.

Доступ к активам

Приложение будет выполнять множество операций, относящихся к управлению активами: поиск, отслеживание и в конечном итоге воспроизведение их как звуков. Для выполнения этих операций создайте новый класс `BeatBox` в пакете `com.bignerdranch.android.beatbox`. Добавьте пару констант: для вывода информации в журнал и для имени папки, в которой были сохранены звуки.

Листинг 19.10. Новый класс BeatBox (BeatBox.kt)

```
private const val TAG = "BeatBox"
private const val SOUNDS_FOLDER = "sample_sounds"

class BeatBox {

}
```

Для обращения к активам используется класс `AssetManager`. Экземпляр этого класса можно получить для любой разновидности `Context`. Так как `BeatBox` понадобится такой экземпляр, предоставьте ему `AssetManager`.

Листинг 19.11. Сохранение AssetManager (BeatBox.kt)

```
private const val TAG = "BeatBox"
private const val SOUNDS_FOLDER = "sample_sounds"

class BeatBox(private val assets: AssetManager) {
```

```
}
```

Как правило, при работе с активами вам не нужно беспокоиться о том, какой именно объект `Context` будет использоваться. Во всех ситуациях, которые вам встретятся на практике, объект `AssetManager` всех разновидностей `Context` будет связан с одним набором активов.

Чтобы получить список доступных активов, используйте функцию `list(String)`. Напишите функцию `loadSounds()`, которая обращается к активам при помощи функции `list(String)`.

Листинг 19.12. Получение списка активов (BeatBox.kt)

```
class BeatBox(private val assets: AssetManager)
{

fun loadSounds(): List<String> {
    try {
                                val soundNames =
assets.list(SOUNDS_FOLDER)!!
                                 Log.d(TAG, "Found
${soundNames.size} sounds")
            return soundNames.asList()
        } catch (e: Exception) {
            Log.e(TAG, "Could not list assets",
e)
            return emptyList()
        }
    }
}
```

Функция AssetManager.list(String) возвращает список имен файлов, содержащихся в заданной папке. Передав ему путь к папке со звуками, вы получите информацию обо всех файлах .wav в этой папке.

Чтобы убедиться в том, что система активов работает правильно, создайте экземпляр BeatBox в MainActivity и вызовите функцию loadSounds().

Листинг 19.13. Создание экземпляра BeatBox (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    private lateinit var beatBox: BeatBox

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

        beatBox = BeatBox(assets)
        beatBox.loadSounds()

        val binding: ActivityMainBinding =
            DataBindingUtil.setContentView(this
, R.layout.activity_main)

        binding.recyclerView.apply {
                                layoutManager   =
GridLayoutManager(context, 3)
            adapter = SoundAdapter()
        }
    }
    ...
```

```
}
```

Запустите приложение. В журнале должна появиться информация о том, сколько звуковых файлов было обнаружено. Мы использовали 22 файла в формате .wav, и если вы использовали наши файлы, результат должен выглядеть так:

```
...1823-1823/com.bignerdranch.android.beatbox
D/BeatBox: Found 22 sounds
```

Подключение активов для использования

Имена файлов активов получены, теперь нужно вывести их для пользователя. В конечном итоге файлы должны воспроизводиться, поэтому стоит создать объект для хранения имени файла, того имени, которое видит пользователь, и всей остальной информации, относящейся к данному звуку.

Создайте класс Sound для хранения всей этой информации.

Листинг 19.14. Создание объекта Sound (Sound.kt)

```
private const val WAV = ".wav"

class Sound(val assetPath: String) {

                        val name =
assetPath.split("/").last().removeSuffix(WAV)
}
```

В конструкторе выполняется небольшая подготовительная работа для генерации удобочитаемого имени звука. Сначала имя файла отделяется вызовом

String.split(String).last(), после чего вызов String.removeSuffix(String) удаляет расширение.

Далее в функции BeatBox.loadSounds() строится список объектов Sound.

Листинг 19.15. Создание объектов Sound (BeatBox.kt)

```
class BeatBox(private val assets: AssetManager)
{

    val sounds: List<Sound>

    init {
        sounds = loadSounds()
    }

                            fun      loadSounds():
   ~~List<String>~~List<Sound> {

        val soundNames: Array<String>

        try {
                                ~~val~~  soundNames  =
assets.list(SOUNDS_FOLDER)!!
                                ~~Log.d(TAG,  "Found
~~${soundNames.size} sounds")~~
                    ~~return soundNames.asList()~~
        } catch (e: Exception) {
            Log.e(TAG, "Could not list assets",
e)
            return emptyList()
        }
```

```
        val sounds = mutableListOf<Sound>()
        soundNames.forEach { filename ->
                            val assetPath =
"$SOUNDS_FOLDER/$filename"
            val sound = Sound(assetPath)
            sounds.add(sound)
        }
        return sounds
    }
}
```

Затем свяжите SoundAdapter со списком объектов Sound.

Листинг 19.16. Связывание со списком объектов Sound (MainActivity.kt)

```
private inner class SoundAdapter(private val
sounds: List<Sound>) :
        RecyclerView.Adapter<SoundHolder>() {
    ...

        override fun onBindViewHolder(holder:
SoundHolder, position: Int) {
    }

            override fun getItemCount() =
    sounds.size
}
```

Звуки из BeatBox передаются функции onCreate(Bundle?).

Листинг 19.17. Передача звуков адаптеру (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    ...

    binding.recyclerView.apply {
                                layoutManager =
GridLayoutManager(context, 3)
        adapter = SoundAdapter(beatBox.sounds)
    }
}
```

Наконец, удалите вызов функции `BeatBox.loadSounds()` из `onCreate`.

Листинг 19.18. Удаление функции BeatBox.loadSounds() из onCreate(...) (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    ...
    beatBox = BeatBox(assets)
        beatBox.loadSounds()
    ...
}
```

Так как функция `BeatBox.loadSounds()` больше не вызывается из-за пределов блока инициализатора `BeatBox`, нет необходимости делать эту функцию публичной. Чтобы предотвратить ложные вызовы `loadSounds()` из других частей кода, обновите модификатор видимости до `private`.

Листинг 19.19. Обновление модификатора видимости BeatBox.loadSounds() до private (BeatBox.kt)

```
class BeatBox(private val assets: AssetManager)
{
    ...
    private fun loadSounds(): List<Sound> {
        ...
    }
}
```

После добавления этого кода при запуске BeatBox на экране появляется сетка с кнопками (рис. 19.6).

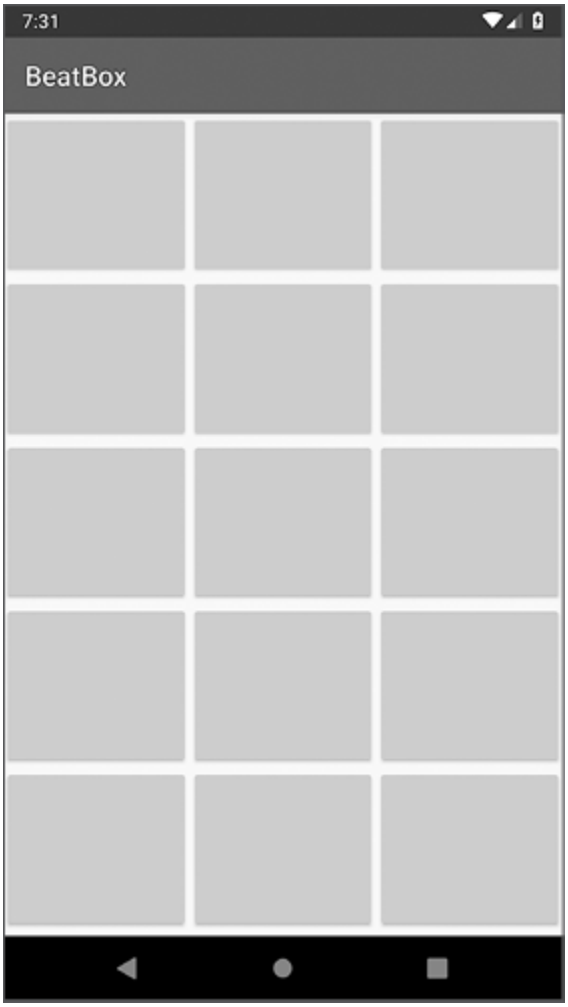


Рис. 19.6. Пустые кнопки

Чтобы заполнить кнопки названиями, необходимо воспользоваться некоторыми дополнительными инструментами привязки данных.

Установление связи с данными

При использовании привязки данных объекты данных могут объявляться в файле макета:

```
<layout
xmlns:android="http://schemas.android.com/apk/r
es/android"
        xmlns:tools="http://schemas.android.com
/tools">
    <data>
        <variable
            name="crime"
            type="com.bignerdranch.android.crim
inalintent.Crime"/>
    </data>
    ...
</layout>
```

Далее значения из этих объектов используются прямо в файле макета при помощи *оператора привязки* `@{}`:

```
<CheckBox
    android:id="@+id/list_item_crime_solved_che
ck_box"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true"
```

```
        android:checked="    @{crime.isSolved()}"
        android:padding="4dp"/>
```

Диаграмма объектов выглядит так, как показано на рис. 19.7.

Рис. 19.7. Связи данных

Наша непосредственная цель — разместить на кнопках названия звуков. Самое простое решение на базе привязки данных основано на прямом связывании с объектом `Sound` в `list_item_sound.xml`, как показано на рис. 19.8.

Рис. 19.8. Прямое связывание

Однако такое решение создает ряд архитектурных проблем. Чтобы понять, в чем дело, взгляните на происходящее с точки зрения MVC (рис. 19.9).

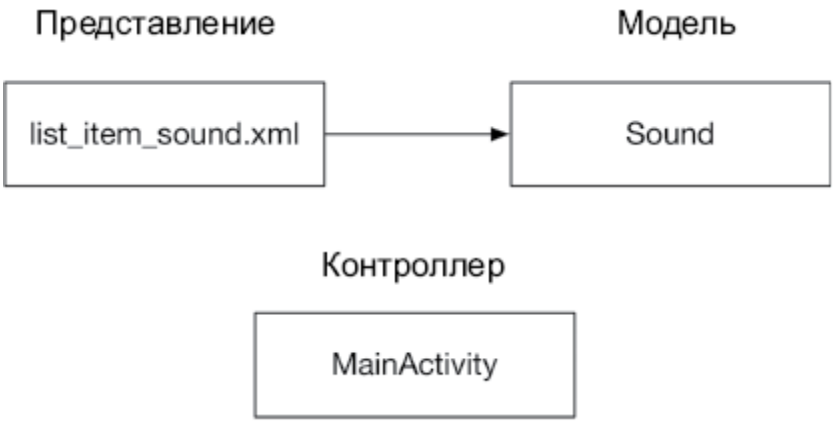


Рис. 19.9. Нарушение архитектуры MVC

Направляющим принципом любой архитектуры должен быть *принцип единственной обязанности*. Этот принцип гласит, что каждый класс должен иметь ровно одну обязанность. MVC дает представление о том, какими могут быть эти обязанности: модель описывает, как работает ваше приложение, контроллер управляет отображением, а представление отображает его на экране так, как вы пожелаете.

Использование привязки данных так, как показано на рис. 19.8, нарушит это разделение обязанностей — ведь, скорее всего, решать, как должны отображаться данные, придется объекту модели `Sound`. В приложении быстро воцарится хаос, потому что файл `Sound.kt` будет загромождан кодом двух видов: определяющим работу приложения и определяющим способ отображения данных.

Вместо того чтобы запутывать обязанности `Sound`, мы добавим новый объект модели представления для привязки данных. Эта модель представления отвечает за принятие решений относительно того, как должны отображаться данные (рис. 19.10).

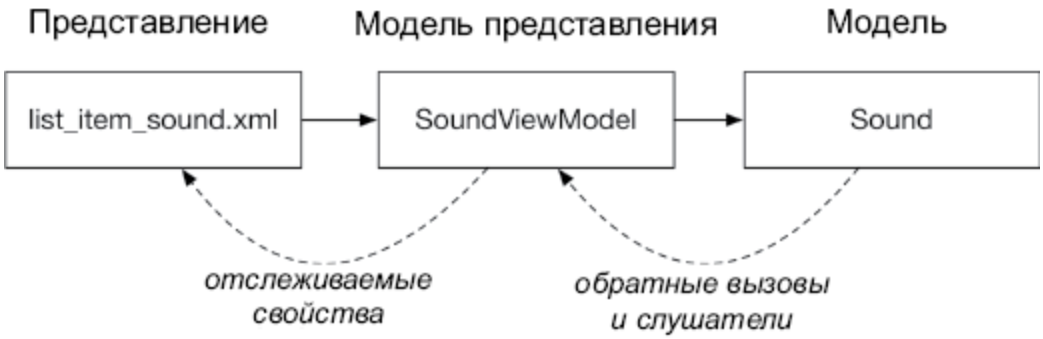


Рис. 19.10. Архитектура MVVM

Такая архитектура называется MVVM, то есть Модель-Представление-Модель Представления. Большая часть работы по форматированию данных, которая ранее выполнялась классами контроллеров, переходит к модели представления.

Подключение виджетов к данным будет осуществляться непосредственно в файле макета с использованием привязки данных к модели представления. Контроллер (ваша activity или фрагмент) будет отвечать за такие операции, как инициализация привязки и модели представления и создание связи между ними.

У модели MVVM нет контроллера, зато activity и фрагменты считаются частью представления.

Создание модели представления

Создайте класс `SoundViewModel`. Класс должен содержать два свойства: для объекта `Sound` и для объекта `BeatBox` (который в конечном итоге будет использоваться для воспроизведения).

Листинг 19.20. Создание SoundViewModel (SoundViewModel.kt)

```
class SoundViewModel {

    var sound: Sound? = null
        set(sound) {
            field = sound
        }
}
```

Эти свойства образуют интерфейс, который будет использоваться вашим адаптером. Файлу макета потребуется дополнительная функция для получения названия, которое должно отображаться на кнопке. Добавьте его в `SoundViewModel`.

Листинг 19.21. Добавление функций привязки (SoundViewModel.kt)

```
class SoundViewModel {

    var sound: Sound? = null
        set(sound) {
            field = sound
        }

    val title: String?
        get() = sound?.name
}
```

Связывание с моделью представления

Теперь модель представления следует связать с файлом макета. Начните с объявления свойства в файле макета.

Листинг 19.22. Объявление свойства для модели представления (res/layout/list_item_sound.xml)

```
<layout
xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable
            name="viewModel"
            type="com.bignerdranch.android.beatbox.SoundViewModel"/>
    </data>
```

```
    ...
</layout>
```

Этот блок разметки определяет свойство `viewModel` для класса привязки вместе с геттером и сеттером. В классе привязки свойство `viewModel` может использоваться в выражениях привязки.

Листинг 19.23. Привязка названия кнопки (res/layout/list_item_sound.xml)

```
<layout
xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable
            name="viewModel"
            type="com.bignerdranch.android.beatbox.SoundViewModel"/>
    </data>
    <Button
        android:layout_width="match_parent"
        android:layout_height="120dp"
        android:text="@{viewModel.title}"
        tools:text="Sound name"/>
</layout>
```

В операторе привязки можно использовать простые выражения Java, включая сцепленные вызовы функций, математические выражения… и вообще практически все, что вы сочтете нужным включить.

Остается подключить модель представления. Создайте объект SoundViewModel и присоедините его к классу привязки. Затем добавьте функцию привязки в SoundHolder.

Листинг 19.24. Подключение модели представления (MainActivity.kt)

```
private inner class SoundHolder(private val
binding: ListItemSoundBinding) :
        RecyclerView.ViewHolder(binding.root) {

    init {
        binding.viewModel = SoundViewModel()
    }

    fun bind(sound: Sound) {
        binding.apply {
        viewModel?.sound = sound
        executePendingBindings()
        }
    }
}
```

В конструкторе создается и присоединяется модель представления. Затем в функции bind обновляются данные, с которыми работает модель представления.

Вызов executePendingBindings() обычно не нужен. Однако в данном случае данные привязки обновляются в виджете RecyclerView, который обновляет представления с очень высокой скоростью. Вызывая эти функции, вы приказываете макету обновить себя немедленно, вместо того чтобы ожидать одну-две миллисекунды. Таким образом

обеспечивается быстрота реакции RecyclerView синхронно с его RecyclerView.Adapter.

Наконец, завершите подключение модели представления реализацией onBindViewHolder(...).

Листинг 19.25. Вызов функции bind(Sound) (MainActivity.kt)

```
private inner class SoundAdapter(private val
sounds: List<Sound>) :
        RecyclerView.Adapter<SoundHolder>() {
    ...
        override fun onBindViewHolder(holder:
SoundHolder, position: Int) {
        val sound = sounds[position]
        holder.bind(sound)
    }

    override fun getItemCount() = sounds.size
}
```

Запустите приложение. На всех кнопках на экране выводятся названия (рис. 19.11).

Рис. 19.11. Кнопки с заполненными названиями

Отслеживаемые данные

На первый взгляд все хорошо, но в вашем коде кроется проблема. Чтобы убедиться в этом, достаточно прокрутить экран вниз (рис. 19.12).

Рис. 19.12. Что-то знакомое

Видите элемент `67_INDIOS2` вверху? Такой же есть ниже. Прокручивая список вверх и вниз, вы увидите, что имена других файлов появляются в неожиданных, случайных на первый взгляд местах. Если вы этого не видите, поверните устройство в альбомный режим и попробуйте еще раз посмотреть, что к чему.

Не беспокойтесь! Это не сбой в матрице. Это происходит потому, что ваш макет не имеет возможности узнать, что вы обновили объект `Sound`, принадлежащий `SoundViewModel`, внутри функции `SoundHolder.bind(Sound)`. Другими словами, модель представления неправильно переносит данные в ваш макетный файл, как показано на рис. 19.10. Помимо четкого разделения обязанностей, этот момент — тот самый секретный ингредиент, который отличает MVVM от других архитектур, таких как MVC.

Необходимо добавить эту связь. Для этого ваша модель представления должна реализовать интерфейс `Observable` привязки данных. Этот интерфейс позволяет классу привязки установить слушателей для модели представления, чтобы он

мог автоматически получать обратные вызовы при изменении полей.

Реализация всего интерфейса возможна, но это потребует большого объема работы. Мы в Big Nerd Ranch работы не боимся, но стараемся избегать ее, насколько это возможно. Поэтому мы покажем, как решить эту задачу более эффективно — используя привязку данных класса `BaseObservable`.

Этот способ состоит из трех шагов:

1. Создайте подкласс `BaseObservable` в модели представления.

2. Снабдите свойства, используемые в привязке, аннотацией `@Bindable`.

3. Вызывайте `notifyChange()` или `notifyPropertyChanged(int)` при каждом изменении значения свойства привязки.

В `SoundViewModel` вся процедура состоит из нескольких строк кода. Внесите изменения в `SoundViewModel`.

Листинг 19.26. Внесение изменений в модель представления (SoundViewModel.kt)

```
class SoundViewModel : BaseObservable() {

    var sound: Sound? = null
        set(sound) {
            field = sound
            notifyChange()
        }
```

```
    @get:Bindable
    val title: String?
        get() = sound?.name
}
```

Когда вы вызываете функцию `notifyChange()`, она оповещает класс привязки о том, что все `Bindable`-свойства ваших объектов были обновлены. Класс привязки выполняет код внутри скобок `{}` для повторного заполнения представления. Таким образом, при установке значения звука объект `ListItemSoundBinding` получит уведомление и вызовет `Button.setText(String)`, как указано в файле `list_item_sound.xml`.

Выше упоминалась другая функция: `notifyPropertyChanged(int)`. Она делает то же самое, что и `notifyChange()`, но отличается большей точностью. Используя запись `notifyChange()`, вы говорите: «Все `Bindable`-свойства изменились; всё нужно обновить». При использовании `notifyPropertyChanged(BR.title)` сообщается: «Изменилось только значение `title`».

`BR.title` — это константа, которая генерируется библиотекой привязки данных. Имя класса `BR` — это сокращение от binding resource — «связующий ресурс». Каждое свойство, которое вы аннотируете с помощью `@Bindable`, дает результат в сгенерированной константе `BR` с тем же именем.

Несколько других примеров:

```
@get:Bindable val title: String // дает BR.title
@get:Bindable val volume: Int // дает BR.volume
```

```
@get:Bindable val etcetera: String // дает
BR.etcetera
```

Возможно, вы думаете, что использование объектов `Obserable` похоже на использование `LiveData`, о чем вы узнали в главе 11, и вы были бы правы. На самом деле, вы можете использовать `LiveData` с привязкой данных вместо интерфейса Observable. Подробнее об этом вы узнаете в разделе под названием «Для любознательных: LiveData и привязка данных» в конце этой главы.

Снова запустите BeatBox. На этот раз при прокрутке все должно работать нормально (рис. 19.13).

Рис. 19.13. Готовый интерфейс BeatBox

Для любознательных: подробнее о привязке данных

Полноценное описание привязки данных выходит за рамки книги. Тем не менее мы отметим хотя бы некоторые интересные моменты.

Лямбда-выражения

Короткие обратные вызовы можно записывать прямо в файле макета в виде *лямбда-выражений*. Они представляют собой упрощенные версии лямбда-выражений в Java:

```
<Button
    android:layout_width="match_parent"
    android:layout_height="120dp"
    android:text="@{viewModel.title}"
            android:onClick=    "@{(view)    ->
viewModel.onButtonClick()}"
    tools:text="Sound name"/>
```

Как и лямбда-выражения в Java 8, они преобразуются в реализации интерфейса, для которого они используются (в данном случае `View.OnClickListener`). Однако в отличие от лямбда-выражений в Java 8, эти выражения должны использовать строго определенный синтаксис: параметр должен быть заключен в круглые скобки, а в правой части должно стоять только одно выражение.

Кроме того, в отличие от лямбда-выражений Java, можно опустить параметры, если вы их не используете. Таким образом, следующая строка тоже нормально работает:

```
android:onClick="@{()                              ->
viewModel.onButtonClick()}"
```

Синтаксический сахар

Также поддерживаются другие удобные синтаксические конструкции привязки данных. Особенно удобна возможность обозначения двойных кавычек обратными апострофами:

```
android:text="@{`File name: ` + viewModel.title}"
```

Здесь `Filename` означает то же самое, что и "Filename".

Выражения привязки также поддерживают оператор выбора с проверкой на null:

```
android:text="@{`File name: ` + viewModel.title ?? `No file`}"
```

Если значение title равно null, оператор ?? возвращает значение "Nofile".

Кроме того, в привязке данных поддерживается автоматическая обработка null. Даже если в приведенном выше коде viewModel содержит null, привязка данных предоставит необходимые проверки, которые предотвратят сбой приложения. Вместо ошибки подвыражение viewModel.title вернет "null".

BindingAdapter

По умолчанию привязка данных интерпретирует выражение привязки как вызов свойства. Таким образом, конструкция

```
android:text="@{`File name: ` + viewModel.title ?? `No file`}"
```

преобразуется в вызов функции setText(String).

Впрочем, иногда этого оказывается недостаточно, и для какого-то атрибута должно применяться нестандартное поведение. В таких случаях следует написать BindingAdapter:

```
@BindingAdapter("app:soundName")
```

```
fun bindAssetSound(button: Button,
assetFileName: String ) {
    ...
}
```

Просто создайте в любом классе своего проекта функцию уровня файловой системы и снабдите ее аннотацией `@BindingAdapter`, передавая имя привязываемого атрибута в параметре (да, это действительно работает). Каждый раз, когда механизму привязки данных потребуется применить этот атрибут, он вызовет вашу статическую функцию.

Вам достаточно создать функцию на уровне файла в любом классе своего проекта и добавить ей аннотацию `@BindingAdapter`, передавая в качестве параметра имя атрибута, к которому вы хотите привязать аннотацию, и `View`, к которому применяется аннотация, в качестве первого параметра к функции (да, это правда работает).

В приведенном выше примере всякий раз, когда при привязке данных встречается объект `Button` с атрибутом `app:soundName`, содержащим *выражение привязки*, оно будет вызывать вашу функцию, передавая `Button` и результат выражения привязки.

Вы даже можете создавать `BindingAdapter` для более обобщенных представлений, таких как `View` или `ViewGroup`. В этом случае `BindingAdapter` будет применяться к этому виду и всем его подклассам.

Например, если вы хотите определить атрибут `app:isGone`, который будет задавать видимость любого объекта `View`, на основании логического значения, это можно сделать так:

```
@BindingAdapter("app:isGone")
fun bindIsGone(view: View, isGone: Boolean ) {
```

```
        view.visibility = if (isGone) View.GONE
    else View.VISIBLE
    }
```

Так как `View` является первым аргументом для `bindIsGone`, этот атрибут доступен для `View` и всех его подклассов в модуле вашего приложения. Это означает, что вы можете использовать его с `Button`, `TextView`, `LinearLayout` и так далее.

Вероятно, вы легко представите одну-две операции, в которых привязка данных использовалась бы с виджетами стандартной библиотеки. Для многих распространенных операций уже определены адаптеры привязки. Например, `TextViewBindingAdapter` предоставляет дополнительные атрибуты для `TextView`. Информацию о них можно получить прямо при просмотре исходного кода в Android Studio. Прежде чем писать свое собственное решение, попробуйте нажать сочетание клавиш ⌘+⇧+O (**Ctrl+Shift+O**) для поиска класса, откройте связанный с ним файл адаптеров привязки и проверьте, нет ли нужного вам готового класса.

Для любознательных: LiveData и привязка данных

`LiveData` и привязка данных схожи в том, что оба эти метода дают возможность наблюдать за данными и реагировать на их изменения. Фактически можно использовать и `LiveData`, и привязку данных вместе, в тандеме. В примере ниже показаны изменения, которые нужно внести в `SoundViewModel`, чтобы связать свойство `title` как `LiveData`, а не как `Observable`.

```
    class SoundViewModel     : BaseObservable() {

            val title: MutableLiveData<String?> =
    MutableLiveData()
```

```
    var sound: Sound? = null
        set(sound) {
            field = sound
                notifyChange()
                title.postValue(sound?.name)
        }

        @get:Bindable
        val title: String?
            get() = sound?.name
}
```

В данном примере больше нет необходимости выделять подкласс `BaseObservable` или предоставлять аннотации `@Bindable`, так как `LiveData` имеет свои собственные механизмы уведомления наблюдателей. Однако, как вы узнали из главы 11, для `LiveData` требуется объект `LifecycleOwner`. Чтобы указать фреймворку привязки данных, какой владелец жизненного цикла должен использоваться при наблюдении за свойством `title`, вы должны обновить `SoundAdapter`, установив свойство `lifecycleOwner` после создания привязки:

```
private inner class SoundAdapter(private val
sounds: List<Sound>) :
        RecyclerView.Adapter<SoundHolder>() {
    ...
       override fun onCreateViewHolder(parent:
ViewGroup, viewType: Int):
            SoundHolder {
```

```
                                val binding =
DataBindingUtil.inflate<ListItemSoundBinding>(
            layoutInflater,
            R.layout.list_item_sound,
            parent,
            false
        )

                        binding.lifecycleOwner =
this@MainActivity

        return SoundHolder(binding)
    }
}
```

Тогда `MainActivity` станет владельцем жизненного цикла. Никаких изменений в представлении не потребуется, пока имя свойства, `title`, остается прежним.

20. Модульное тестирование и воспроизведение звуков

Одно из достоинств архитектуры MVVM заключается в том, что она упрощает важнейшую практическую задачу программирования: *модульное тестирование*. Под этим термином понимается практика написания небольших программ для проверки автономного поведения отдельных частей (модулей) основного приложения. Так как каждый модуль BeatBox является классом, модульные тесты будут тестировать классы.

В этой главе мы наконец-то воспроизведем файлы `.wav`, загруженные в предыдущей главе. В процессе сборки и интеграции воспроизведения звука мы напишем модульные тесты для интеграции `SoundViewModel` с `BeatBox`.

API для работы со звуком в Android в основном являются низкоуровневыми, но существует инструмент, практически идеально подходящий для нашего приложения: `SoundPool`. Класс `SoundPool` позволяет загрузить в память большой набор звуков и управлять их максимальным количеством, воспроизводимым одновременно. Таким образом, если пользователь войдет в азарт и начнет жать на все кнопки одновременно, это не приведет к сбою приложения или чрезмерному расходованию ресурсов на телефоне.

Готовы? Пора начинать.

Создание объекта SoundPool

Начнем с создания объекта `SoundPool` для воспроизведения звуков в BeatBox.

Листинг 20.1. Создание объекта SoundPool (BeatBox.kt)

```
private const val TAG = "BeatBox"
private const val SOUNDS_FOLDER = "sample_sounds"
private const val MAX_SOUNDS = 5

class BeatBox(private val assets: AssetManager) {

    val sounds: List<Sound>
    private val soundPool = SoundPool.Builder()
        .setMaxStreams(MAX_SOUNDS)
        .build()

    init {
        sounds = loadSounds()
    }
    ...
}
```

Конструктор `Builder` используется для создания экземпляра `SoundPool`. Опция `setMaxStreams(Int)` в конструкторе определяет, сколько звуков может проигрываться в любой момент времени. Здесь их пять. Если воспроизводится пять звуков и вы попытаетесь воспроизвести шестой, то `SoundPool` перестанет играть запущенный раньше других.

В дополнение к опции `MaxStreams` конструктор `SoundPool` позволяет указать различные атрибуты аудиопотока с помощью опции `setAudioAttributes(AudioAttributes)`. Установите флажок документации для получения более подробной информации о том, что это дает. Атрибуты звука по

умолчанию достаточно хорошо работают в этом примере, поэтому вы можете пока оставить эту опцию без внимания.

Доступ к активам

Вспомните, что ваши звуковые файлы хранятся в активах вашего приложения. Перед тем как вы получите доступ к файлам для воспроизведения звука, давайте немного обсудим, как работают активы.

У вашего объекта `Sound` указан путь к файлам активов. Пути к файлам активов не будут работать, если вы попытаетесь открыть их с помощью `File`. Вместо этого нужно использовать `AssetManager`.

```
val assetPath = sound.assetPath

val assetManager = context.assets

val soundData = assetManager.open(assetPath)
```

Вы получите стандартный входной поток для данных, который можете использовать как любой другой `InputStream` в Kotlin.

Некоторым API вместо этого требуется `FileDescriptor`. Именно его мы будем использовать с `SoundPool`. Если нужно, вы можете вызвать `AssetManager.openFd(String)` вместо этого.

```
val assetPath = sound.assetPath

val assetManager = context.assets
```

```
// AssetFileDescriptors отличается от FileDescriptors...
val assetFileDescriptor = assetManager.openFd(assetPath)

// ... но вы можете лего получить обычный FileDescriptor если вам нужен
val fileDescriptor = assetFileDescriptor.fileDescriptor
```

Загрузка звуков

Следующий шаг — загрузка звуков в SoundPool. Главное преимущество класса SoundPool перед другими механизмами воспроизведения звука — быстрая реакция: когда вы приказываете ему воспроизвести звук, воспроизведение начинается немедленно, без задержки.

За это приходится расплачиваться необходимостью загрузки звуков в SoundPool перед воспроизведением. Каждому загружаемому звуку назначается собственный целочисленный идентификатор. Добавьте свойство soundId в Sound.

Листинг 20.2. Добавление свойства идентификатора звука (Sound.kt)

```
class Sound(val assetPath: String, var soundId: Int? = null) {
    val name = assetPath.split("/").last().removeSuffix(WAV)
}
```

Объявление soundId с обнуляемым типом (Int?) позволяет представить неопределенное состояние Sound — для этого

soundId присваивается значение null.

Теперь можно переходить к загрузке звуков. Добавьте в BeatBox функцию load(Sound) для загрузки Sound в SoundPool.

Листинг 20.3. Загрузка звуков в SoundPool (BeatBox.kt)

```
class BeatBox(private val assets: AssetManager)
{
    ...
    private fun loadSounds(): List<Sound> {
        ...
    }

    private fun load(sound: Sound) {
            val afd: AssetFileDescriptor =
assets.openFd(sound.assetPath)
        val soundId = soundPool.load(afd, 1)
        sound.soundId = soundId
    }
}
```

Вызов soundPool.load(AssetFileDescriptor,Int) загружает файл в SoundPool для последующего воспроизведения. Для управления звуком, его повторного воспроизведения (или выгрузки) soundPool.load(...) возвращает идентификатор типа int, который сохраняется в только что определенном поле soundId.

А так как вызов openFd(String) инициирует IOException, load(Sound) тоже инициирует IOException. Это означает, что вам придется обрабатывать это исключение при каждом вызове функции load(Sound).

Загрузите все звуки, вызывая функцию load(Sound) в BeatBox.loadSounds().

Листинг 20.4. Загрузка всех звуков (BeatBox.kt)

```
private fun loadSounds(): List<Sound> {
    ...
    val sounds = mutableListOf<Sound>()
    soundNames.forEach { filename ->
                            val   assetPath   =
"$SOUNDS_FOLDER/$filename"
        val sound = Sound(assetPath)
           sounds.add(sound)
        try {
            load(sound)
            sounds.add(sound)
        } catch (ioe: IOException) {
              Log.e(TAG, "Cound not load sound
$filename", ioe)
        }
    }
    return sounds
}
```

Запустите приложение BeatBox и убедитесь в том, что все звуки были загружены правильно. Если при загрузке произошла ошибка, на панели **LogCat** появятся красные сообщения об исключениях.

Воспроизведение звуков

Остается последний шаг: воспроизвести загруженные звуки. Добавьте в BeatBox функцию play(Sound).

Листинг 20.5. Воспроизведение звуков (BeatBox.kt)

```
class BeatBox(private val assets: AssetManager)
{
    ...
    init {
        sounds = loadSounds()
    }

    fun play(sound: Sound) {
        sound.soundId?.let {
            soundPool.play(it, 1.0f, 1.0f, 1,
0, 1.0f)
        }
    }
    ...
}
```

Прежде чем воспроизводить звук с идентификатором soundId, необходимо сначала убедиться в том, что он отличен от null. Такое возможно, если объект Sound не удалось загрузить.

Если вы уверены, что значение отлично от null, воспроизведите звук вызовом SoundPool.play(int,float,float,int,int,float). Параметры содержат соответственно: идентификатор звука, громкость слева, громкость справа, приоритет (игнорируется), признак циклического воспроизведения и скорость воспроизведения. Для полной громкости и нормальной

скорости воспроизведения передайте `1.0`. Передача `0` в признаке циклического воспроизведения означает «без зацикливания». (Передайте `-1`, если хотите, чтобы воспроизведение длилось бесконечно долго. Мы считаем, что это только раздражает.)

После написания такой функции вы сможете интегрировать воспроизведение звука в `SoundViewModel`. Мы выполним эту интеграцию по принципу «сначала тесты» — иначе говоря, сначала пишется модульный тест для неудачного случая, и только потом реализуется интеграция, обеспечивающая его прохождение.

Зависимости при тестировании

Прежде чем писать тест, необходимо добавить в среду тестирования пару инструментов: Mockito и Hamcrest. Mockito — Java-фреймворк, упрощающий создание простых *тестовых объектов*. Эти объекты обеспечивают изоляцию тестов `SoundViewModel`, чтобы тестирование разных объектов не проводилось одновременно.

Hamcrest — библиотека для *проверки условий* в коде и инициирования сбоев в случае нарушения этих условий. Hamcrest позволяет убедиться в том, что ваш код работает так, как вы ожидаете.

Версия Hamcrest автоматически включается в библиотеку JUnit, а JUnit автоматически включается в качестве зависимости при создании нового проекта в Android Studio. Это означает, что вам остается лишь добавить в вашу тестовую сборку зависимости для Mockito. Откройте файл `build.gradle` вашего модуля приложения и добавьте зависимости для Mockito (листинг 20.6). Синхронизируйте файлы, когда закончите.

Листинг. 20.6. Добавление зависимости Mockito (app/build.gradle)

```
dependencies {
    ...
                                    implementation
'androidx.recyclerview:recyclerview:1.0.0'
      testImplementation 'org.mockito:mockito-
core:2.25.0'
      testImplementation 'org.mockito:mockito-
inline:2.25.0'
}
```

Область применения `testImplementation` означает, что эта зависимость включается только в тестовые сборки вашего приложения. Благодаря этому в вашем APK не будет лишнего неиспользуемого кода.

Библиотека `mockito-core` включает в себя все функции, которые вы будете использовать для создания и настройки ваших поддельных объектов.

Библиотека `mockito-inline` — это специальная зависимость, которая делает Mockito проще в использовании с Kotlin.

По умолчанию все классы Kotlin являются последними в иерархии наследования. Это означает, что вы не можете наследовать от этих классов, если вы явно не пометили их как открытые. К сожалению, Mockito активно использует наследование при создании болванок. Это означает, что Mockito не может притворяться классами Kotlin без дополнительных настроек. Зависимость `mockito-inline` дает функциональность, которая позволяет Mockito наследовать от таких классов и функций, работая вокруг этой проблемы

наследования. Это позволяет «пародировать» классы Kotlin без необходимости изменять исходные файлы.

Создание класса теста

Модульные тесты удобнее всего создавать при помощи *тестового фреймворка*. Фреймворк упрощает написание и запуск пакетов тестов, и просмотр их вывода в Android Studio.

Практически все программисты используют при Android-программировании тестовый фреймворк JUnit. В JUnit предусмотрены удобные средства интеграции с Android Studio. Работа начинается с создания класса, в котором должны находиться тесты JUnit. Для этого откройте файл `SoundViewModel.kt` и нажмите сочетание клавиш ⌘+⇧+T (**Ctrl+Shift+T**). Android Studio пытается перейти к классу теста, связанному с тем классом, который вы просматриваете. Если класс теста не найден (как в нашем случае), вам будет предложено создать новый класс теста (рис. 20.1).

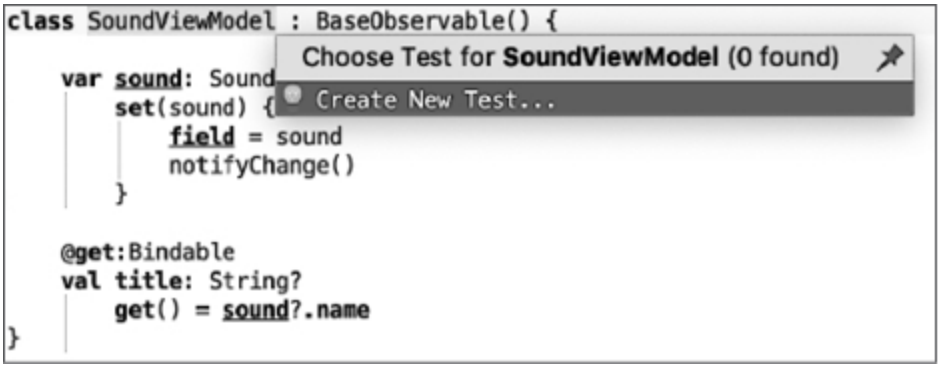


Рис. 20.1. Попытка открытия класса теста

Выберите команду **CreateNewTest**, чтобы создать новый класс теста. Выберите тестовую библиотеку **JUnit4** и установите флажок **setUp/@Before**. Оставьте остальные параметры без изменений (рис. 20.2).

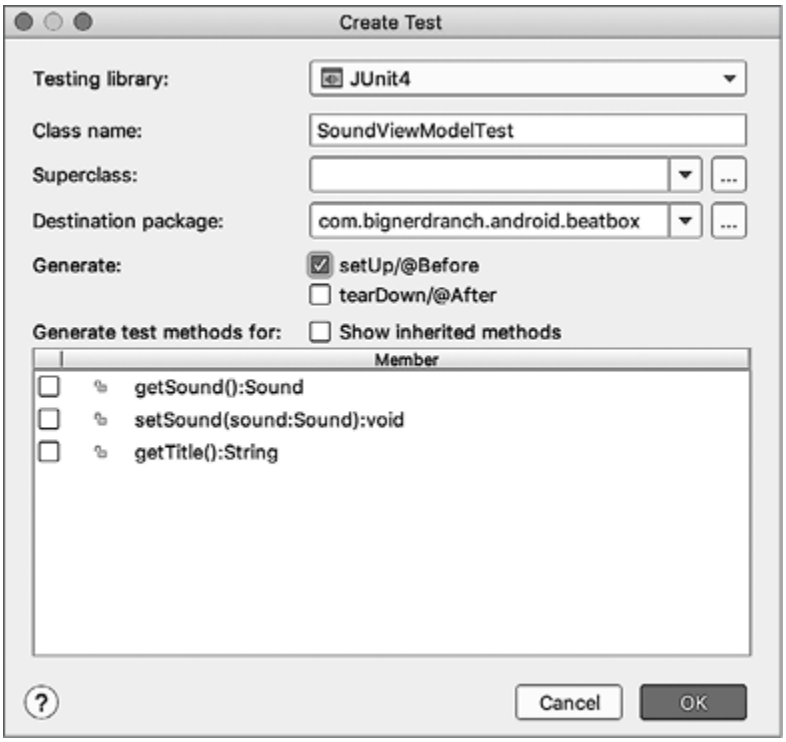


Рис. 20.2. Создание нового класса теста

Нажмите кнопку **OK**, чтобы перейти к следующему диалоговому окну.

Остается выбрать тип класса теста. Тесты в папке `androidTest` относятся к категории *инструментальных тестов*. Инструментальные тесты работают на устройстве Android или в эмуляторе. Преимущество такого решения заключается в том, что вы можете протестировать любые аспекты поведения вашего приложения во время выполнения; недостаток — в том, что тесты работают в полной версии операционной системы Android, из-за чего их выполнение может занимать много времени.

Тесты в папке `test` относятся к категории модульных тестов. Модульные тесты выполняются на локальной машине и не требуют наличия исполнительной среды Android. Отказ от балласта ускоряет их выполнение.

Термин «модульный тест» на Android используется слишком обширно: иногда для описания типа теста, который проверяет один класс или единицу функциональности по отдельности, в других случаях — для описания любого теста, находящегося в каталоге тестов. Даже если тест в этом каталоге может проверять один класс или единицу функциональности, он может быть *интеграционнымтестом*, который тестирует раздел приложения совокупно с множеством других частей. Вы больше узнаете об интеграционных тестах в разделе «Для любознательных: интеграционное тестирование» в конце этой главы.

В оставшейся части этой главы мы будем использовать термин *JVM-тест* для теста любого типа, который расположен в папке `test` и работает на JVM. Мы будем называть модульными только те тесты, которые проверяют один класс или модуль.

Модульные тесты содержат минимум тестового кода: тест одного компонента. Для их запуска не требуется все приложение или устройство, и они должны выполняться достаточно быстро для многократного проведения тестирования в ходе работы. Из-за этого они редко используются в качестве инструментальных тестов. Выберите папку `test` для своего класса теста (рис. 20.3) и нажмите кнопку **OK**.

Подготовка теста

На следующем шаге строится тест `SoundViewModel`. Android Studio создает класс `SoundViewModelTest.kt`. (Вы можете найти этот класс с меткой `test` внутри вашего модуля приложения.) Шаблон начинается с вызова единственной функции `setUp()`:

```
class SoundViewModelTest {

    @Before
    fun setUp() {
    }
}
```

Рис. 20.3. Выбор целевого каталога

Для большинства объектов тест должен делать одно и то же: собирать экземпляр объекта для тестирования и создавать другие объекты, от которых зависит этот объект. Вместо того чтобы писать одинаковый код для всех тестов, JUnit предоставляет аннотацию `@Before`. Код, содержащийся в функции с пометкой `@Before`, будет выполнен один раз перед выполнением каждого теста. По действующим соглашениям большинство классов модульных тестов содержит одну функцию `setUp()` с пометкой `@Before`.

Настройка тестируемых объектов

Внутри функции `setUp()` вы конструируете экземпляр `SoundViewModel` для тестирования. Для этого вам нужен экземпляр `Sound`, потому что `SoundViewModel` необходим звуковой объект, позволяющий понять, как отображать заголовок.

Создайте объект `SoundViewModel` и `Sound` для его использования (листинг 20.7). Так как `Sound` является простым объектом данных без сложного поведения, его можно не макетировать.

Листинг 20.7. Создание тестируемого объекта SoundViewModel (SoundViewModelTest.kt)

```
class SoundViewModelTest {

    private lateinit var sound: Sound
    private lateinit var subject: SoundViewModel

    @Before
    fun setUp() {
        sound = Sound("assetPath")
        subject = SoundViewModel()
        subject.sound = sound
    }
}
```

На любой другой странице этой книги вы бы назвали `SoundViewModel` именем `soundViewModel`. Здесь, однако, используем имя `subject`. Это имя мы любим использовать в наших тестах на Big Nerd Ranch по двум причинам:

- так понятно, что `subject` — это тестируемый объект (а другие объекты — нет);
- если какие-либо функции `SoundViewModel` когда-либо будут перемещены в другой класс (скажем, `BeatBoxSoundViewModel`), тестовые функции можно будет скопировать/вставить без переименования `soundViewModel` в `beatBoxSoundViewModel`.

Написание тестов

Итак, функция `setUp()` готова; можно переходить к написанию тестов. Тест представляет собой функцию в тестовом классе, помеченную аннотацией `@Test`.

Начните с написания теста, который проверяет существующее поведение в `SoundViewModel`: свойство `title` связывается со свойством `name` объекта `Sound`. Напишите функцию для тестирования этого поведения (листинг 20.8).

Листинг 20.8. Тестирование свойства title (SoundViewModelTest.kt)

```
class SoundViewModelTest {
    ...
    @Before
    fun setUp() {
        ...
    }

    @Test
    fun exposesSoundNameAsTitle() {
```

```
                assertThat(subject.title,
`is`(sound.name))
        }
}
```

Две функции выделяются красным цветом: `assertThat(...)` и `is(...)`. Нажмите сочетание клавиш ⌥+⏎ (**Alt+Enter**) в `assertThat(...)` и выберите `Assert.assertThat(...)` из `org.junit`. Сделайте то же самое для функции `is(...)`, но на этот раз выберите `Is.is` из `org.hamcrest`.

Тест проверяет функцию `is(...)` с помощью библиотеки Hamcrest с JUnit-функцией `assertThat(...)`. Этот код читается почти как предложение: «Положим, что свойство заголовка это субъекта будет тем же, что и имя звука». Если две функции возвращают разные значения, тест не пройдет.

Чтобы выполнить модульные тесты, щелкните правой кнопкой мыши по имени класса `SoundViewModelTest` и выберите команду **Run'SoundViewModelTest'** в контекстном меню. В окне Android Studio появится результат (рис. 20.4).

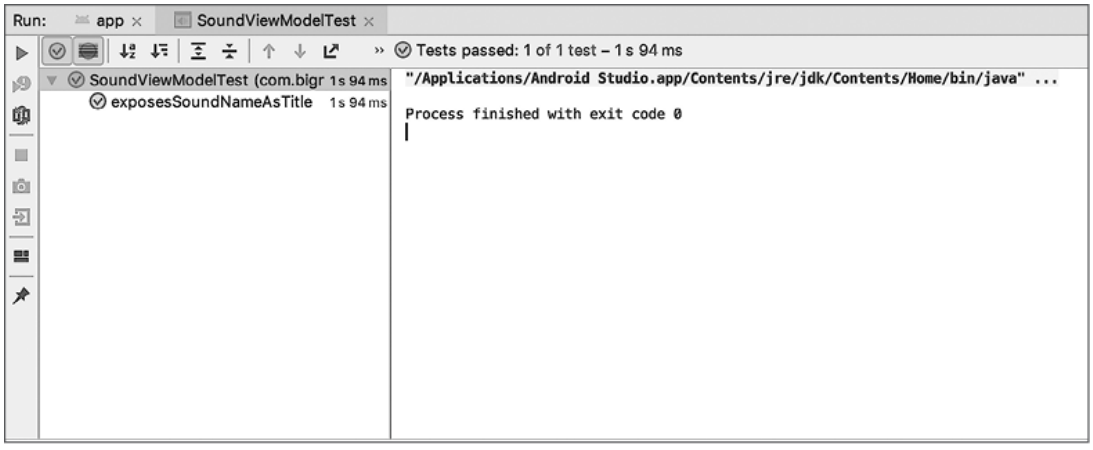


Рис. 20.4. Тесты прошли успешно

По умолчанию в результатах тестов выводится информация только о пройденных тестах, потому что лишь они представляют интерес для разработчика. Приведенный результат означает, что все прошло замечательно — тесты выполнены, и они прошли успешно.

Взаимодействия тестируемых объектов

А теперь займемся настоящей работой: организацией взаимодействия между `SoundViewModel` и новой функцией `BeatBox.play(Sound)`. Обычно для этого пишется тест, который показывает, какое поведение ожидается от новой функции, причем делается это *до* написания самой функции. Мы напишем новую функцию класса `SoundViewModel` с именем `onButtonClicked()`, которая вызывает `BeatBox.play(Sound)`. Напишите тестовую функцию, которая вызывает `onButtonClicked()` (листинг 20.9).

Листинг 20.9. Написание теста для onButtonClicked() (SoundViewModelTest.kt)

```
class SoundViewModelTest {
    ...
    @Test
    fun exposesSoundNameAsTitle() {
                        assertThat(subject.title,
`is`(sound.name))
    }

    @Test
    fun callsBeatBoxPlayOnButtonClicked() {
        subject.onButtonClicked()
```

```
    }
}
```

Функция еще не существует, поэтому она выделяется красным цветом. Наведите на нее указатель мыши и нажмите сочетание клавиш ⌥+⏎ (**Alt+Enter**). Выберите команду **Creatememberfunction'SoundViewModel.onButtonClicked'**; функция будет сгенерирована автоматически.

Листинг 20.10. Создание onButtonClicked() (SoundViewModel.kt)

```
class SoundViewModel : BaseObservable() {
    fun onButtonClicked() {
            TODO("not implemented") // Изменить
...
    }
    ...
}
```

Пока оставьте функцию пустой и нажмите сочетание клавиш ⌘+⇧+T (**Ctrl+Shift+T**), чтобы вернуться к `SoundViewModelTest`.

Ваш тест вызывает функцию и проверяет, делает ли она то, что ей сказано: вызывает `BeatBox.play(Sound)`. Первым шагом к реализации является предоставление `SoundViewModel` объекта `BeatBox`.

Вы можете создать экземпляр `BeatBox` в вашем тесте и передать его конструктору модели представления. Не сделайте это в модульном тесте, иначе возникнет проблема: если `BeatBox` работает неправильно, то тесты, которые вы пишете в `SoundViewModel` и которые используют `BeatBox`, тоже могут не работать должным образом. Так не пойдет. Модульные тесты `SoundViewModel` должны сбоить только тогда, когда не функционирует `SoundViewModel`.

Другими словами, вы хотите изолированно проверить поведение SoundViewModel и его взаимодействие с другими классами. Это ключевой принцип модульного тестирования.

Решение — использование *имитации*BeatBox. Эта имитация будет подклассом BeatBox, которая имеет все те же функции, что и BeatBox, но ни одна из этих функций ничего не делает. Таким образом, ваш тест SoundViewModel может проверить, что SoundViewModel использует BeatBox правильно, независимо от самой BeatBox.

Чтобы создать объект для имитации с помощью Mockito, вы вызываете статическую функцию mock(Class), передавая класс, который хотите сымитировать. Создайте имитацию экземпляра BeatBox и поле для его хранения в SoundViewModelTest (листинг 20.11).

Листинг 20.11. Создание имитации BeatBox (SoundViewModelTest.kt)

```
class SoundViewModelTest {

    private lateinit var beatBox: BeatBox
    private lateinit var sound: Sound
            private lateinit var subject:
SoundViewModel

    @Before
    fun setUp() {
        beatBox = mock(BeatBox::class.java)
        sound = Sound("assetPath")
        subject = SoundViewModel()
        subject.sound = sound
```

```
    }
    ...
}
```

Функция `mock(Class)` должна быть импортирована, как и ссылка на класс. Эта функция автоматически создаст для вас имитацию `BeatBox`. Довольно хитро.

Когда имитация `BeatBox` готова, вы можете закончить написание теста, чтобы убедиться, что вызвана функция воспроизведения. Mockito может помочь вам в этой сложной задаче. Все объекты Mockito отслеживают, какие из их функций вызывались и какие параметры передавались при каждом вызове. Функция `verify(Object)` объекта Mockito проверяет, вызывались ли эти функции так, как вы ожидали.

Вызовите функцию `verify(Object)`, чтобы убедиться в том, что `onButtonClicked()` вызывает `BeatBox.play(Sound)` с объектом `Sound`, связанным с `SoundViewModel`.

Листинг 20.12. Проверка вызова BeatBox.play(Sound) (SoundViewModelTest.kt)

```
class SoundViewModelTest {
    ...
    @Test
    fun callsBeatBoxPlayOnButtonClicked() {
        subject.onButtonClicked()

        verify(beatBox).play(sound)
    }
}
```

Функция `verify(Object)` использует динамический интерфейс и включает следующий код:

```
verify(beatBox)
beatBox.play(sound)
```

Вызов `verify(beatBox)` означает: «Я хочу проверить, что для `beatBox` была вызвана функция». Следующий вызов функции интерпретируется так: «Проверить, что эта функция был вызвана именно так». Таким образом, вызов `verify(...)` означает: «Проверить, что функция `play(...)` была вызвана для `beatBox` с передачей `sound` в качестве параметра».

Конечно, ничего подобного не происходило. Код `SoundViewModel.onButtonClicked()` пуст, так что функция `beatBox.play(Sound)` не вызывалась. А следовательно, тест *не пройдет*. При опережающем написании тестов это хорошо — если тест проходит с первого раза, то он ничего не проверяет.

Запустите тест и убедитесь в том, что он не проходит. Используйте действия, описанные выше, или нажмите сочетание клавиш ⌘+R (**Ctrl+R**) для повторения последней выполненной команды запуска. Результат показан на рис. 20.5.

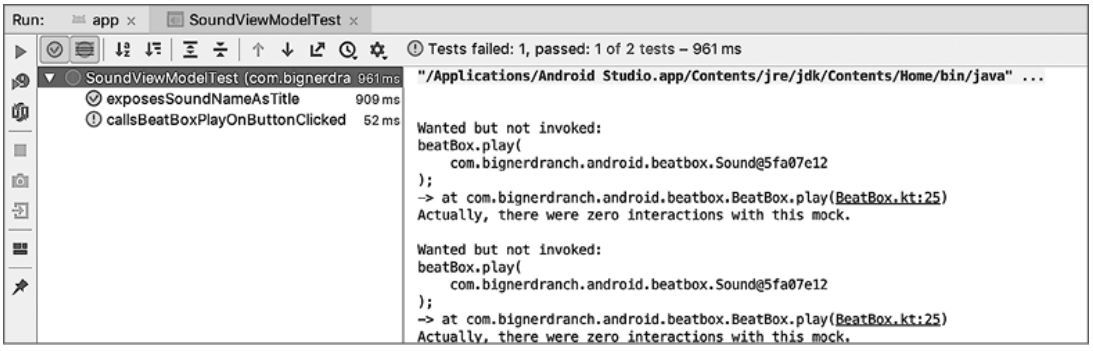


Рис. 20.5. Тест не пройден

В результатах указано, что тест ожидает вызова `beatBox.play(Sound)`, но не получает его:

```
Wanted but not invoked:
beatBox.play(
    com.bignerdranch.android.beatbox.Sound@3571
b748
);
->                                                  at
….callsBeatBoxPlayOnButtonClicked(SoundViewMode
lTest.java:28)
Actually, there were zero interactions with
this mock.
```

Во внутренней реализации `verify(Object)` проверяет условие, как и `assertThat(...)`. Условие оказалось не выполненным, тест не прошел, и был выдан результат с описанием проблемы.

Теперь нужно привести тест в порядок. Во-первых, создайте свойство конструктора для `SoundViewModel`, принимающее `BeatBox`.

Листинг 20.13. Предоставление BeatBox для SoundViewModel (SoundViewModel.kt)

```
class SoundViewModel(private val beatBox:
BeatBox) : BaseObservable() {
    ...
}
```

Это изменение приведет к двум ошибкам в вашем коде: одной в ваших тестах и одной в производстве. Сначала исправьте ошибку производства. Откройте файл `MainActivity.kt` и предоставьте объект `beatBox` вашим моделям представления, когда они будут созданы в `SoundHolder` (листинг 20.14).

Листинг 20.14. Исправление ошибки в SoundHolder (MainActivity.kt)

```
private inner class SoundHolder(private val
binding: ListItemSoundBinding) :
    RecyclerView.ViewHolder(binding.root) {

    init {
                              binding.viewModel =
SoundViewModel(beatBox)
    }

    fun bind(sound: Sound) {
        ...
    }
}
```

Далее предоставьте имитацию `BeatBox` для вашей модели представления в вашем тестовом классе (листинг 20.15).

Листинг 20.15. Предоставление имитации BeatBox в тестовом классе (SoundViewModelTest.kt)

```
class SoundViewModelTest {
    ...
    @Before
    fun setUp() {
        beatBox = mock(BeatBox::class.java)
        sound = Sound("assetPath")
        subject = SoundViewModel(beatBox)
        subject.sound = sound
    }
```

```
    ...
}
```

Тест почти готов. Теперь реализуйте функцию `onButtonClicked()`, чтобы она делала то, чего ожидает тест (листинг 20.16).

Листинг 20.16. Реализация onButtonClicked() (SoundViewModel.kt)

```
class SoundViewModel(private val beatBox:
BeatBox) : BaseObservable() {
    ...
    fun onButtonClicked() {
        sound?.let {
            beatBox.play(it)
        }
    }
}
```

Снова выполните тест. На этот раз зеленый индикатор указывает на то, что все тесты прошли успешно (рис. 20.6).

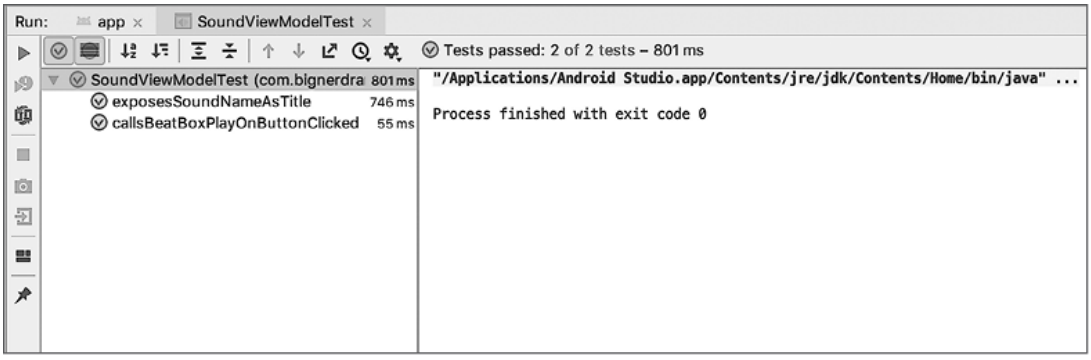


Рис. 20.6. Зеленый индикатор успешного выполнения

Обратные вызовы привязки данных

Чтобы кнопки заработали, осталось сделать последний шаг: связать функцию `onButtonClicked()` с кнопкой.

По аналогии с тем, как вы использовали привязку данных для размещения данных в пользовательском интерфейсе, вы также можете воспользоваться ей для подключения слушателей щелчков и т.д. при помощи лямбда-выражений. (Если вы забыли, что это такое, обратитесь к разделу «Лямбда-выражения» главы 19.)

Добавьте выражение обратного вызова для связывания кнопки с функцией `SoundViewModel.onButtonClicked()`.

Листинг 20.17. Подключение кнопки (res/layout/list_item_sound.xml)

```
<Button
    android:layout_width="match_parent"
    android:layout_height="120dp"
                    android:onClick="@{()     ->
viewModel.onButtonClicked()}"
    android:text="@{viewModel.title}"
    tools:text="Sound name"/>
```

При следующем запуске BeatBox кнопки будут успешно воспроизводить звуки. При этом попытка запустить BeatBox зеленой кнопкой снова приведет к выполнению тестов. Дело в том, что щелчок правой кнопкой мыши изменил *конфигурацию выполнения* — настройку, которая определяет, что должна делать среда Android Studio при нажатии кнопки запуска.

Чтобы запустить приложение BeatBox, щелкните мышью по селектору конфигурации выполнения рядом с кнопкой запуска и вернитесь к конфигурации запуска приложения (рис. 20.7).

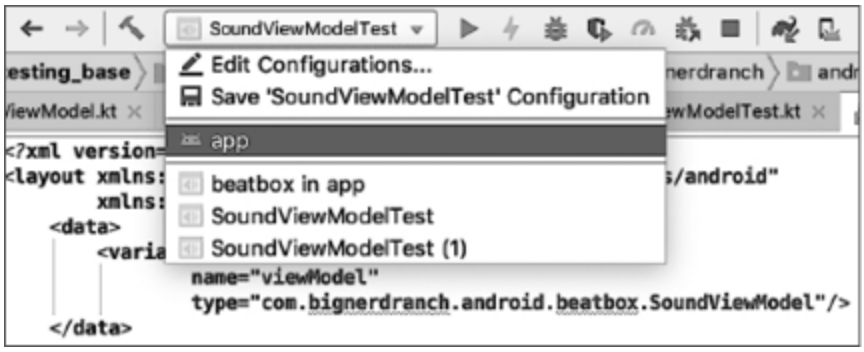


Рис. 20.7. Изменение конфигурации выполнения

Запустите BeatBox и поэкспериментируйте с нажатием кнопок. Приложение будет издавать всевозможные угрожающие звуки. Не пугайтесь — оно для этого и создавалось.

Выгрузка звуков

Приложение работает, но мы еще должны прибрать за собой. Корректное приложение должно освободить ресурсы SoundPool вызовом SoundPool.release() после завершения работы (листинг 20.18).

Листинг 20.18. Освобождение SoundPool (BeatBox.kt)

```
class BeatBox(private val assets: AssetManager)
{
    ...
    fun play(sound: Sound) {
        ...
    }

    fun release() {
        soundPool.release()
    }
```

```
    private fun loadSounds(): List<Sound> {
        ...
    }
    ...
}
```

Добавьте соответствующую функцию `BeatBox.release()` в `MainActivity` (листинг 20.19).

Листинг 20.19. Освобождение BeatBox (BeatBoxFragment.kt)

```
class MainActivity : AppCompatActivity() {

    private lateinit var beatBox: BeatBox

     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
    }

    override fun onDestroy() {
        super.onDestroy()
        beatBox.release()
    }
    ...
}
```

Запустите приложение и убедитесь в том, что оно правильно работает с новой функцией `release()`. Если при воспроизведении длительного звука вы повернете устройство или нажмете кнопку «Назад», звучание должно прекратиться.

Для любознательных: интеграционное тестирование

Как вы помните, упоминавшийся ранее тест `SoundViewModelTest` был модульным. Также вам предоставлялась другая возможность: создать интеграционный тест. Собственно, что это — интеграционный тест?

При единичном тестировании тестируемое изделие относится к отдельному классу. В интеграционном тесте тестируемый субъект — это раздел вашего приложения, многие части которого работают вместе. Оба эти теста важны и служат разным целям. Модульные тесты гарантируют правильное поведение каждого класса и оправдывают ваши ожидания относительно того, как эти предметы будут взаимодействовать друг с другом. Интеграционные тесты проверяют, что протестированные единицы действительно интегрируются правильно и функционируют так, как ожидается.

Интеграционные тесты используются для частей вашего приложения, не относящихся к пользовательскому интерфейсу, например для взаимодействия с базами данных, но на Android они чаще всего пишутся для тестирования приложения на уровне пользовательского интерфейса путем взаимодействия с его интерфейсом и проверки ожиданий. Обычно они пишутся экран за экраном. Например, вы можете протестировать, что при запуске экрана `MainActivity` на первой кнопке отображается имя первого файла из папки `sample_sounds`: `65_cjipie`.

Для интеграционного тестирования по отношению к пользовательскому интерфейсу требуется наличие классов фреймворка, таких как activity и фрагменты. Для него также могут потребоваться системные сервисы, файловые системы и другие фрагменты, недоступные для JVM-тестирования. По этой причине интеграционные тесты на Android чаще всего реализуются в виде инструментальных тестов.

Интеграционные тесты считаются пройденными, когда приложение *делает* то, что вы хотите, а не тогда, когда приложение *реализовано* так, как вы хотите. Изменение идентификатора кнопки не повлияет на работу приложения, но если вы напишете интеграционный тест «Вызвать `findViewById(R.id.button)` и убедиться в том, что на найденной кнопке выводится правильный текст», этот тест не будет пройден. Итак, вместо стандартных средств Android (таких как `findViewById(int)`) интеграционные тесты чаще пишутся во фреймворках тестирования пользовательского интерфейса, в которых проще выражаются запросы вида «Убедиться в том, что на экране присутствует кнопка с текстом, который на ней должен выводиться».

Espresso — фреймворк компании Google для тестирования пользовательского интерфейса Android-приложений. Чтобы включить его, добавьте `com.android.support.test.espresso:espresso-core` как артефакт `androidTestImplementation` в файл `app/build.gradle`.

После добавления в состав зависимостей вы можете использовать Espresso для проверки условий относительно запущенной activity. Следующий пример показывает, как проверить существование на экране представления с первым именем файла из `sample_sounds`:

```
@RunWith(AndroidJUnit4::class)
class MainActivityTest {

    @get:Rule
                    val     activityRule     =
ActivityTestRule(MainActivity::class.java)
```

```
    @Test
    fun showsFirstFileName() {
        onView(withText("65_cjipie"))
            .check(matches(isDisplayed()))
    }
}
```

Работа этого кода основана на паре аннотаций. Аннотация `@RunWith(AndroidJUnit4.class)` указывает, что перед нами тест Android, который должен работать с activity и другими средствами времени выполнения Android. После этого аннотация `@get:Rule` у `activityRule` сообщает JUnit о необходимости запускать экземпляр `MainActivity` перед запуском каждого теста.

После предварительной подготовки вы сможете проверять условия, относящиеся к `MainActivity`, в ваших тестах. В функции `showsFirstFileName()` строка `onView(withText("65_cjipie"))` находит представление с текстом `65_cjipie` для выполнения тестовой операции. Вызов `check(matches(isDisplayed()))` проверяет, что такое представление существует. Если представления с таким текстом нет, то проверка завершается неудачей. Функция `check(...)` используется Espresso для проверки условий типа `assertThat(...)` относительно представлений.

Часто требуется щелкнуть на представлении и проверить некоторое условие относительно результата щелчка. Также можно воспользоваться Espresso для моделирования щелчков на представлениях или других взаимодействий с ними:

```
onView(withText("65_cjipie"))
        .perform(click())
```

Когда вы взаимодействуете с представлением, Espresso дождется *бездействия* приложения, перед тем как продолжать тестирование. В Espresso существуют встроенные средства проверки завершения обновлений пользовательского интерфейса, но если вам потребуется более продолжительное ожидание — используйте подкласс `IdlingResource` для передачи Espresso сигнала о том, что приложение еще не завершило свои операции.

За дополнительной информацией о том, как использовать Espresso для манипуляций с пользовательским интерфейсом и его тестирования, обращайтесь к документации Espresso по адресу **developer.android.com/training/testing/espresso**.

Интеграционные и модульные тесты служат разным целям. Как правило, разработчики предпочитают начинать с модульных тестов, потому что они выполняются достаточно быстро; их можно запускать без долгих раздумий, что упрощает выработку привычки. Интеграционные тесты выполняются достаточно долго, что не позволяет выполнять их с такой частотой. При этом каждый вид тестов дает важную информацию о состоянии вашего приложения под особым углом, поэтому действительно хорошие разработчики выполняют тесты обоих типов.

Для любознательных: фиктивные объекты и тестирование

Фиктивные объекты (имитации) играют в интеграционном тестировании совершенно иную роль, чем в модульном тестировании. Они выдают себя за другие несвязанные компоненты и тем самым обеспечивают изоляцию тестируемых компонентов. Модульные тесты тестируют отдельные классы; каждый класс обладает собственными специфическими зависимостями, поэтому каждый класс теста

использует свой набор фиктивных объектов. Так как фиктивные объекты различаются между классами тестов, а поведение особой роли не играет, для модульных тестов замечательно подходят фреймворки, упрощающие создание простых фиктивных объектов (такие как Mockito).

С другой стороны, интеграционные тесты предназначены для тестирования всего приложения как единого целого. Вместо того чтобы обеспечивать изоляцию компонентов приложения, фиктивные объекты используются для изоляции приложения от всех внешних объектов, с которыми может взаимодействовать приложение, — например, для обеспечения веб-службы фиктивными данными и откликами на запросы. В приложении BeatBox можно было бы предоставить фиктивный объект `SoundPool`, который сообщал бы о воспроизведении конкретного звукового файла. Так как фиктивные объекты имеют большие размеры и совместно используются многими тестами и поскольку они чаще применяются для реализации фиктивного поведения, при интеграционном тестировании лучше избегать использования автоматизированных фреймворков и вместо этого писать фиктивные объекты вручную.

В любом случае действует один принцип: моделирование сущностей на границе тестируемого компонента. Он определяет область действия теста и гарантирует, что тест не будет проходить только в случае неработоспособности самого компонента.

Упражнение. Управление скоростью воспроизведения

В этом упражнении вы добавите в BeatBox элемент управления скоростью воспроизведения, который значительно расширит репертуар воспроизводимых звуков (рис. 20.8). В

MainActivity подключите виджет SeekBar для управления значением скорости, передаваемым вызову play(Int,Float,Float,Int,Int,Float) класса SoundPool.

Рис. 20.8. BeatBox с возможностью управления скоростью воспроизведения

Чтобы это получилось, изучите документацию по адресу **developer.android.com/reference/android/widget/SeekBar.html**.

Упражнение. Воспроизведение звуков после поворота

Сейчас приложение BeatBox прекращает воспроизведение звука при повороте устройства. В этом упражнении мы это

исправим.

Проблема заключается в том, где именно хранится объект `BeatBox`. Ваш `MainActivity` содержит ссылку на ваш `BeatBox`, но он уничтожен и воссоздан во время вращения. Это означает, что ваш первоначальный `BeatBox` освобождает `SoundPool` и воссоздает его каждый раз при вращении.

Вы уже видели, как сохранять информацию во время вращения в GeoQuiz и CriminalIntent. Добавьте модель Jetpack-класса `ViewModel` в BeatBox, чтобы ваш объект `BeatBox` сохранялся после поворота.

Вы можете получить доступ к объекту `BeatBox` как публичному свойству из `ViewModel`. Таким образом, `MainActivity` может получить доступ к экземпляру `BeatBox` из Jetpack-класса `ViewModel`, чтобы передать его модели представления привязки данных.

21. Стили и темы

Приложение BeatBox звучит эффектно; пора придать ему столь же эффектный внешний вид.

До настоящего момента в BeatBox использовалось стандартное стилевое оформление пользовательского интерфейса. Кнопки — стандартные. Цвета — стандартные. Приложение ничем не выделяется на общем фоне. У него нет своего «лица».

К счастью, ситуацию можно изменить. У нас для этого имеются подходящие технологии.

На рис. 21.1 изображено заметно улучшенное (или по крайней мере более стильное) приложение BeatBox.

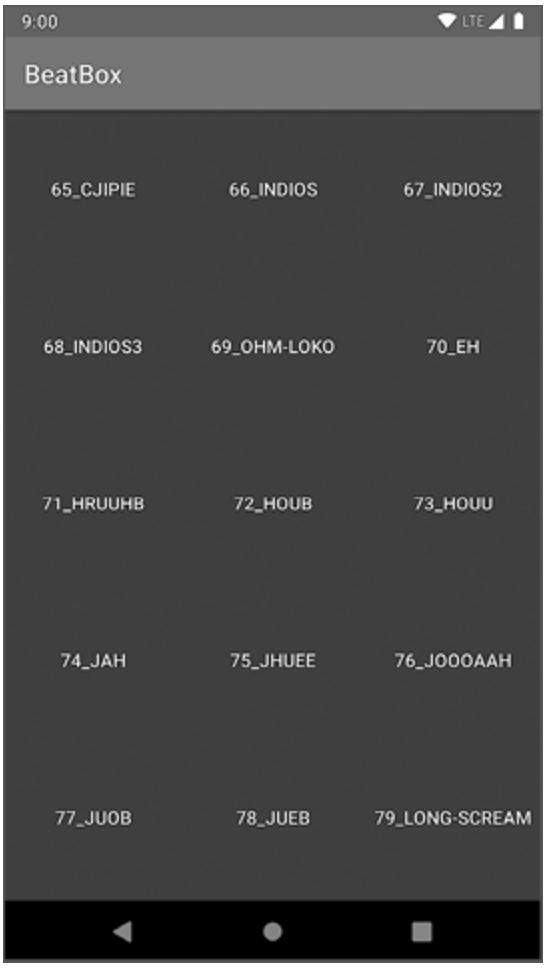


Рис. 21.1. Приложение BeatBox с новым оформлением

Цветовые ресурсы

Начнем с определения нескольких цветов, которые будут использоваться в этой главе. Создайте файл `colors.xml` в папке `res/values`.

Листинг 21.1. Определение цветов (res/values/colors.xml)

```
<resources>
```

```
    <color name="colorPrimary">#008577</color>
                                        <color
name="colorPrimaryDark">#00574B</color>
    <color name="colorAccent">#D81B60</color>

    <color name="red">#F44336</color>
    <color name="dark_red">#C3352B</color>
    <color name="gray">#607D8B</color>
    <color name="soothing_blue">#0083BF</color>
    <color name="dark_blue">#005A8A</color>
</resources>
```

Цветовые ресурсы удобны тем, что вы можете в одном месте определить цветовые значения, которые затем будут использоваться в разных местах приложения.

Стили

А теперь займемся изменением внешнего вида кнопок BeatBox. *Стиль* (*style*) представляет собой набор атрибутов, которые могут применяться к виджетам.

Откройте файл `res/values/styles.xml` и добавьте стиль `BeatBoxButton`. (При создании приложения BeatBox в новый проект должен быть включен встроенный файл `styles.xml`. Если в вашем проекте его нет, создайте этот файл.)

Листинг 21.2. Добавление стиля (res/values/styles.xml)

```
<resources>

                    <style     name="AppTheme"
parent="Theme.AppCompat.Light.DarkActionBar">
```

```
        <!-- Настройте свою тему здесь. -->
                                                <item
name="colorPrimary">@color/colorPrimary</item>
                                                <item
name="colorPrimaryDark">@color/colorPrimaryDark
</item>
                                                <item
name="colorAccent">@color/colorAccent</item>
    </style>

    <style name="BeatBoxButton">
                                                <item
name="android:background">@color/dark_blue</ite
m>
    </style>

</resources>
```

Мы создали стиль BeatBoxButton. Этот стиль определяет всего один атрибут android:background, которому назначается темно-синий цвет. Вы можете применить этот стиль к любому количеству виджетов, а потом обновить атрибуты всех этих виджетов в одном месте.

Примените стиль BeatBoxButton к кнопкам приложения BeatBox.

Листинг 21.3. Использование стиля (res/layout/list_item_sound.xml)

```
<Button
        style="@style/BeatBoxButton"
        android:layout_width="match_parent"
```

```
        android:layout_height="120dp"
                    android:onClick="@{()   ->
viewModel.onButtonClicked()}"
        android:text="@{viewModel.title}"
        tools:text="Sound name"/>
```

Запустив BeatBox, вы увидите, что все кнопки окрасились в темно-синий цвет фона (рис. 21.2).

Стиль можно создать для любого набора атрибутов, которые должны многократно использоваться в приложении. Что и говорить, удобно.

Наследование стилей

Стили также поддерживают наследование. Стиль может наследовать и переопределять атрибуты из других стилей.

Создайте новый стиль `BeatBoxButton.Strong`, который наследует от `BeatBoxButton`, но дополнительно выделяет текст полужирным шрифтом.

Листинг 21.4. Наследование от BeatBoxButton (res/values/styles.xml)

```
<style name="BeatBoxButton">
                                        <item
name="android:background">@color/dark_blue</ite
m>
</style>

<style name="BeatBoxButton.Strong">
    <item name="android:textStyle">bold</item>
</style>
```

Рис. 21.2. Приложение BeatBox со стилями кнопок

(Хотя атрибут `android:textStyle` также можно было добавить в стиль `BeatBoxButton` напрямую, мы создали `BeatBoxButton.Strong` для демонстрации механизма наследования стилей.)

Схема формирования имен в данном случае выглядит немного странно. Когда вы присваиваете стилю имя `BeatBoxButton.Strong`, вы тем самым указываете, что он наследует атрибуты от `BeatBoxButton`.

Также существует альтернативный механизм формирования имен при наследовании — с явным указанием родителя при объявлении стиля:

```
<style name="BeatBoxButton">
                                                <item
name="android:background">@color/dark_blue</ite
m>
</style>

<style                name="StrongBeatBoxButton"
parent="@style/BeatBoxButton">
    <item name="android:textStyle">bold</item>
</style>
```

В приложении BeatBox будет использоваться схема `BeatBoxButton.Strong`.

Обновите файл `res/layout/list_item_sound.xml`, чтобы использовать новый стиль с жирным шрифтом.

Листинг 21.5. Применение жирного шрифта (res/layout/list_item_sound.xml)

```
<Button
    style="@style/BeatBoxButton.Strong"
    android:layout_width="match_parent"
    android:layout_height="120dp"
                  android:onClick="@{()     ->
viewModel.onButtonClicked()}"
    android:text="@{viewModel.title}"
    tools:text="Sound name"/>
```

Запустите приложение BeatBox и убедитесь в том, что текст на кнопках действительно выводится жирным шрифтом (рис. 21.3).

Рис. 21.3. Приложение BeatBox с жирным шрифтом

Темы

Стили — классная штука. Они позволяют определить набор атрибутов в одном месте, а затем применить их к любому количеству виджетов на ваше усмотрение. Впрочем, у них есть и недостаток: вам придется последовательно применять их к каждому виджету. А если вы пишете сложное приложение с множеством кнопок в многочисленных макетах? Добавление стиля `BeatBoxButton` ко всем кнопкам станет весьма масштабной задачей.

На помощь приходят *темы* (*themes*). Темы идут еще дальше стилей: они, как и стили, позволяют определить набор атрибутов в одном месте, но затем эти атрибуты автоматически применяются во всем приложении. В атрибутах темы могут содержаться ссылки на конкретные ресурсы (например, цвета), а также ссылки на стили. Например, в определении темы можно сказать: «Я хочу, чтобы все кнопки использовали этот стиль». И вам не придется отыскивать каждый виджет кнопки и приказывать ему использовать указанную тему.

Изменение темы

При создании приложения BeatBox ему была назначена тема по умолчанию. Откройте файл `manifests/AndroidManifest.xml` и найдите атрибут `theme` в теге `application`.

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
     package="com.bignerdranch.android.beatbox"
>

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
            android:theme="@style/AppTheme">
        ...
    </application>
```

```
</manifest>
```

Атрибут `theme` ссылается на тему `AppTheme`. Тема `AppTheme` была объявлена в файле `styles.xml`, который мы изменили ранее.

Как видите, тема также является стилем. Однако темы определяют другие атрибуты (вскоре вы убедитесь в этом). Кроме того, определение тем в манифесте наделяет их суперспособностями: именно этот факт позволяет автоматически применить тему в границах целого приложения.

Перейдите к определению темы `AppTheme`, щелкнув с нажатой клавишей ⌘ (**Ctrl**) на `@style/AppTheme`. Android Studio откроет файл `res/values/styles.xml`.

```
<resources>

                    <style    name="AppTheme"
parent="Theme.AppCompat.Light.DarkActionBar">
        ...
       </style>

    <style name="BeatBoxButton">
                                          <item
name="android:background">@color/dark_blue</ite
m>
    </style>
    ...
</resources>
```

При выборе опции **UseartifactsAndroidX** в новых проектах, создаваемых в Android Studio, им присваивается тема

AppCompat. AppTheme наследует атрибуты от Theme.AppCompat.Light.DarkActionBar. Внутри AppTheme можно добавлять или переопределять дополнительные значения из родительской темы.

Библиотека AppCompat включает три основные темы:

- Theme.AppCompat — темная тема;
- Theme.AppCompat.Light — светлая тема;
- Theme.AppCompat.Light.DarkActionBar — светлая тема с темной панелью приложения.

Замените родительскую тему на Theme.AppCompat, чтобы в BeatBox использовалась базовая темная тема.

Листинг 21.6. Переключение на темную тему (res/values/styles.xml)

```
<resources>

    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        ...
    </style>
    ...
</resources>
```

Запустите BeatBox и посмотрите, как выглядит темная тема (рис. 21.4).

Рис. 21.4. BeatBox с темной темой

Добавление цветов в тему

Разобравшись с выбором базовой темы, мы переходим к настройке атрибутов темы `AppTheme` приложения BeatBox.

В файле `styles.xml` содержатся три атрибута вашей темы. Приведите их в соответствие с листингом 21.7.

Листинг 21.7. Изменение атрибутов темы (res/values/styles.xml)

```
<style                                    name="AppTheme"
parent="Theme.AppCompat">
    <!-- Настройте свою тему здесь. -->
                                                  <item
name="colorPrimary">@color/    colorPrimaryred<
/item>
                                                  <item
name="colorPrimaryDark">@color/    colorPrimary
Darkdark_red</item>
                                                  <item
name="colorAccent">@color/    colorAccentgray</
item>
</style>
```

Эти атрибуты похожи на атрибуты стилей, которыми мы занимались ранее, но они задают другие свойства. Атрибуты стиля задают свойства индивидуального виджета, как, например, `textStyle` при выделении текста кнопки жирным шрифтом. Атрибуты темы имеют более широкую область действия: это свойства, которые задаются на уровне темы и становятся доступными для любых виджетов. Например, панель приложения обращается к атрибуту темы `colorPrimary` для назначения своего цвета фона.

Назначение этих трех атрибутов приводит к масштабным последствиям. Атрибут `colorPrimary` определяет первичный цвет фирменного стиля вашего приложения. Этот цвет будет использоваться для фона панели приложения, а также в нескольких других местах.

Атрибут `colorPrimaryDark` используется для окраски строки состояния, отображаемой у верхнего края экрана. Обычно цвет `colorPrimaryDark` представляет собой чуть

более темную версию цвета `colorPrimary`. Тематическое оформление строки состояния относится к числу возможностей, добавленных в Android в Lollipop. Помните, что строка состояния будет окрашена в черный цвет на старых устройствах (независимо от настроек темы). На рис. 21.5 показан эффект от назначения этих двух атрибутов темы в приложении BeatBox.

Рис. 21.5. BeatBox с пользовательскими цветовыми атрибутами AppCompat

Наконец, `colorAccent` назначается серый цвет. Цвет `colorAccent` должен контрастировать с атрибутом `colorPrimary`; он используется для формирования оттенка в некоторых виджетах (например, `EditText`).

Атрибут `colorAccent` на внешнем виде приложения BeatBox не отражается, потому что кнопки не поддерживают оттенки. Тем не менее мы все равно задаем `colorAccent`, потому что эти три цветовых атрибута удобнее рассматривать вместе. Запустите приложение BeatBox и понаблюдайте за новыми цветами в действии.

Ваше приложение должно выглядеть так, как показано на рис. 21.5.

Переопределение атрибутов темы

Теперь пора поглубже изучить вопрос и поближе познакомиться с атрибутами тем, которые вы можете переопределять. Учтите, что исследование тем — непростое дело. Почти не существует документации, в которой было бы указано, какие существуют атрибуты, какие из них вы можете переопределять и что делает тот или иной атрибут. Вам придется самостоятельно прокладывать путь в этих джунглях. Хорошо, что у вас есть надежный проводник (эта книга).

Начнем с изменения цвета фона BeatBox посредством модификации темы. Хотя теоретически вы можете открыть файл `res/layout/fragment_beat_box.xml` и вручную задать атрибут `android:background` для виджета `RecyclerView`, а затем повторить процесс со всеми остальными файлами макетов фрагментов и activity. Конечно, это будет весьма неэффективно — и не только по затратам времени, но и по затратам труда программиста.

Тема всегда назначает цвет фона. Назначая другой цвет поверх имеющегося, вы выполняете лишнюю работу. Кроме того, дублирование атрибута в приложении усложняет сопровождение кода.

Исследование тем

Вместо этого было бы правильнее переопределить атрибут цвета фона в теме. Чтобы узнать имя атрибута, взгляните, как атрибут задается в родительской теме: `Theme.AppCompat`.

Возникает резонный вопрос: как узнать, какой атрибут нужно переопределить, если неизвестно его имя? Никак. Вы будете читать имена атрибутов, и в какой-то момент у вас блеснет догадка: «А вот это похоже». Вы переопределите атрибут, запустите приложение и будете надеяться, что сделали разумный выбор.

В конечном итоге требуется найти самого первого предка вашей темы: ее прапрапра... в общем, предка неизвестно какого уровня. Для этого вы будете переходить к родителю более высокого уровня, пока не найдете подходящий атрибут.

Откройте файл `styles.xml` и щелкните мышью по `Theme.AppCompat` с нажатой клавишей ⌘ (**Ctrl**). Посмотрим, как далеко уходит этот лабиринт.

(Если вам не удается переходить по атрибутам тем прямо в Android Studio или вы предпочитаете делать это вне Android Studio, исходные файлы тем Android находятся в каталоге `ваш-каталог-SDK/platforms/android-28/data/res/values`.)

На момент написания книги при этом открывался очень большой файл, в котором выделялась следующая строка:

```
<style name="Theme.AppCompat"
parent="Base.Theme.AppCompat" />
```

Тема `Theme.AppCompat` наследует атрибуты от `Base.Theme.AppCompat`. Интересно, что `Theme.AppCompat` не переопределяет никакие атрибуты, а только содержит ссылку на своего родителя.

Щелкните мышью по `Base.Theme.AppCompat` с нажатой клавишей ⌘ (**Ctrl**). Android Studio сообщит, что тема уточняется по ресурсам. Существует несколько разных версий этой темы в зависимости от используемой версии Android.

Выберите версию values/values.xml; откроется определение Base.Theme.AppCompat (рис. 21.6).

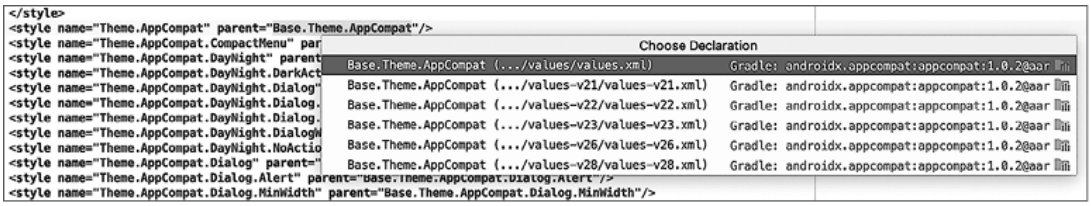


Рис. 21.6. Список версий родительской темы для выбора

Выбор неклассифицированной версии может показаться странным, так как минимальная поддерживаемая версия API вашего приложения — 21. Мы сделали именно так, потому что атрибут фоновой темы существует гораздо дольше, чем API 21, и поэтому он должен существовать в оригинальной версии Base.Theme.AppCompat.

```
<style name="Base.Theme.AppCompat"
parent="Base.V7.Theme.AppCompat">
</style>
```

Base.Theme.AppCompat — еще одна тема, которая существует только ради имени и не переопределяет никакие атрибуты. Вернитесь к родительской теме: Base.V7.Theme.AppCompat.

```
<style name="Base.V7.Theme.AppCompat"
parent="Platform.AppCompat">
    <item name="viewInflaterClass">
            androidx.appcompat.app.AppCompatVie
wInflater</item>
    <item name="windowNoTitle">false</item>
    <item name="windowActionBar">true</item>
                                        <item
name="windowActionBarOverlay">false</item>
```

```
    ...
</style>
```

Постепенно круг замыкается. Просмотрите список атрибутов `Base.V7.Theme.AppCompat`.

На первый взгляд нет атрибута, который соответствовал бы цели: изменению цвета фона. Перейдите к `Platform.AppCompat`. Вы увидите, что и эта тема уточняется по ресурсам. Выберите версию `values/values.xml`.

```
<style                  name="Platform.AppCompat"
parent="android:Theme.Holo">
                                                <item
name="android:windowNoTitle">true</item>
                                                <item
name="android:windowActionBar">false</item>

          <item  name="android:buttonBarStyle">?
attr/buttonBarStyle</item>
    <item name="android:buttonBarButtonStyle">?
attr/buttonBarButtonStyle</item>
                                                <item
name="android:borderlessButtonStyle">?
attr/borderlessButtonStyle</item>
    ...
</style>
```

Наконец-то мы видим, что родителем темы `Platform.AppCompat` является тема `android:Theme.Holo`. Обратите внимание: ссылка на родительскую тему не записывается в виде `Theme`, а имеет пространство имен `android`. К ней также добавляется префикс пространства имен android.

Считайте, что библиотека AppCompat находится внутри вашего приложения. При сборке проекта вы включаете библиотеку AppCompat, которую сопровождает набор файлов с кодом Kotlin (и Java) и разметкой XML. Эти файлы практически ничем не отличаются от файлов, которые вы пишете самостоятельно. Если вы захотите сослаться на какой-либо компонент из библиотеки AppCompat, вы делаете это напрямую, используя запись `Theme.AppCompat`, потому что эти файлы существуют в вашем приложении.

Темы, существующие в ОС Android, такие как `Theme`, должны объявляться с пространством имен, указывающим на их местоположение. Библиотека AppCompat использует запись `android:Theme`, потому что тема существует в ОС Android.

Наконец мы пришли к месту назначения. Здесь представлено гораздо больше атрибутов, чем вам захочется переопределять в своей теме. Конечно, вы можете перейти к родителю `Platform.AppCompat` — `Theme.Holo`, но это не обязательно. Тема определяет все атрибуты, которые вам понадобятся.

Прямо над текстовыми атрибутами есть объявление `windowBackground`. По имени атрибута можно предположить, что он определяет цвет фона темы.

```
<style name="Platform.AppCompat"
parent="android:Theme.Holo">
    ...

    <!-- Window colors -->
    ...
                                    <item
name="android:windowBackground">@color/backgrou
nd_material_dark</item>
```

Этот атрибут должен переопределяться в приложении BeatBox. Вернитесь к файлу styles.xml и переопределите атрибут windowBackground.

Листинг 21.8. Настройка фона окна (res/values/styles.xml)

```
<style name="AppTheme" parent="Theme.AppCompat">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/red</item>
    <item name="colorPrimaryDark">@color/dark_red</item>
    <item name="colorAccent">@color/gray</item>

    <item name="android:windowBackground">@color/soothing_blue</item>
</style>
```

Обратите внимание на необходимость использования пространства имен android при переопределении атрибута, потому что windowBackground объявляется в ОС Android.

Запустите BeatBox, прокрутите представление до конца и убедитесь в том, что в том месте, где фон не закрыт кнопками, он окрашен в синий цвет (рис. 21.7).

Рис. 21.7. BeatBox с окраской фона с помощью темы

Действия, которые мы только что совершили для нахождения атрибута `windowBackground`, приходится выполнять каждому разработчику Android при модификации темы приложения. Документация по этим атрибутам практически отсутствует. Чтобы узнать о доступных возможностях, обычно приходится обращаться к источникам.

Итак, мы переходили по следующим темам:

- `Theme.AppCompat`
- `Base.Theme.AppCompat`

- `Base.V7.Theme.AppCompat`
- `Platform.AppCompat`

Мы переходили по иерархии тем до тех пор, пока не добрались до корневой темы AppCompat. Когда вы лучше освоитесь с возможностями работы с темами, вероятно, вы сразу будете переходить к соответствующей теме. Тем не менее полезно пройти по иерархии до ее корня, чтобы понять логику происхождения темы.

Иерархия тем может измениться со временем, но задача перемещения по иерархии останется неизменной. Вы переходите по иерархии тем до тех пор, пока не будет найден атрибут, который требуется переопределить.

Изменение атрибутов кнопки

Ранее мы настраивали кнопки приложения BeatBox вручную, задавая атрибут `style` в файле `res/layout/list_item_sound.xml`. В более сложных приложениях, в которых кнопки распределены по многим фрагментам, решение с назначением атрибута `style` для каждой кнопки плохо масштабируется. В такой ситуации можно пойти дальше и определить в теме стиль для всех кнопок приложения.

Прежде чем добавлять в тему стиль кнопок, удалите атрибут `style` из файла `res/layout/list_item_sound.xml`.

Листинг 21.9. Долой! Есть способ получше (res/layout/list_item_sound.xml)

```
<Button
```

```
        style="@style/BeatBoxButton.Strong"
    android:layout_width="match_parent"
    android:layout_height="120dp"
                    android:onClick="@{()     ->
viewModel.onButtonClicked()}"
    android:text="@{viewModel.title}"
    tools:text="Sound name"/>
```

Запустите приложение BeatBox и убедитесь в том, что кнопки вернулись к старому невыразительному виду (рис. 21.8).

Пора снова заняться исследованиями темы. На этот раз нас интересует атрибут `buttonStyle`. Вы найдете его в теме `Base.V7.Theme.AppCompat`.

Рис. 21.8. Приложение BeatBox со старомодными и невыразительными кнопками

```
<style            name="Base.V7.Theme.AppCompat"
parent="Platform.AppCompat">
    ...
    <!-- Button styles -->
                                          <item
name="buttonStyle">@style/Widget.AppCompat.Butt
on</item>
                                              <item
name="buttonStyleSmall">@style/Widget.AppCompat
.Button.Small</item>
    ...
</style>
```

Атрибут `buttonStyle` определяет стиль любой нормальной кнопки в приложении.

Атрибуту `buttonStyle` вместо значения назначается ресурс стиля. При обновлении атрибута `windowBackground` передается значение: цвет. В нашем случае атрибут `buttonStyle` должен ссылаться на стиль. Перейдите к `Widget.AppCompat.Button`, чтобы просмотреть стиль кнопки. (Если не получается щелкнуть по стилю с нажатой клавишей (**Ctrl**), найдите нужное значение в файле `values.xml`, и там вы увидите определение стиля.)

```
<style            name="Widget.AppCompat.Button"
parent="Base.Widget.AppCompat.Button"/>
```

Стиль `Widget.AppCompat.Button` самостоятельно никакие атрибуты не определяет. Чтобы просмотреть его содержимое, перейдите к его родителю. Вы увидите, что базовый стиль существует в двух версиях. Выберите версию `values/values.xml`.

```
<style name="Base.Widget.AppCompat.Button" parent="android:Widget">
                                                <item name="android:background">@drawable/abc_btn_default_mtrl_shape</item>
        <item name="android:textAppearance">?android:attr/textAppearanceButton</item>
    <item name="android:minHeight">48dip</item>
    <item name="android:minWidth">88dip</item>
    <item name="android:focusable">true</item>
    <item name="android:clickable">true</item>
                                                <item name="android:gravity">center_vertical|center_horizontal</item>
</style>
```

Каждой кнопке Button, используемой в BeatBox, назначаются эти атрибуты.

Воспроизведите в BeatBox то, что происходит в теме Android. Измените родителя BeatBoxButton, чтобы атрибуты наследовались от существующего стиля кнопки (листинг 21.10). Также удалите стиль BeatBoxButton.Strong, который использовался ранее.

Листинг 21.10. Создание стиля кнопки (res/values/styles.xml)

```
<resources>

                        <style name="AppTheme" parent="Theme.AppCompat">
                                                <item name="colorPrimary">@color/red</item>
```

```
                                                    <item
name="colorPrimaryDark">@color/dark_red</item>
                                                    <item
name="colorAccent">@color/gray</item>
                                                    <item
name="android:windowBackground">@color/soothing
_blue</item>
    </style>

                    <style    name="BeatBoxButton"
parent="Widget.AppCompat.Button">
                                                    <item
name="android:background">@color/dark_blue</ite
m>
    </style>

        <style name="BeatBoxButton.Strong">
                                                    <item
name="android:textStyle">bold</item>
        </style>

</resources>
```

Мы указали родителя `Widget.AppCompat.Button`. Наша кнопка должна наследовать все свойства обычной кнопки, а затем избирательно изменять некоторые атрибуты.

Если родительская тема для `BeatBoxButton` не указана, то внешний вид кнопок резко ухудшается, и кнопка вообще перестает напоминать кнопку. Многие привычные свойства, например выравнивание текста по центру кнопки, пропадают.

Итак, стиль `BeatBoxButton` полностью определен, и мы можем использовать его в приложении. Вернитесь к атрибуту

buttonStyle, который мы обнаружили ранее в ходе исследования тем Android. Продублируйте этот атрибут в своей теме.

Листинг 21.11. Использование стиля BeatBoxButton (res/values/styles.xml)

```
<resources>

                    <style     name="AppTheme"
parent="Theme.AppCompat">
        <!-- Customize your theme here. -->
                                                <item
name="colorPrimary">@color/red</item>
                                                <item
name="colorPrimaryDark">@color/dark_red</item>
                                                <item
name="colorAccent">@color/gray</item>

                                                <item
name="android:windowBackground">@color/soothing
_blue</item>
                                                <item
name="buttonStyle">@style/BeatBoxButton</item>
    </style>

                  <style    name="BeatBoxButton"
parent="Widget.AppCompat.Button">
                                                <item
name="android:background">@color/dark_blue</ite
m>
    </style>
```

```
</resources>
```

Обратите внимание: при определении `buttonStyle` префикс android: не используется. Дело в том, что переопределяемый атрибут `buttonStyle` реализован в библиотеке AppCompat.

Мы переопределяем атрибут `buttonStyle` и подставляем свой собственный стиль: `BeatBoxButton`.

Запустите приложение BeatBox и обратите внимание на то, что все кнопки окрасились в темно-синий цвет (рис. 21.9). Таким образом, мы изменили внешний вид всех обычных кнопок в BeatBox без прямой модификации файлов макетов. Атрибуты темы в Android — сила!

Рис. 21.9. Приложение BeatBox с полностью определенными темами

Конечно, было бы хорошо, если бы кнопки были больше похожи на кнопки. В следующей главе проблема будет решена, и тогда кнопки проявят себя в полной мере.

Для любознательных: подробнее о наследовании стилей

Описание наследований стилей, приведенное ранее, не содержит полной информации. Возможно, вы заметили изменение в стиле наследования во время изучения иерархии тем. В темах AppCompat имя темы используется для обозначения наследования — до того момента, когда мы приходим к теме `Platform.AppCompat`.

```
<style                    name="Platform.AppCompat"
parent="android:Theme.Holo">
    ...
</style>
```

Здесь вместо «наследного» стиля формирования имен происходит переключение на прямое определение родителя в атрибуте `parent`. Почему?

Указание родительской темы в имени темы работает только для тем, существующих в одном пакете. Таким образом, в темах ОС Android в большинстве случаев используется «наследный» стиль имен, и библиотека AppCompat действует так же. Но как только наследование пересекает границу AppCompat, используется атрибут `parent`.

Это правило желательно соблюдать и в ваших приложениях. Укажите родителя темы в имени темы, если вы наследуете от одной из своих собственных тем. Если же наследование осуществляется от стиля или темы ОС Android, используйте явное задание атрибута `parent`.

Для любознательных: доступ к атрибутам тем

После того как атрибуты будут объявлены в теме, вы сможете обращаться к ним из XML или из кода.

Для обращения к атрибуту темы из разметки XML используется запись, продемонстрированная для атрибута `listSeparatorTextViewStyle` в главе 8. Для ссылки на конкретное значение в XML (например, цвет) используется синтаксис `@`. Запись `@color/gray` указывает на конкретный ресурс.

Для ссылок на ресурс в теме используется знак `?`:

```
<Button
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/list_item_sound_button"
    android:layout_width="match_parent"
    android:layout_height="120dp"
       android:background="?attr/colorAccent"
    tools:text="Sound name"/>
```

Знак ? указывает на использование ресурса, на который указывает атрибут `colorAccent` вашей темы. В данном случае это серый цвет, определенный в файле `colors.xml`.

Также можно использовать атрибуты темы в коде, хотя на этот раз запись получается не столь компактной.

```
val theme: Resources.Theme = activity.theme
val attrsToFetch = intArrayOf(R.attr.colorAccent)
val a: TypedArray = theme.obtainStyledAttributes(R.style.AppTheme, attrsToFetch)
val accentColor = a.getInt(0, 0)
a.recycle()
```

В объекте `Theme` мы приказываем разрешить атрибут `R.attr.colorAccent`, определенный в `AppTheme`: `R.style.AppTheme`. Вызов возвращает массив `TypedArray`, содержащий данные. Из массива `TypedArray` извлекается значение типа `int`, которое в дальнейшем может использоваться, например, для изменения фона кнопки.

Панель приложения и кнопки в BeatBox именно так поступают для определения своего стиля на основании атрибутов темы.

22. Графические объекты

После назначения тем BeatBox пришло время поработать над кнопками.

В текущей версии кнопки никак не реагируют на нажатия — это просто синие прямоугольники. В этой главе мы при помощи *графических объектов* (*drawable*) *XML* поднимем приложение BeatBox на новый уровень (рис. 22.1).

Рис. 22.1. Новая версия BeatBox

В Android графическим объектом называется все, что предназначено для прорисовки на экране, будь то абстрактная

фигура, класс, производный от `Drawable`, или растровое изображение. В этой главе мы рассмотрим несколько видов графических объектов: списки состояний, геометрические фигуры и списки слоев. Все три вида определяются в файлах XML, поэтому мы объединим их в более широкую категорию графических объектов XML.

Унификация кнопок

Прежде чем браться за создание графических объектов XML, внесите изменения в файл `res/layout/list_item_sound.xml`, чтобы подготовить ваши кнопки к будущим изменениям.

Листинг 22.1. Определение размеров (res/layout/list_item_sound.xml)

```
<layout
xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable
                name="viewModel"
                type="com.bignerdranch.android.beatbox.SoundViewModel"/>
    </data>
    <FrameLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
```

```
            android:layout_margin="8dp">
        <Button
                    android:layout_width="match
_parent"
                    android:layout_height="120d
p"
                android:layout_width="100dp"
                android:layout_height="100dp"
                android:layout_gravity="center"
                      android:onClick="@{() ->
viewModel.onButtonClicked()}"
                android:text="@{viewModel.title
}"
                tools:text="Sound name"/>
    </FrameLayout>
</layout>
```

Каждой кнопке назначается ширина и высота `100dp`, чтобы последующее преобразование кнопок в круги не приводило к искажениям.

В `RecyclerView` независимо от размера экрана всегда отображаются три столбца. При наличии свободного места `RecyclerView` растянет эти столбцы по размерам экрана. В нашем приложении кнопки растягиваться не должны, поэтому они были заключены в виджет `FrameLayout`: он будет растягиваться, а кнопки — нет.

Запустите приложение BeatBox. Вы увидите, что все кнопки имеют одинаковые размеры и разделяются небольшими интервалами (рис. 22.2).

Рис. 22.2. Равномерное распределение кнопок

Геометрические фигуры

Придадим кнопкам круглую форму с помощью *shape drawable*.

Поскольку графические объекты XML не привязаны к конкретной плотности пикселов, обычно они размещаются в папке `drawable` по умолчанию (а не в одной из специализированных папок). На панели **Project** создайте в папке `res/drawable` файл `button_beat_box_normal.xml`. (Почему кнопка названа «нормальной»? Потому что скоро появится другая, менее нормальная.)

Листинг 22.2. Создание графического объекта (res/drawable/button_beat_box_normal.xml)

```
<shape
xmlns:android="http://schemas.android.com/apk/res/android"
        android:shape="oval">

    <solid android:color="@color/dark_blue"/>

</shape>
```

Этот файл создает эллиптический графический объект, заполненный темно-синим цветом. Вы можете использовать элемент shape для создания других фигур: прямоугольников, линий, градиентов и т.д. За подробностями обращайтесь к документации по адресу **developer.android.com/guide/topics/resources/drawable-resource.html**.

Назначьте button_beat_box_normal в качестве фона кнопок.

Листинг 22.3. Изменение фона кнопок (res/values/styles.xml)

```
<resources>

                    <style      name="AppTheme"
parent="Theme.AppCompat">
        ...
    </style>

                <style    name="BeatBoxButton"
parent="Widget.AppCompat.Button">
```

```
                                        <item
name="android:background">@color/dark_blue</ite
m>
                                        <item
name="android:background">@drawable/button_beat
_box_normal</item>
    </style>
</resources>
```

Запустите приложение BeatBox. Все кнопки стали круглыми (рис. 22.3).

Рис. 22.3. Круглые кнопки

Нажмите кнопку. Вы услышите звук, но внешний вид кнопки не изменится. Было бы лучше, если бы нажатая кнопка отличалась внешним видом от ненажатой.

Списки состояний

Чтобы решить эту проблему, сначала определите новую геометрическую фигуру для кнопки в нажатом состоянии.

Создайте в папке `res/drawable` файл `button_beat_box_pressed.xml`. Нажатая кнопка будет выглядеть так же, как обычная, но ей будет назначен красный цвет фона.

Листинг 22.4. Определение графического объекта для нажатой кнопки (res/drawable/button_beat _box_pressed.xml)

```
<shape
xmlns:android="http://schemas.android.com/apk/res/android"
        android:shape="oval">

    <solid android:color="@color/red"/>

</shape>
```

Новая версия должна использоваться тогда, когда пользователь нажимает кнопку. Для решения этой задачи используется *список состояний*.

Список состояний представляет собой графический объект, который указывает на другие графические объекты в зависимости от состояния некоторого селектора. Кнопка может находиться в нажатом и ненажатом состоянии. Список

состояний будет определять один графический объект — как фон для кнопки в нажатом состоянии и другой графический объект — для ненажатых кнопок.

Определите список состояний в папке `res/drawable`.

Листинг 22.5. Создание списка состояний (res/drawable/button_beat_box.xml)

```
<selector
xmlns:android="http://schemas.android.com/apk/res/android">
                                                    <item
android:drawable="@drawable/button_beat_box_pressed"
          android:state_pressed="true"/>
                                                    <item
android:drawable="@drawable/button_beat_box_normal" />
</selector>
```

Измените стиль кнопок, чтобы список состояний использовался в качестве фона.

Листинг 22.6. Назначение списка состояний (res/values/styles.xml)

```
<resources>

                        <style      name="AppTheme"
parent="Theme.AppCompat">
        ...
    </style>
```

```
                <style name="BeatBoxButton"
parent="Widget.AppCompat.Button">
                                        <item
name="android:background">@drawable/button_beat
_box_normal</item>
                                        <item
name="android:background">@drawable/button_beat
_box</item>
    </style>

</resources>
```

Когда кнопка находится в нажатом состоянии, в качестве фона используется графический объект `button_beat_box_pressed`. В противном случае фон кнопки определяется `button_beat_box_normal`.

Запустите приложение BeatBox и нажмите кнопку. Фон кнопки изменится (рис. 22.4). Впечатляет, не правда ли?

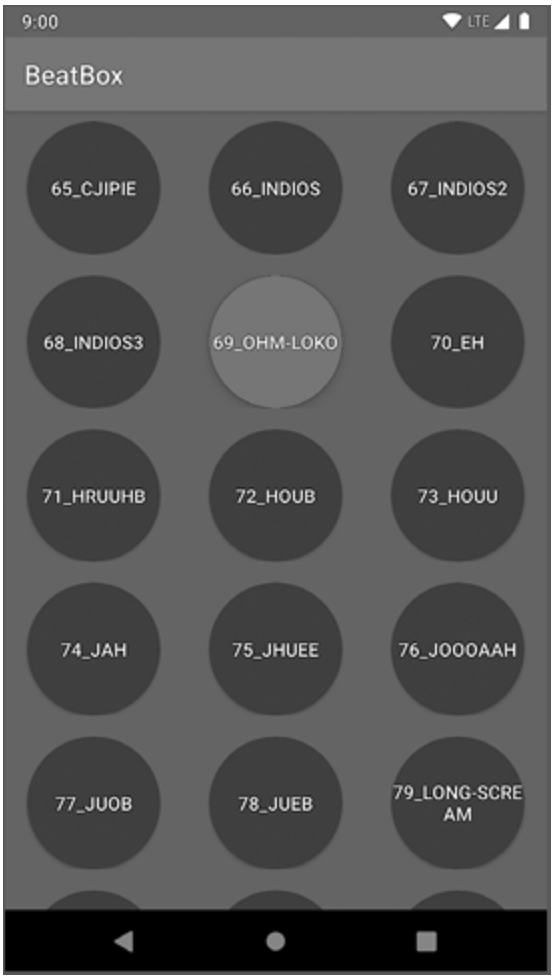


Рис. 22.4. BeatBox с кнопкой в нажатом состоянии

Списки состояний — удобный инструмент настройки элементов интерфейса. Также поддерживаются и другие состояния, включая состояние блокировки, наличия фокуса и активации. За подробностями обращайтесь к документации по адресу **developer.android.com/reference/android/graphics/drawable/StateListDrawable**.

Списки слоев

Приложение BeatBox хорошо смотрится: в нем используются круглые кнопки, которые визуально реагируют на нажатия. Но

пришло время сделать что-то более интересное.

Списки слоев (*layer list*) позволяют объединить два графических объекта XML в один объект. Вооружившись этим инструментом, мы добавим темное кольцо вокруг кнопки в нажатом состоянии.

Листинг 22.7. Использование списка слоев (res/drawable/button_beat_box_pressed.xml)

```
<layer-list
xmlns:android="http://schemas.android.com/apk/res/android">
    <item>
                                                <shape
   xmlns:android="http://schemas.android.com/apk/res/android"
            android:shape="oval">

          <solid android:color="@color/red"/>
        </shape>
    </item>
    <item>
        <shape android:shape="oval">

            <stroke android:width="4dp"
                    android:color="@color/dark_red"/>
        </shape>
    </item>
</layer-list>
```

Список слоев состоит из двух графических объектов. Первый объект — красный круг, как и перед изменением. Второй графический объект будет нарисован поверх первого. Второй графический объект представляет собой другой овал с толщиной линии `4dp`; в результате создается кольцо темно-красного цвета.

Объединение этих двух графических объектов образует список слоев. В список слоев можно включить более двух графических объектов, чтобы добиться еще более сложного результата.

Запустите приложение BeatBox и нажмите кнопку. Вы увидите, что в нажатом состоянии она выделяется кольцом (рис. 22.5). Еще лучше, чем прежде.

Добавление списка слоев завершает приложение BeatBox. Помните, как банально выглядел исходный интерфейс? Чтобы создать нечто куда более интересное, нам потребовалось совсем немного времени. Сделайте ваше приложение приятным для глаз — и ваши усилия окупятся его популярностью.

Рис. 22.5. Готовое приложение BeatBox

Для любознательных: для чего нужны графические объекты XML

Нам всегда нужно нажатое состояние для кнопок, поэтому список состояний является критическим компонентом любого Android-приложения. Но как насчет геометрических фигур и списков слоев? Стоит ли использовать их?

Графические объекты XML отличаются гибкостью. Их можно использовать для многих целей и легко обновлять в будущем. Комбинации списков слоев и геометрических фигур позволяют создавать сложные фоны без графического редактора. Если вы

решите изменить цветовую схему в BeatBox, обновить цвета в графическом объекте XML будет несложно.

В этой главе графические объекты XML определялись в каталоге `drawable` без квалификаторов, уточняющих плотность пикселов. Это объясняется тем, что графические объекты XML не зависят от плотности. Стандартные растровые изображения обычно приходится создавать для нескольких вариантов плотности, чтобы изображение выглядело четко на большинстве устройств. Графические объекты XML достаточно определить только один раз, и они будут четко выглядеть при любой плотности.

Для любознательных: Mipmap

Квалификаторы ресурсов и графические объекты удобны. Когда вам потребуется добавить графику в приложение, вы генерируете изображение нескольких разных размеров и раскладываете их по папкам с квалификаторами: `drawable-mdpi`, `drawable-hdpi` и т.д. Затем вы обращаетесь к изображению по имени, а Android выбирает версию с нужной плотностью в зависимости от текущего устройства.

Однако у этой системы есть свой недостаток. Файл APK, публикуемый в Google Play Store, содержит все изображения из каталогов `drawable` для всех вариантов плотности, добавленные в проект, при том что многие из них использоваться не будут. Конечно, это приводит к неэффективному увеличению размера приложения.

Для решения этой проблемы можно сгенерировать разные APK для всех вариантов плотности пикселов: APK с mdpi-версией приложения, APK с hdpi-версией приложения и т.д. (За дополнительной информацией о сборке разных версий APK

обращайтесь к документации: **developer.android.com/studio/build/configure-apk-splits.html**.)

Впрочем, у этого правила есть одно исключение: вы должны поддерживать значки приложения для лаунчера во всех вариантах плотности.

В Android лаунчер представляет собой домашний экран со значками приложений (эта тема подробнее рассматривается в главе 23); он открывается при нажатии кнопки «Главный экран».

Некоторые новые лаунчеры отображают значки приложений в большем размере, чем принято. Чтобы увеличенный значок нормально смотрелся, такие лаунчеры должны взять значок из версии со следующей плотностью. Например, на hdpi-устройствах для представления приложения в лаунчере будет использоваться значок `xhdpi`. Но если xhdpi-версия исключена из APK, лаунчеру придется вернуться к версии с более низким разрешением.

Масштабированные значки с низким разрешением кажутся размытыми, а мы хотим, чтобы значок приложения выглядел четко и аккуратно.

В Android для решения этой проблемы используется каталог *mipmap*. При включении разбивки APK ресурсы mipmap не удаляются из APK. В остальном такие ресурсы не отличаются от графических объектов.

На момент написания книги новые проекты в Android Studio настроены на использование mipmap-ресурсов для значка лаунчера (рис. 22.6).

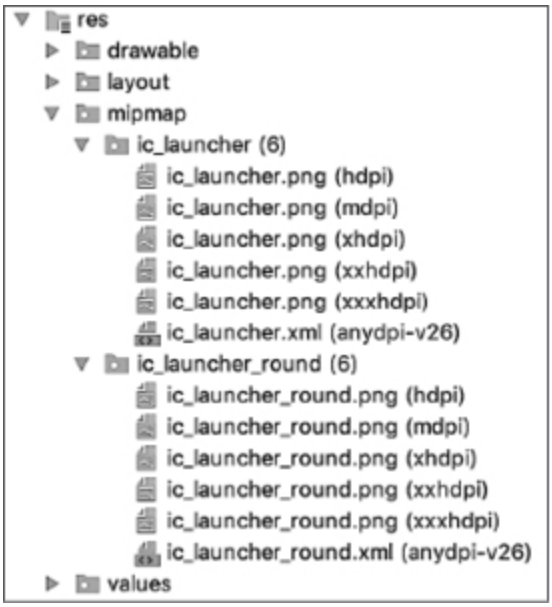


Рис. 22.6. Значки mipmap

Итак, мы рекомендуем просто разместить значок лаунчера в разных каталогах `mipmap`. Всем остальным изображениям — место в каталогах `drawable`.

Для любознательных: девятизонные изображения

Иногда (а может быть, очень часто) вы будете использовать обычные графические изображения в качестве фона кнопок. Но что произойдет с этими изображениями, если кнопка может отображаться в разных вариантах размеров? Если ширина кнопки превышает ширину фонового изображения, то изображение просто растянется, верно? Но всегда ли результат будет хорошо выглядеть?

Равномерное растяжение фонового изображения не всегда приводит к хорошему результату. Иногда требуется лучше управлять тем, как будет растягиваться изображение.

В этом разделе приложение BeatBox будет переведено на использование *9-зонного* изображения в качестве фона кнопок (вскоре эта тема будет рассмотрена более подробно). Выбор

решения объясняется даже не тем, что оно хорошо подходит для BeatBox — просто оно демонстрирует, как работают 9-зонные изображения в ситуациях с использованием графического файла.

Для начала изменим файл `list_item_sound.xml`, для того чтобы размеры кнопки могли изменяться по размерам свободного пространства.

Листинг 22.8. Включение растяжения кнопок (res/layout/list_item_sound.xml)

```
<layout
xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable
                name="viewModel"
                type="com.bignerdranch.android.beatbox.SoundViewModel"/>
    </data>
    <FrameLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_margin="8dp">
        <Button
                android:layout_width="100dp"
                android:layout_height="100dp"
```

```
                android:layout_width="match_par
ent"
                android:layout_height="match_pa
rent"
                android:layout_gravity="center"
                        android:onClick="@{() ->
viewModel.onButtonClicked()}"
                android:text="@{viewModel.title
}"
                tools:text="Sound name"/>
    </FrameLayout>
</layout>
```

Теперь кнопки занимают все доступное место, оставляя поля шириной 8dp. На рис. 22.7 показано, какое изображение будет использоваться в качестве нового фона кнопок BeatBox.

Рис. 22.7. Новое фоновое изображение кнопки (res/drawable-xxhdpi/ic_button_beat_box_default.png)

В решениях этой главы (см. ссылку **www.bignerdranch.com/solutions/AndroidProgramming4e.zip**) вы найдете это изображение вместе с версией для нажатого состояния в папке xxhdpi. Скопируйте оба изображения в папку drawable-xxhdpi вашего проекта и внесите изменения в файл button_beat_box.xml, чтобы назначить их фонами кнопок.

Листинг 22.9. Назначение новых фоновых изображений кнопок (res/drawable/button_beat_box.xml)

```
<selector
xmlns:android="http://schemas.android.com/apk/res/android">
                                                    <item
android:drawable="@drawable/button_beat_box_pressed"
                                                    <item
android:drawable="@drawable/ic_button_beat_box_pressed"
        android:state_pressed="true"/>
                                                    <item
android:drawable="@drawable/button_beat_box_normal"
                                                    <item
android:drawable="@drawable/ic_button_beat_box_default"
</selector>
```

Запустите BeatBox; вы увидите, что кнопки выводятся с новым фоном (рис. 22.8).

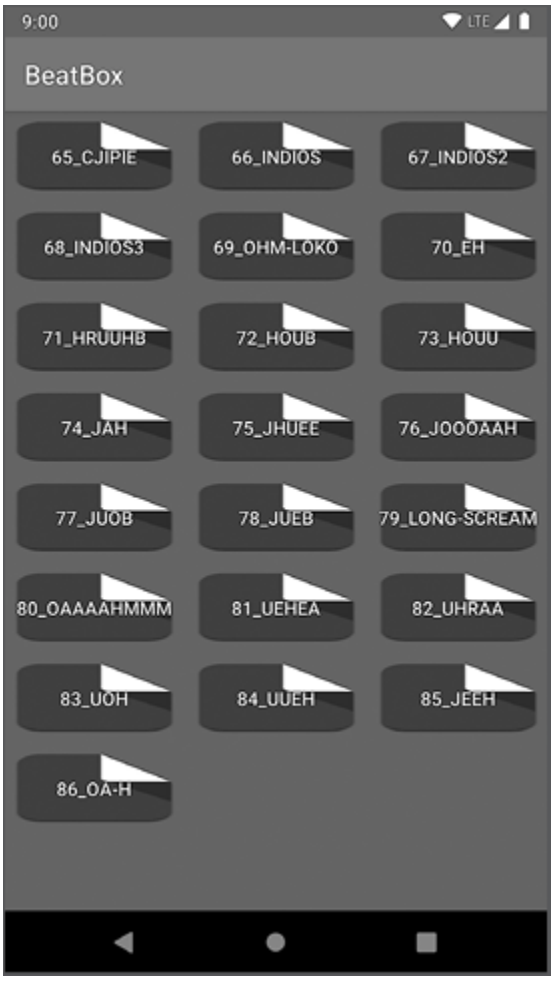


Рис. 22.8. Искажение вида кнопок

И что получилось?.. Да ничего хорошего.

Почему результат плохо выглядит? Android равномерно растягивает изображение `ic_button_beat_box_default.png`, включая загнутый уголок и закругления. Было бы лучше, если бы вы могли явно указать, какие части изображения можно растягивать, а какие должны остаться в исходном виде. На помощь приходят 9-зонные изображения.

Файл 9-зонного изображения специально форматируется так, чтобы система Android знала, какие части изображения можно или нельзя форматировать. При правильной реализации

это гарантирует, что края и углы фона будут соответствовать изображению в том виде, в каком оно было создано.

Почему изображения называются 9-зонными? Такие изображения разбиваются сеткой 3 × 3, то есть сетка состоит из 9 элементов, или *зон* (*patches*). Углы сетки не масштабируются, стороны масштабируются только в одном направлении, а центральная зона масштабируется в обоих направлениях (рис. 22.9).

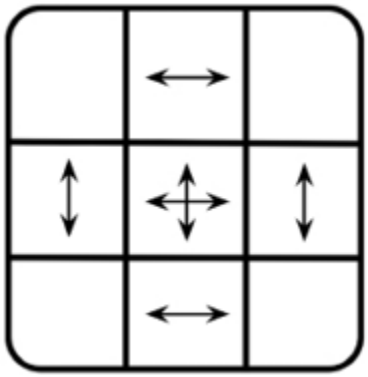


Рис. 22.9. Как работает девятизонное изображение

Девятизонное изображение во всех отношениях напоминает обычный файл `png`, за исключением двух аспектов: имя файла завершается суффиксом `.9.png` и изображение имеет дополнительную рамку толщиной в один пиксел. Рамка задает местонахождение центрального квадрата. Черными пикселами рамки обозначается центр, а прозрачными — края.

Девятизонное изображение можно создать в любом графическом редакторе, но проще воспользоваться программой `draw9patch`, включенной в поставку Android SDK, или средствами Android Studio.

Сначала преобразуйте два новых фоновых изображения в 9-зонный формат: щелкните правой кнопкой мыши по файлу `ic_button_beat_box_default.png` на панели **Project** и выберите команду **Refactor⇒Rename** в контекстном меню.

Переименуйте файл в `ic_button_beat_box_default.9.png`. (Если Android Studio предупредит вас о том, что такой файл уже существует, нажмите кнопку **Continue**.) Затем повторите процесс и переименуйте файл с «нажатой» версией изображения в `ic_button_beat_box_pressed.9.png`.

Дважды щелкните мышью по стандартному изображениию на панели **Project**, чтобы открыть его во встроенном в Android Studio редакторе 9-зонных изображений (рис. 22.10). (Если Android Studio не откроет редактор, закройте файл и сверните папку `drawable` на панели **Project**, после чего откройте изображение заново.)

В 9-зонном редакторе сначала установите флажок **Showpatches**, чтобы зоны были лучше видны. Заполните черные пикселы на верхней и левой границе, чтобы обозначить растягиваемые области изображения, как показано на рис. 22.10. Также перетащите стороны цветной накладки в соответствии с иллюстрацией.

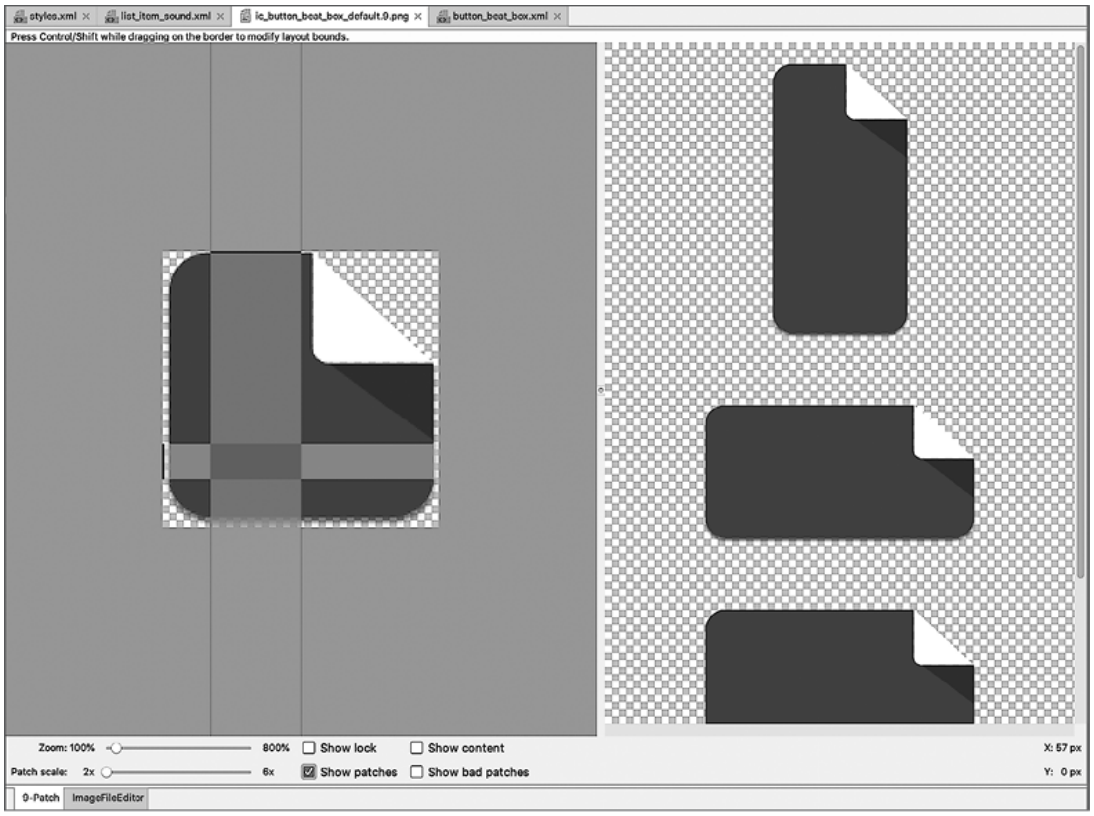


Рис. 22.10. Создание 9-зонного изображения

Черная линия в верхней части изображения определяет регион, который должен растягиваться при растяжении изображения по горизонтали. Линия слева показывает, какие пикселы должны растягиваться при растяжении изображения по вертикали. В области предварительного просмотра справа показано, как изображение будет выглядеть при различных вариантах растяжения.

Повторите процедуру с «нажатой» версией изображения. Запустите приложение BeatBox и посмотрите, как работает новое 9-зонное изображение (рис. 22.11).

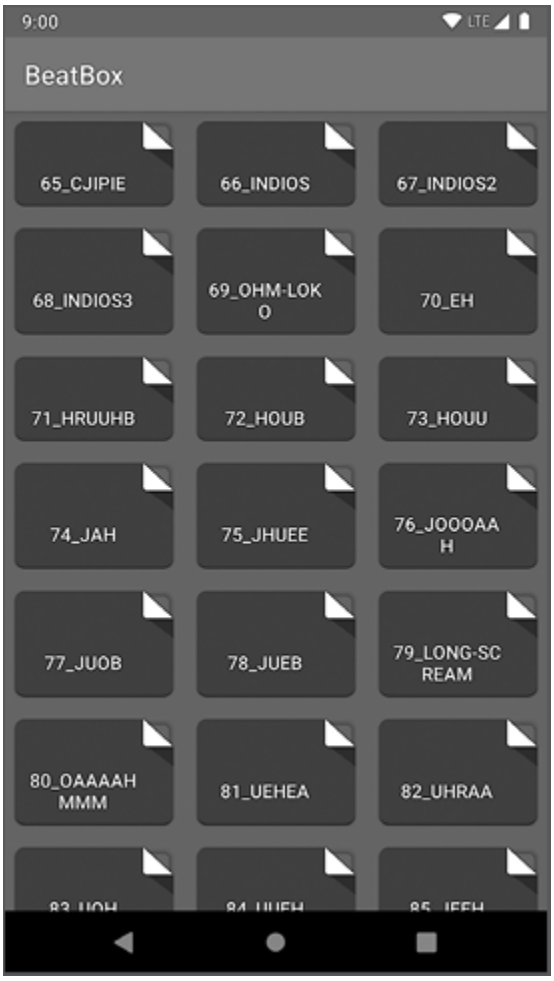


Рис. 22.11. Новая улучшенная версия

Итак, левый и верхний участки границы отмечают растягиваемую область изображения. А как насчет правого и нижнего участка? Они отмечают необязательную область содержимого — область, в которой должно выводиться некое содержимое (обычно текст). Если область содержимого не задана, по умолчанию считается, что она совпадает с растягиваемой областью.

Воспользуемся областью содержимого для выравнивания по центру текста внутри кнопок под сложенным уголком. Вернитесь к файлу `ic_button_beat_box_default.9.png` и добавьте линии внизу и справа, как показано на рис. 22.12. Установите флажок **Showcontent** в редакторе 9-зонных

изображений. На экране выделяются области изображения, которые будут содержать текст.

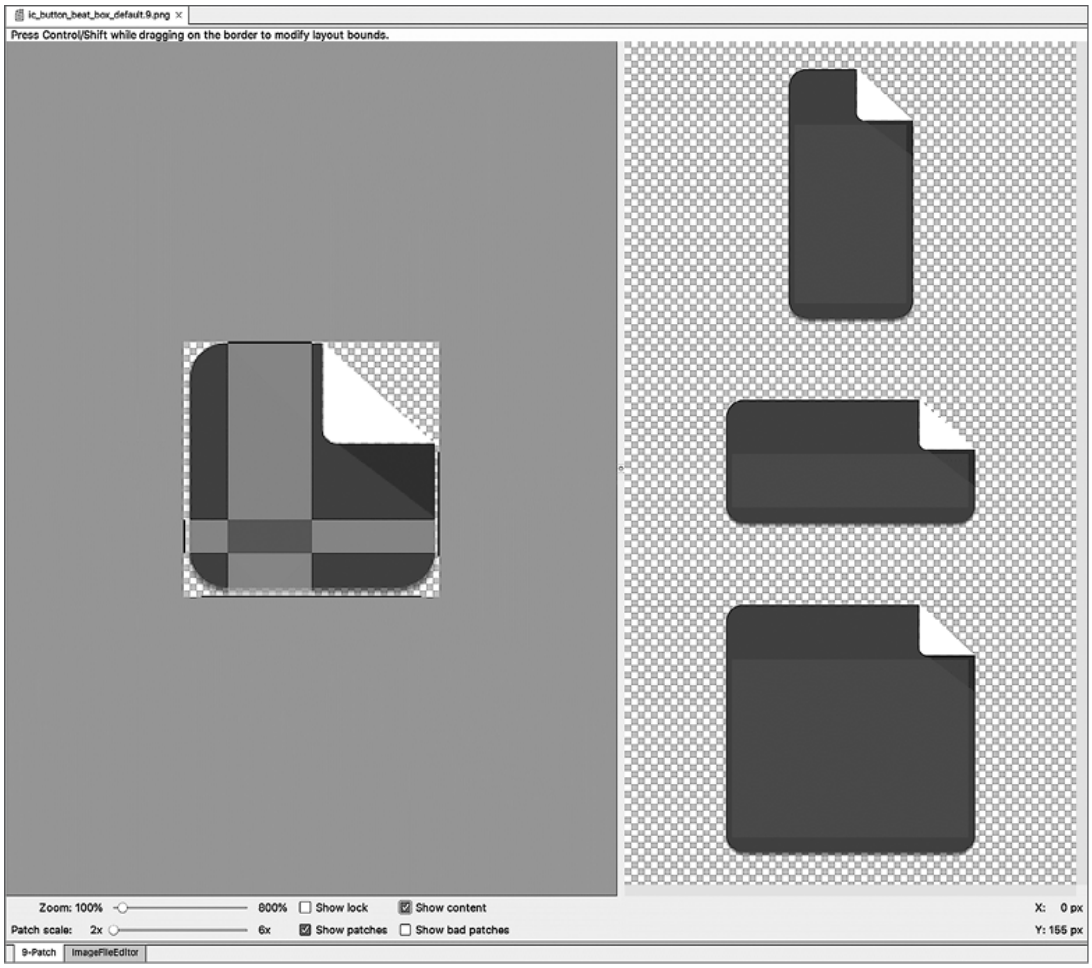


Рис. 22.12. Определение области содержимого

Повторите процесс для «нажатой» версии изображения. Внимательно проследите за тем, чтобы оба изображения обновились с правильными границами области содержимого. Когда 9-зонные изображения задаются при помощи графических объектов списка состояний (как в BeatBox), область содержимого может работать не так, как вы ожидаете.

Android задает область содержимого при инициализации фона и не изменяет ее при нажатии кнопки. Таким образом, область содержимого одного из двух изображений будет игнорироваться! Изображение, которое Android будет использовать для заполнения области содержимого, не задано, поэтому будет лучше, если все ваши 9-зонные изображения в списке состояний будут использовать одну область содержимого.

Запустите BeatBox — текст аккуратно выравнивается по центру (рис. 22.13).

Поверните устройство в альбомную ориентацию. Изображения растягиваются еще сильнее, но фон кнопки все равно выглядит хорошо, а текст остается выровненным по центру.

Упражнение. Темы кнопок

После обновления 9-зонного изображения можно заметить, что с фоном кнопок что-то не так. За загнутым уголком виднеется нечто похожее на тень. Когда вы нажимаете кнопку, кажется, что она приближается к вашему пальцу.

Рис. 22.13. Дополнительные усовершенствования

Ваша задача — убрать тень. Используйте свои навыки исследования тем для определения того, как применяется эта тень. Существует ли другой стиль кнопки, который можно использовать в качестве родителя для стиля `BeatBoxButton`?

23. Подробнее об интентах и задачах

В этой главе мы используем неявные интенты для создания приложения-лаунчера, заменяющего стандартный лаунчер Android. На рис. 23.1 показано, как будет выглядеть приложение NerdLauncher.

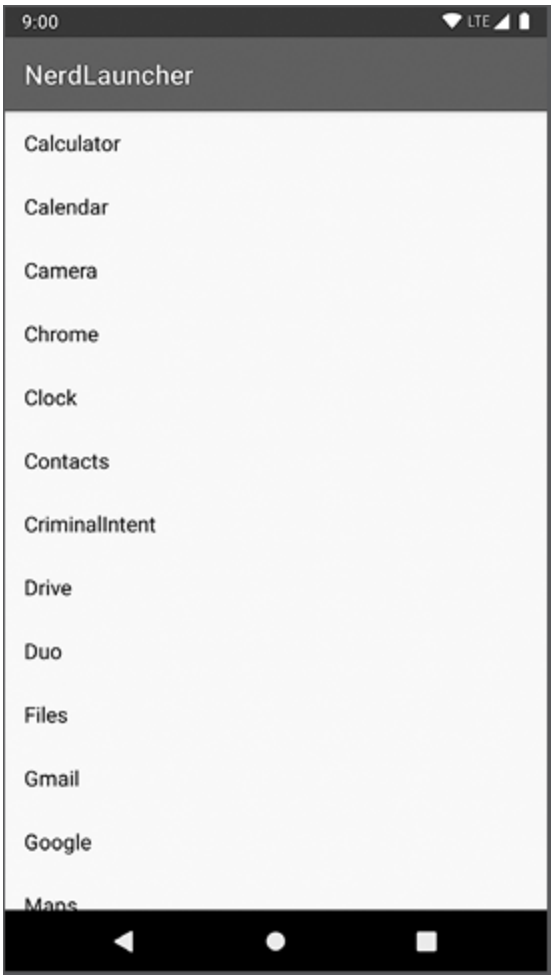


Рис. 23.1. Итоговый вид NerdLauncher

NerdLauncher выводит список приложений на устройстве. Пользователь нажимает элемент списка, чтобы запустить соответствующее приложение.

Чтобы приложение работало правильно, нам придется углубить свое понимание интентов, фильтров интентов и схем взаимодействий между приложениями в среде Android.

Создание приложения NerdLauncher

В Android Studio выберите команду **File⇒NewProject**, чтобы создать новый проект. Выберите пункт **AddNoActivity** на вкладке **PhoneandTablet**. Присвойте приложению название **NerdLauncher** и назначьте имя пакета `com.bignerdranch.android.nerdlauncher`. Установите флажок **UseAndroidXartifacts**, а остальные настройки не изменяйте.

После инициализации проекта в Android Studio создайте новую пустую activity, выбрав команду **File⇒New⇒Activity⇒EmptyActivity**. Присвойте activity имя `NerdLauncherActivity` и установите флажок **LauncherActivity**.

`NerdLauncherActivity` отображает список названий приложений в `RecyclerView`. Добавьте зависимость `androidx.recyclingerview:recyclingerview:1.0.0` в файл `app/build.gradle`, как вы делали это в главе 9. Если вы хотите использовать более новые версии RecyclerView, их можно найти по ссылке **developer.android.com/jetpack/androidx/releases/reecycleerview**.

Измените содержимое файла `res/layout/activity_nerd_launcher.xml` в части кода `RecyclerView`, как показано в листинге 23.1.

Листинг 23.1. Обновление макета NerdLauncherActivity (layout/activity_nerd_launcher.xml)

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<androidx.recyclerview.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/app_recycler_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Откройте файл NerdLauncherActivity.kt и спрячьте ссылку на объект RecyclerView в свойстве (уже скоро мы подключим данные к RecyclerView).

Листинг 23.2. Базовая реализация NerdLauncherActivity (NerdLauncherActivity.kt)

```
class NerdLauncherActivity : AppCompatActivity() {

    private lateinit var recyclerView: RecyclerView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_nerd_launcher)

        recyclerView = findViewById(R.id.app_recycler_view)
        recyclerView.layoutManager = LinearLayoutManager(this)
    }
}
```

Запустите приложение и убедитесь в том, что пока все компоненты взаимодействуют правильно. Если все сделано без ошибок, вы становитесь владельцем приложения NerdLauncher, в котором отображается пустой виджет `RecyclerView` (рис. 23.2).

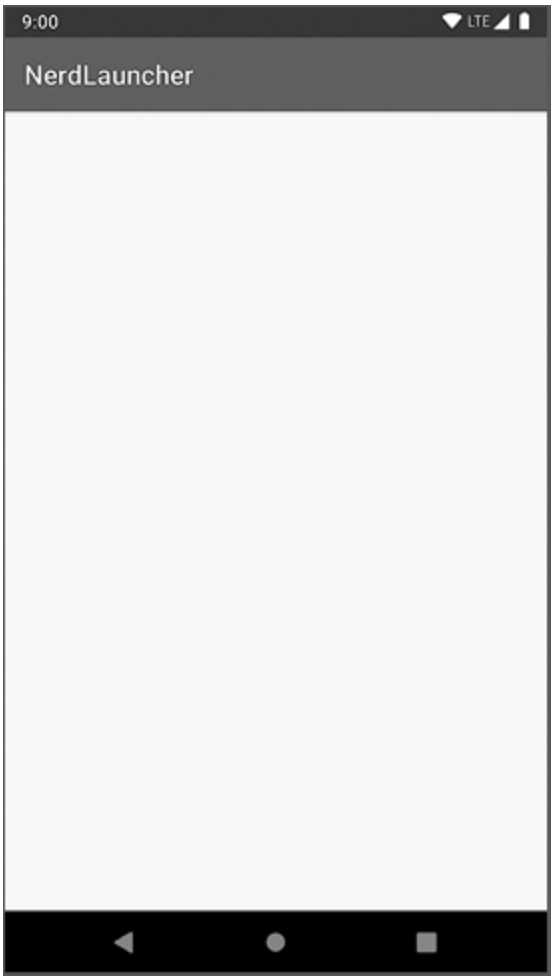


Рис. 23.2. NerdLauncher — начало

Обработка неявного интента

NerdLauncher отображает список запускаемых (*launchable*) приложений на устройстве. («Запускаемым» называется приложение, которое может быть запущено пользователем,

если он щелкнет на значке на «Главном экране» или на экране лаунчера.) Для этого NerdLauncher запрашивает у системы список запускаемых главных activity.

`PackageManager`, о котором мы говорили в главе 15, используется для разрешения activity. У запускаемых главных activity фильтры интентов включают действие `MAIN` и категорию `LAUNCHER`. Вы уже видели в своих проектах фильтр интентов в файле `manifests/AndroidManifest.xml`:

```
<intent-filter>
                                        <action
android:name="android.intent.action.MAIN" />
                                      <category
android:name="android.intent.category.LAUNCHER"
/>
</intent-filter>
```

Когда `NerdLauncherActivity` стал запускающей activity, фильтры интентов добавляются автоматически. (Проверьте манифест, если хотите.)

В файле `NerdLauncherActivity.kt` добавьте функцию `setupAdapter()` и вызовите его из `onCreateView(...)`. (Позднее эта функция создаст экземпляр `RecyclerView.Adapter` и назначит его объекту `RecyclerView`, но пока она просто генерирует список данных приложения.)

Также создайте неявный интент и получите список activity, соответствующих интенту, от `PackageManager`. Пока мы ограничимся простой регистрацией количества activity, возвращенных `PackageManager`.

Листинг 23.3. Получение информации у PackageManager (NerdLauncherActivity.kt)

```
private const val TAG = "NerdLauncherActivity"
class NerdLauncherActivity : AppCompatActivity() {

    private lateinit var recyclerView: RecyclerView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_nerd_launcher)

        recyclerView = findViewById(R.id.app_recycler_view)
        recyclerView.layoutManager = LinearLayoutManager(this)

        setupAdapter()
    }

    private fun setupAdapter() {
        val startupIntent = Intent(Intent.ACTION_MAIN).apply {
            addCategory(Intent.CATEGORY_LAUNCHER)
        }

        val activities = packageManager.queryIntentActivities(startupInt
```

```
ent, 0)

            Log.i(TAG, "Found ${activities.size}
activities")
        }
}
```

Здесь мы создаем неявный интент с заданным действием `ACTION_MAIN`. Переменная `CATEGORY_LAUNCHER` добавлена в категории интента.

Вызов `PackageManager.requestIntentActivities(Intent,Int)` возвращает список, содержащий `ResolveInfo` для всех activity, у которых есть фильтр, соответствующий данному интенту. Вы можете указать флаги для изменения результатов. Например, флаг `PackageManager.GET_SHARED_LIBRARY_FILES` заставляет запрос включать в результаты дополнительные данные (пути к библиотекам, которые связаны с каждым приложением, удовлетворяющим требованиям). Здесь вы передаете `0`, что указывает на то, что вы не хотите изменять результаты.

Запустите приложение NerdLauncher и посмотрите в выводе **LogCat**, сколько приложений вернул экземпляр `PackageManager` (у нас при первом пробном запуске их было 30).

В CriminalIntent для отправки отчетов использовался неявный интент. Чтобы представить на экране список выбора приложений, мы создали неявный интент, упаковали его в интент выбора и отправили ОС вызовом `startActivity(Intent)`:

```
val intent = Intent(Intent.ACTION_SEND)
```

```
... // Создание и размещение дополнений интентов
chooserIntent = Intent.createChooser(intent, getString(R.string.send_report)
startActivity(chooserIntent)
```

Почему мы не используем этот подход здесь? Вкратце: дело в том, что фильтр интентов MAIN/LAUNCHER может соответствовать или не соответствовать неявному интенту MAIN/LAUNCHER, отправленному через startActivity(Intent).

Оказывается, вызов startActivity(Intent) не означает «Запустить activity, соответствующую этому неявному интенту». Он означает «Запустить activity *поумолчанию*, соответствующую этому неявному интенту». Когда вы отправляете неявный интент с использованием startActivityForResult(Intent) (или startActivity(...)), ОС незаметно включает в интент категорию Intent.CATEGORY_DEFAULT.

Таким образом, если вы хотите, чтобы фильтр интентов соответствовал неявным интентам, отправленным через startActivity(Intent), вы должны включить в этот фильтр интентов категорию DEFAULT.

Activity с фильтром интентов MAIN/LAUNCHER является главной точкой входа приложения, которому она принадлежит. Для нее важно лишь то, что она является главной точкой входа приложения, а является ли она главной точкой входа «по умолчанию» — несущественно, поэтому она не обязана включать категорию CATEGORY_DEFAULT.

Так как фильтры интентов MAIN/LAUNCHER могут не включать CATEGORY_DEFAULT, надежность их соответствия неявным интентам, отправленным вызовом start-

Activity(Intent), не гарантирована. Поэтому мы используем интент для прямого запроса у PackageManager информации об activity с фильтром интентов MAIN/LAUNCHER.

Следующий шаг — отображение меток этих activity в списке RecyclerView экземпляра NerdLauncherFragment. *Метка* (*label*) activity представляет собой отображаемое имя — нечто, понятное пользователю. Если учесть, что эти activity относятся к лаунчеру, такой меткой, скорее всего, должно быть имя приложения.

Метки activity вместе с другими метаданными содержатся в объектах ResolveInfo, возвращаемых PackageManager.

Сначала отсортируйте объекты ResolveInfo, возвращаемые PackageManager, в алфавитном порядке меток, получаемых функцией ResolveInfo.loadLabel(PackageManager).

Листинг 23.4. Алфавитная сортировка (NerdLauncherActivity.kt)

```
class NerdLauncherActivity : AppCompatActivity() {
    ...
    private fun setupAdapter() {
        val startupIntent = Intent(Intent.ACTION_MAIN).apply {
            addCategory(Intent.CATEGORY_LAUNCHER)
        }

        val activities = packageManager.queryIntentActivities(startupIntent, 0)
```

```
            activities.sortWith(Comparator { a, b ->
                String.CASE_INSENSITIVE_ORDER.compare(
                    a.loadLabel(packageManager).toString(),
                    b.loadLabel(packageManager).toString()
                )
            })

            Log.i(TAG, "Found ${activities.size} activities")
        }
    }
```

Теперь определите класс `ViewHolder` для отображения метки activity. Сохраните объект `ResolveInfo` activity в переменной класса (позднее мы еще не раз используем его).

Листинг 23.5. Реализация ViewHolder (NerdLauncherActivity.kt)

```
class NerdLauncherActivity : AppCompatActivity() {
    ...
    private fun setupAdapter() {
        ...
    }

    private class ActivityHolder(itemView: View) :
            RecyclerView.ViewHolder(itemView) {
```

```
        private val nameTextView = itemView as
TextView
            private lateinit var resolveInfo:
ResolveInfo

                fun bindActivity(resolveInfo:
ResolveInfo) {
            this.resolveInfo = resolveInfo
                        val packageManager =
itemView.context.packageManager
                            val appName =
resolveInfo.loadLabel(packageManager).toString(
)
            nameTextView.text = appName
        }
    }
}
```

Затем добавьте реализацию `RecyclerView.Adapter`.

Листинг 23.6. Реализация RecyclerView.Adapter (NerdLauncherActivity.kt)

```
class NerdLauncherActivity :
AppCompatActivity() {
    ...
        private class ActivityHolder(itemView:
View) :
            RecyclerView.ViewHolder(itemView) {
            ...
    }
```

```
        private class ActivityAdapter(val
activities: List<ResolveInfo>) :
            RecyclerView.Adapter<ActivityHolder
>() {

                                    override fun
onCreateViewHolder(container:       ViewGroup,
viewType: Int):
                ActivityHolder {
                            val layoutInflater =
LayoutInflater.from(container.context)
            val view = layoutInflater
                .inflate(android.R.layout.simpl
e_list_item_1, container, false)
            return ActivityHolder(view)
        }

          override fun onBindViewHolder(holder:
ActivityHolder, position: Int) {
                              val resolveInfo =
activities[position]
            holder.bindActivity(resolveInfo)
        }

        override fun getItemCount(): Int {
            return activities.size
        }
    }
}
```

Здесь мы заполняем файл android.R.layout.simple_list_item_1 в функции onCreateViewHolder(...). Файл simple_list_item_1layout является частью фреймворка Android, поэтому вы ссылаетесь на него как на layoutandroid.R.layout, а не как на R.layout. В нем содержится один TextView.

Наконец, измените код функции setupAdapter(), чтобы она создавала экземпляр ActivityAdapter и назначала его адаптером RecyclerView.

Листинг 23.7. Назначение адаптера RecyclerView (NerdLauncherActivity.kt)

```
class NerdLauncherActivity : AppCompatActivity() {
    ...
    private fun setupAdapter() {
        ...
            Log.i(TAG, "Found ${activities.size} activities")
                        recyclerView.adapter = ActivityAdapter(activities)
    }
    ...
}
```

Запустите NerdLauncher; вы увидите список RecyclerView, заполненный метками activity (рис. 23.3).

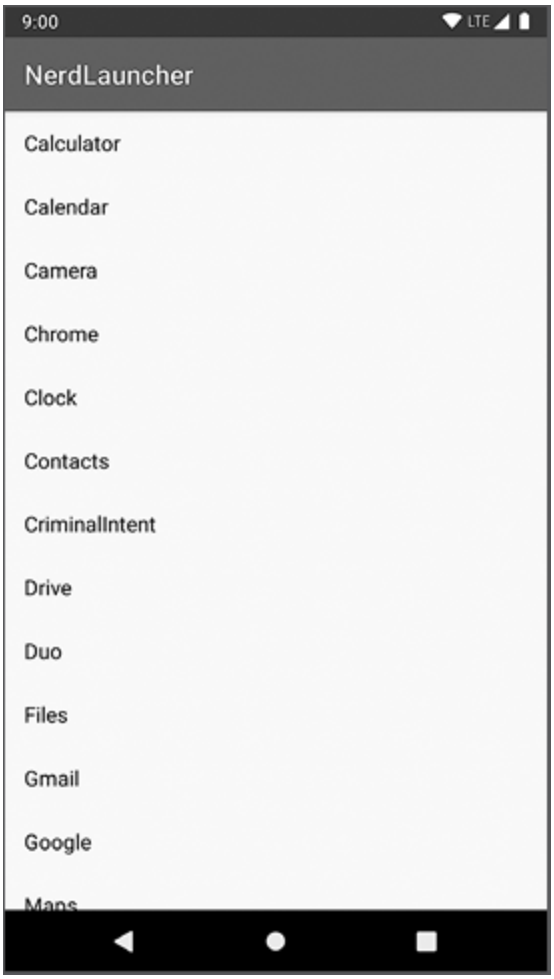


Рис. 23.3. Список activity

Создание явных интентов на стадии выполнения

Мы использовали неявный интент для сбора информации об activity и выводе ее в формате списка. Следующим шагом должен стать запуск выбранной activity при нажатии пользователем на элементе списка. Для запуска activity будет использоваться явный интент.

Для создания явного интента необходимо извлечь из `ResolveInfo` имя пакета и имя класса activity. Эти данные можно получить из части `ResolveInfo` с именем `ActivityInfo`. (О том, какие данные доступны в разных частях `ResolveInfo`, можно узнать из документации: **developer.android.com/reference/kotlin/android/content/pm/ResolveInfo.html**.)

Реализуйте в `ActivityHolder` слушателя нажатий. При нажатии на activity в списке по данным `ActivityInfo` этой activity создайте явный интент. Затем используйте этот явный интент для запуска выбранной activity.

Листинг 23.8. Запуск выбранной activity (NerdLauncherActivity.kt)

```
class NerdLauncherActivity : AppCompatActivity() {
    ...
        private class ActivityHolder(itemView: View) :
                RecyclerView.ViewHolder(itemView),
                View.OnClickListener {

            private val nameTextView = itemView as TextView
            private lateinit var resolveInfo: ResolveInfo

            init {
                nameTextView.setOnClickListener(this)
            }
```

```
                fun bindActivity(resolveInfo:
ResolveInfo) {
            ...
        }

        override fun onClick(view: View) {
                        val activityInfo =
resolveInfo.activityInfo

                                val intent =
Intent(Intent.ACTION_MAIN).apply {
                setClassName(activityInfo.appli
cationInfo.packageName,
                    activityInfo.name)
            }

            val context = view.context
            context.startActivity(intent)
        }
    }
    ...
}
```

Обратите внимание: в этом интенте мы отправляем действие как часть явного интента. Большинство приложений ведет себя одинаково независимо от того, включено действие или нет, однако некоторые приложения могут изменять свое поведение. Одна и та же activity может отображать разные интерфейсы в зависимости от того, как она была запущена. Вам как программисту лучше всего четко объявить свои намерения и позволить activity запуститься так, как они считают нужным.

В листинге 23.8 мы получаем имя пакета и имя класса из метаданных и используем их для создания явной activity функцией Intent:

```
fun setClassName(packageName: String, className: String): Intent
```

Этот способ отличается от того, который использовался нами для создания явных интентов в прошлом. Ранее мы использовали конструктор Intent, получающий объекты Context и Class:

```
Intent(packageContext: Context, cls: Class<?>)
```

Этот конструктор использует свои параметры для получения того, в чем Intent реально нуждается — ComponentName, имени пакета, объединенного с именем класса. Когда вы передаете Activity и Class для создания Intent, конструктор определяет полное имя пакета по Activity.

```
fun setComponent(component: ComponentName): Intent
```

Однако решение с функцией setClassName(...), автоматически создающей имя компонента, получается более компактным.

Запустите NerdLauncher и посмотрите, как работает запуск приложений.

Задачи и стек возврата

Android использует задачи для отслеживания текущего состояния пользователя в каждом выполняемом приложении. Каждому приложению, открытому из стандартного лаунчера

Android, назначается собственная задача. К сожалению для NerdLauncher, это вполне логичное поведение не используется по умолчанию. Прежде чем разбираться, как заставить приложение запуститься в отдельной задаче, следует понять, что такое задачи и как они работают.

Задача (*task*) представляет собой стек activity, с которыми имеет дело пользователь. Activity в нижней позиции стека называется *базовой activity*, а activity в верхней позиции видна пользователю. При нажатии кнопки «Назад» верхняя activity извлекается из стека. Если нажать кнопку «Назад» при просмотре базовой activity, вы вернетесь к главному экрану.

По умолчанию новые activity запускаются в текущей задаче. В приложении CriminalIntent все запускавшиеся activity добавлялись к текущей задаче (рис. 23.4). Это относилось даже к activity, которые не являлись частью приложения CriminalIntent (например, при запуске activity для отправки отчета).

Рис. 23.4. Задача CriminalIntent

Преимущество добавления activity к текущей задаче заключается в том, что пользователь может выполнять обратную навигацию внутри задачи, а не в иерархии приложений (рис. 23.5).

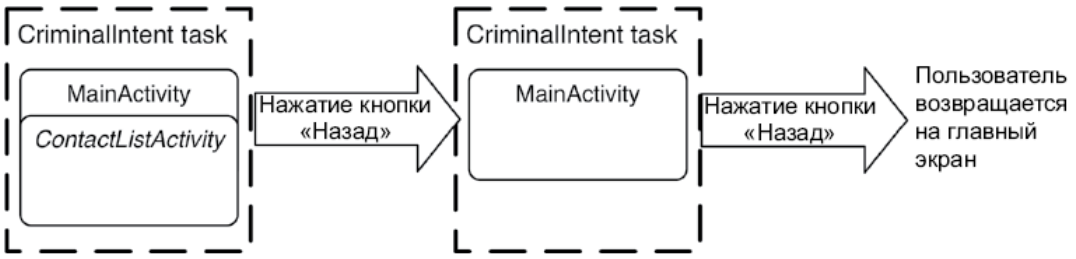


Рис. 23.5. Нажатие кнопки «Назад» в CriminalIntent

Переключение между задачами

Экран «Обзор приложений» позволяет переключаться между задачами без изменения состояния каждой задачи. Например, если вы начинаете вводить новый контакт и переключаетесь на проверку своей публикации в «Твиттере», будут запущены две задачи. При переключении на редактирование контактов сохраняется ваша текущая позиция в обеих задачах.

Попробуйте переключить задачи с помощью экрана «Обзор приложений» на вашем устройстве или эмуляторе. Сначала запустите CriminalIntent с главного экрана или из списка приложений. (Если на вашем устройстве или эмуляторе нет установленной программы CrimeIntent, откройте проект CriminalIntent в Android Studio и запустите его оттуда. Если вы пропустили главы про приложение CriminalIntent и не создали приложение, код можно найти в файле решений по ссылке **www.bignerdranch.com/solutions/AndroidProgramming4e.zip**.)

Выберите преступление из списка. Затем нажмите кнопку «Главный экран», чтобы вернуться на главный экран. Затем запустите BeatBox с главного экрана или из списка приложений (или, при необходимости, из Android Studio). Наконец, откройте экран «Обзор приложений» с помощью соответствующей кнопки (рядом с кнопкой «Главный экран») (рис. 23.6).

Экран «Обзор приложений», показанный на рис. 23.6 слева, — это то, что пользователи увидят в версии Android Nougat (уровень API 24). Экран «Обзор приложений» справа — вариант для Android Pie (уровень API 28) (если у них не включена опция **SwipeuponHome**, о чем мы говорили в главе 3).

В обоих случаях запись, отображаемая для каждого приложения (*карточка*), — это задача приложения. Отображается скриншот activity в верхней части обратного стека каждой задачи. Вы можете нажать на карточку BeatBox или CriminalIntent, чтобы вернуться к приложению (и к любой activity, с которой вы взаимодействовали в этом приложении).

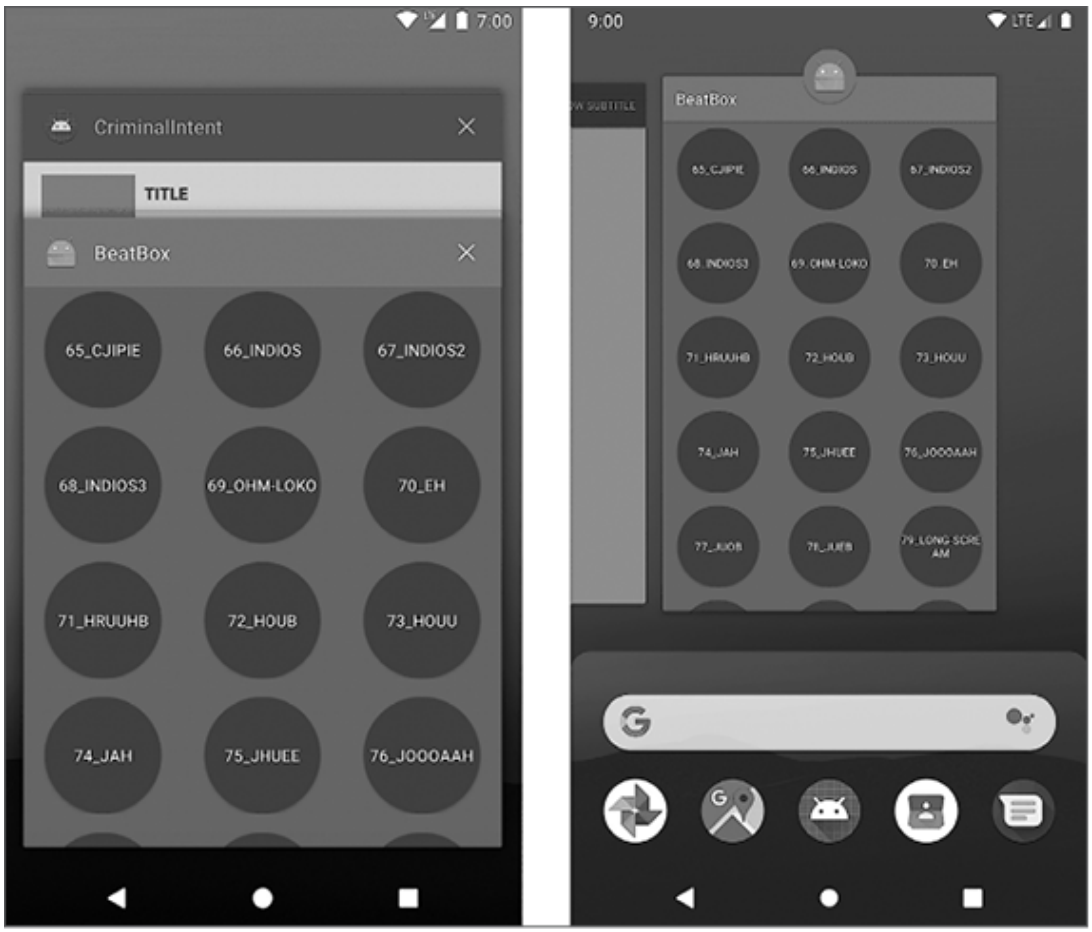


Рис. 23.6. Экран «Обзор приложений» в версии Nougat (слева) и Pie (справа)

Вы можете удалить из памяти задачу приложения; для этого следует провести пальцем по карточке задачи, чтобы удалить карточку из списка задач. При удалении задачи все ее activity исключаются из стека возврата приложения.

Удалите задачу CriminalIntent и перезапустите приложение. Вы увидите список преступлений вместо того преступления, которое редактировалось до удаления задачи.

Запуск новой задачи

Иногда запускаемая activity должна добавляться к текущей задаче. В других случаях она должна запускаться в новой задаче, независимой от запустившей ее activity.

В текущей версии все activity, запускаемые из NerdLauncher, добавляются в задачу NerdLauncher, как показано на рис. 23.7.

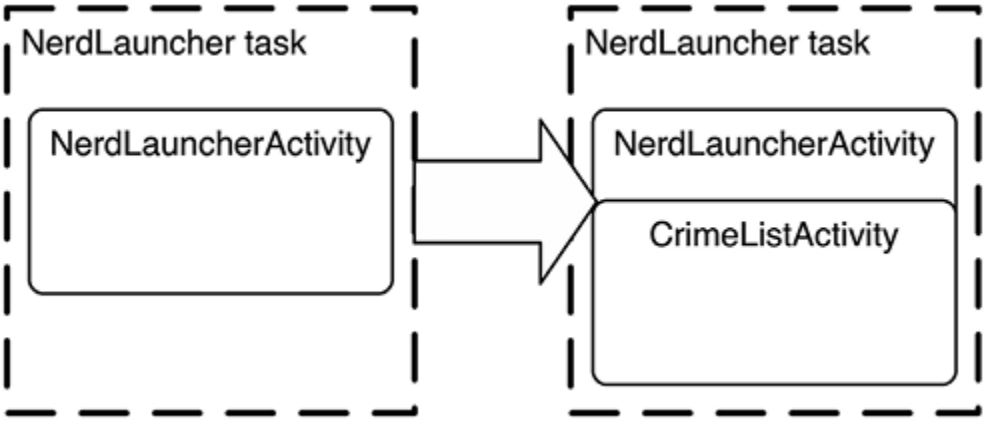


Рис. 23.7. Задача NerdLauncher содержит CriminalIntent

Чтобы убедиться в этом, удалите все задачи, отображаемые на экране «Обзор приложений». Запустите NerdLauncher и щелкните мышью по элементу CriminalIntent, чтобы запустить приложение CriminalIntent. Снова откройте экран «Обзор

приложений» — вы найдете на нем только одну задачу, хотя запускали другое приложение.

Запущенная activity `MainActivity` в CriminalIntent была добавлена в задачу NerdLauncher (рис. 23.8). Нажатие на задаче NerdLauncher вернет вас к экрану CriminalIntent, который вы просматривали перед переходом к экрану «Обзор приложений».

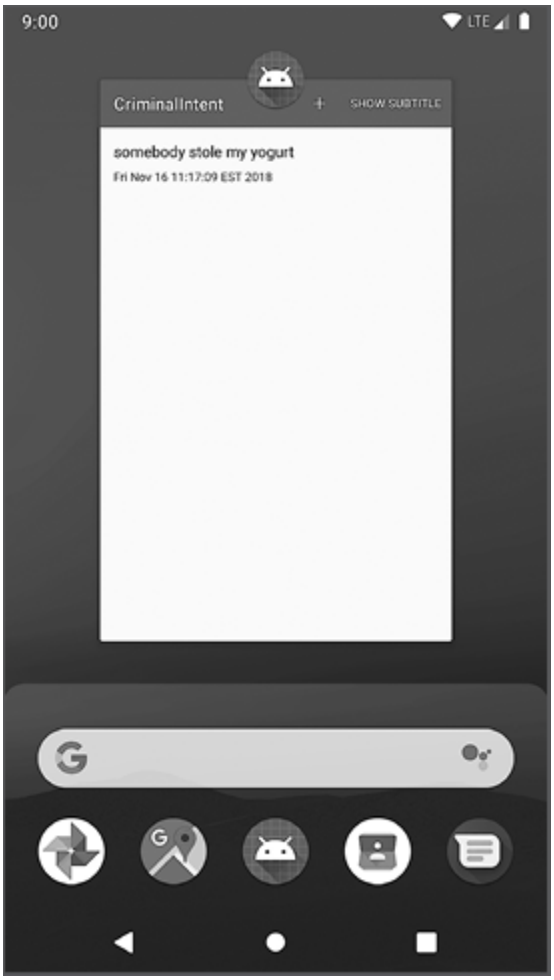


Рис. 23.8. Приложение CriminalIntent не выполняется в отдельной задаче

Мы хотим, чтобы приложение NerdLauncher запускало activity в новых задачах (рис. 23.9). Далее пользователь может переключаться между выполняемыми приложениями так, как считает нужным (при помощи экрана «Обзор приложений», «Главный экран» или NerdLauncher).

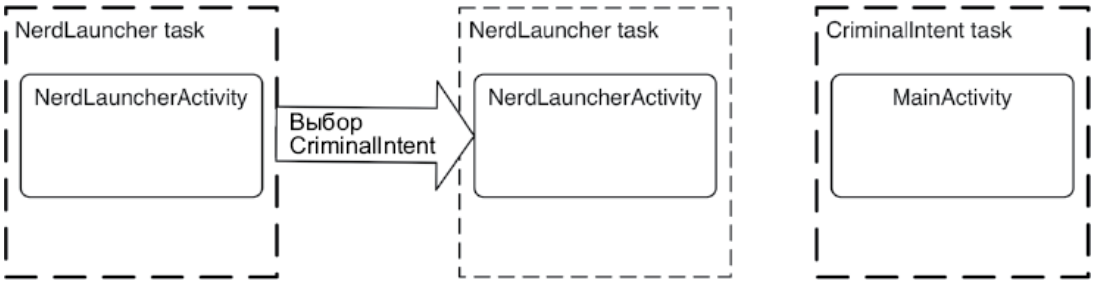


Рис. 23.9. Запуск CriminalIntent в отдельной задаче

Чтобы при запуске новой activity запускалась новая задача, следует добавить в интент соответствующий флаг в файле NerdLauncherActivity.kt.

Листинг 23.9. Добавление флага новой задачи в интент (NerdLauncherActivity.kt)

```
class NerdLauncherActivity : AppCompatActivity() {
    ...
        private class ActivityHolder(itemView: View) :
            RecyclerView.ViewHolder(itemView),
            View.OnClickListener {
        ...
        override fun onClick(view: View) {
                        val activityInfo = resolveInfo.activityInfo

                                val intent = Intent(Intent.ACTION_MAIN).apply {
                    setClassName(activityInfo.applicationInfo.packageName,
                        activityInfo.name)
```

```
                    addFlags(Intent.FLAG_ACTIVITY_N
EW_TASK)
                }

                val context = view.context
                context.startActivity(intent)
            }
        }
        ...
}
```

Удалите задачи на экране «Обзор приложений». Запустите приложение NerdLauncher и выберите CriminalIntent. На этот раз при вызове экрана «Обзор приложений» становится видно, что CriminalIntent выполняется в отдельной задаче (рис. 23.10).

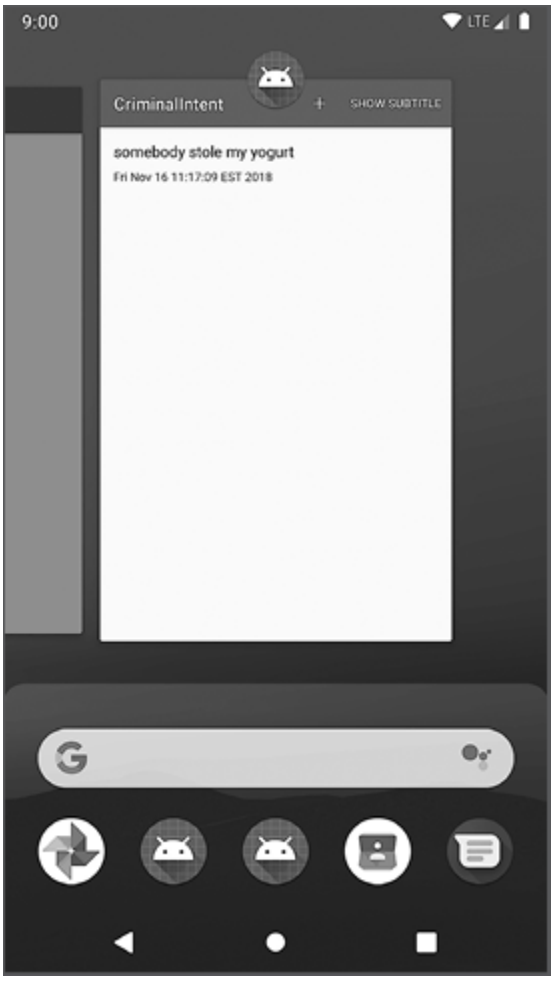


Рис. 23.10. CriminalIntent выполняется в собственной задаче

Повторный запуск CriminalIntent из NerdLauncher не приведет к созданию второй задачи CriminalIntent. Флаг `FLAG_ACTIVITY_NEW_TASK` создает только одну задачу на activity. У `MainActivity` уже имеется работающая задача, поэтому Android переключится на эту задачу вместо запуска новой.

Убедитесь в этом. Откройте экран с подробной информацией об одном из преступлений в CriminalIntent. Используйте экран «Обзор приложений» для переключения на NerdLauncher. Нажмите на элемент CriminalIntent в списке. Вы вернетесь к прежнему состоянию в приложении CriminalIntent:

просмотру подробной информации об отдельном преступлении.

Использование NerdLauncher в качестве главного экрана

Но кому захочется запускать приложение, чтобы запускать другие приложения? Гораздо логичнее использовать NerdLauncher как альтернативу главного экрана устройства. Откройте файл `AndroidManifest.xml` в NerdLauncher и добавьте следующий фрагмент в главный фильтр интентов.

Листинг 23.10. Изменение категорий NerdLauncherActivity (manifests/AndroidManifest.xml)

```
<intent-filter>
                                        <action
android:name="android.intent.action.MAIN" />
                                       <category
android:name="android.intent.category.LAUNCHER"
/>
                                       <category
android:name="android.intent.category.HOME" />
                                       <category
android:name="android.intent.category.DEFAULT"
/>
</intent-filter>
```

Добавление категорий `HOME` и `DEFAULT` означает, что `NerdLauncherActivity` должна включаться в число вариантов главного экрана. Нажмите кнопку «Главный экран», и вам будет предложено использовать NerdLauncher (рис. 23.11).

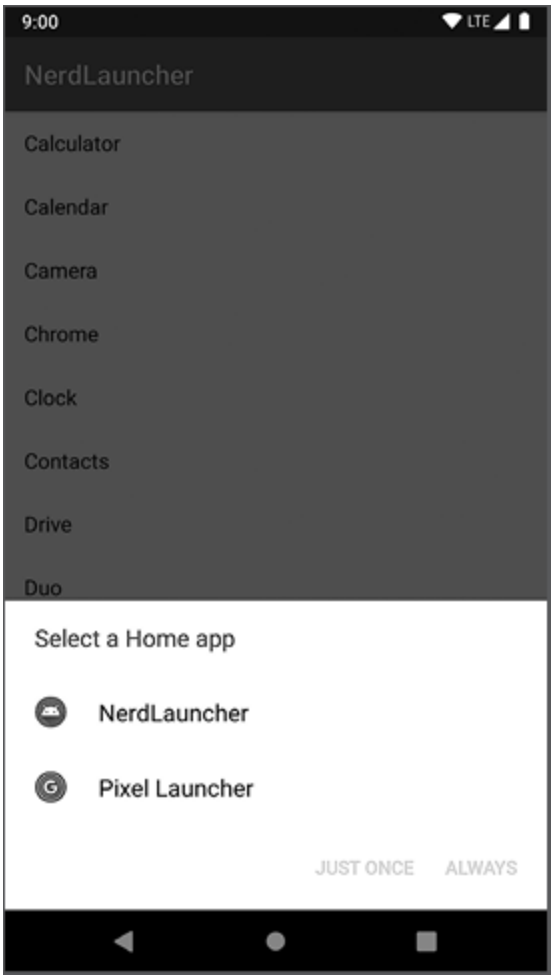


Рис. 23.11. Выбор приложения для главного экрана

(Если вы назначите NerdLauncher главным экраном, а потом захотите отменить свой выбор, запустите приложение Settings из NerdLauncher. Перейдите к экрану **Settings⇒Apps&Notification** и выберите в списке приложений пункт **NerdLauncher**. Если вы не видите пункт **NerdLauncher**, коснитесь ссылки **Seeallapps**, найдите NerdLauncher и выберите его.

Выбрав NerdLauncher, вы окажетесь на экране **AppInfo**. Раскройте категорию настроек **Advanced** и выберите пункт **Openbydefault**. Нажмите кнопку **CLEARDEFAULTS**. При следующем нажатии кнопки «Главный экран» вы сможете выбрать новый главный экран по умолчанию.)

Для любознательных: процессы и задачи

Для существования любого объекта необходима память и виртуальная машина. *Процесс* представляет собой пространство, выделенное в ОС, в котором существуют объекты вашего приложения и в котором выполняется само приложение.

Процессам могут принадлежать ресурсы, находящиеся под управлением ОС, — память, сетевые сокеты, открытые файлы и т.д. Процесс также содержит минимум один (а вероятно, несколько) программный поток (thread). На платформе Android процесс всегда выполняется исключительно в одной виртуальной машине.

До версии 4.4 (KitKat) в операционной системе Android использовалась виртуальная машина процессов Dalvik. При каждом запуске процесса для его размещения создавался новый экземпляр виртуальной машины Dalvik. Затем на смену Dalvik пришла Android Runtime (ART), которая заняла роль виртуальной машины процессов с версии Android 5.0 (Lollipop).

Как правило, каждый компонент приложения в Android связывается ровно с одним процессом (хотя встречаются довольно смутные исключения). Приложение создается с собственным процессом, который становится процессом по умолчанию для всех компонентов приложения.

(Отдельные компоненты можно назначать разным процессам, но мы рекомендуем придерживаться процесса по умолчанию. Если вы думаете, что какой-то код должен выполняться в другом процессе, аналогичного результата обычно удается добиться с использованием многопоточности (multi-threading), которая программируется в Android намного проще, чем многопроцессное выполнение.)

Каждый экземпляр activity существует ровно в одном процессе и ровно в одной задаче. Впрочем, на этом все сходство и завершается. Задачи содержат только activity и часто состоят из activity разных приложений. С другой стороны, процессы содержат только выполняемый код и объекты приложения.

Процессы и задачи легко спутать, потому что эти концепции отчасти перекрываются, а для ссылок на них часто используются имена приложений. Например, при запуске CriminalIntent из NerdLauncher ОС создает процесс CriminalIntent и новую задачу, для которой `MainActivity` является базовой activity. На экране «Обзор приложений» эта задача снабжается меткой CriminalIntent.

Задача, в которой существует activity, может быть не связана с процессом, в котором она существует. Для примера возьмем приложение CriminalIntent и приложение адресной книги и рассмотрим следующий сценарий.

Откройте CriminalIntent, выберите преступление в списке (или добавьте новое) и нажмите кнопку **CHOOSESUSPECT**. При этом запускается приложение адресной книги для выбора подозреваемого. Activity списка контактов добавляется в задачу CriminalIntent. Это означает, что, когда пользователь нажимает кнопку «Назад» для перехода между разными activity, он может незаметно для себя переключаться между процессами.

При этом экземпляр activity списка контактов создается в пространстве памяти процесса приложения адресной книги и выполняется на виртуальной машине, существующей в процессе приложения адресной книги. (Состояния экземпляров activity и задачи в этом сценарии изображены на рис. 23.12.)

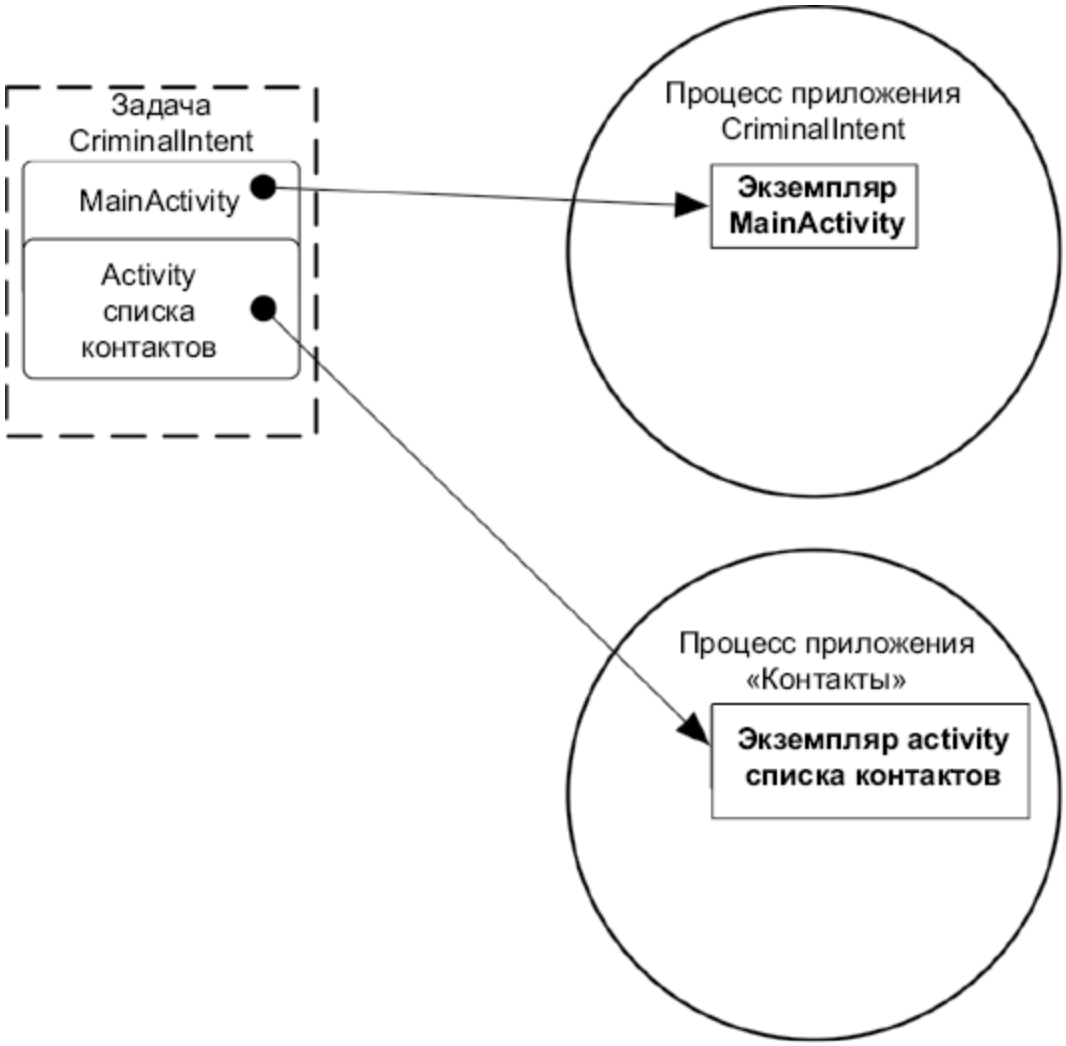


Рис. 23.12. Задача ссылается на несколько процессов

Чтобы продолжить исследование различий между задачами и процессами, оставьте CriminalIntent работать. (Проследите за тем, чтобы само приложение адресной книги не было представлено на экране «Обзор приложений»; если оно там есть, удалите задачу.) Нажмите кнопку «Главный экран». Запустите приложение адресной книги с главного экрана. Выберите контакт в списке (или добавьте новый контакт).

При этом экземпляры activity нового списка контактов и подробной информации о контакте будут созданы в процессе приложения адресной книги. Для приложения адресной книги будет создана новая задача, которая содержит ссылки на

экземпляры activity списка контактов и подробной информации (рис. 23.13).

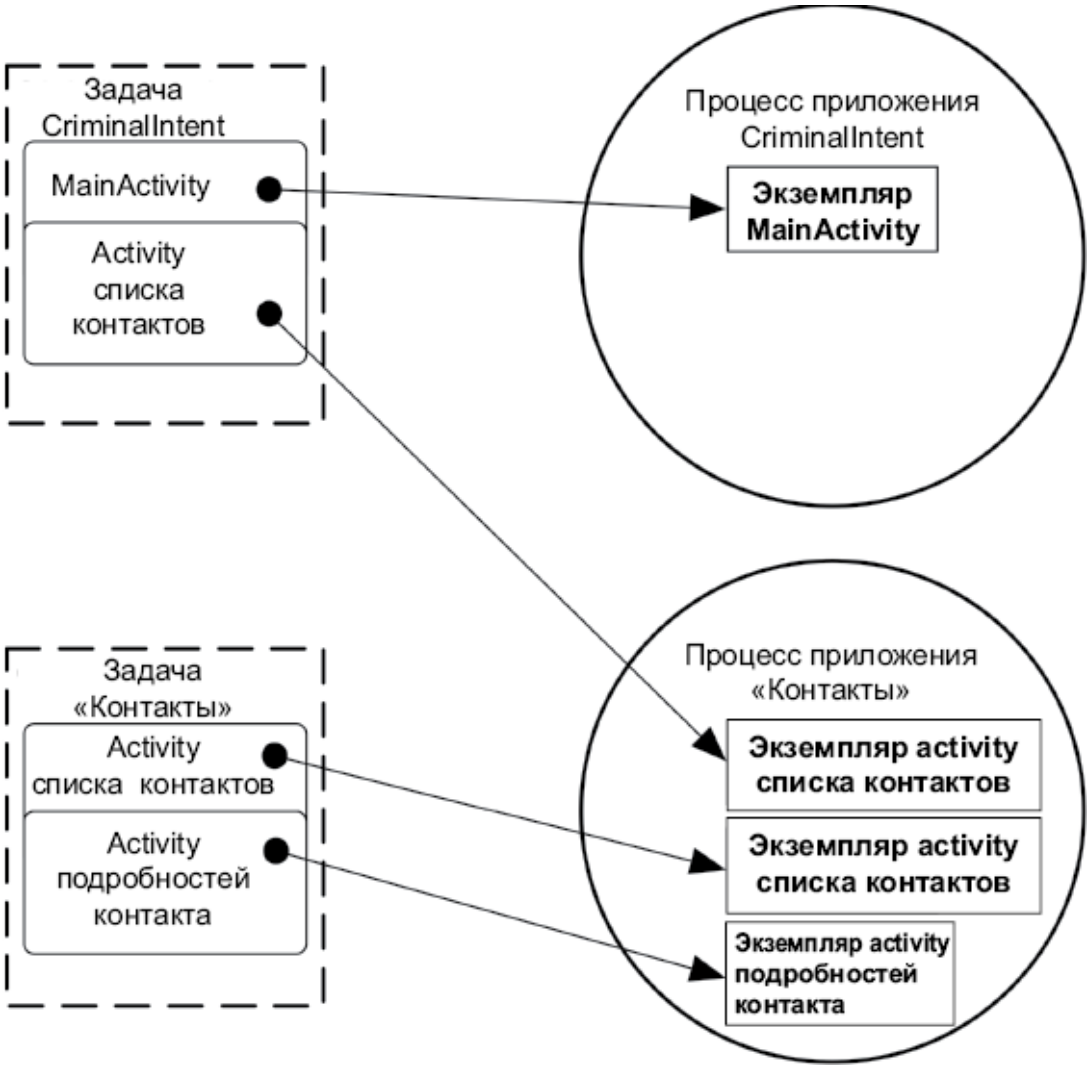


Рис. 23.13. Процесс, на который ссылаются несколько задач

В этой главе мы создавали задачи и переключались между ними. Как насчет замены стандартного экрана «Обзор приложений» Android? К сожалению, Android не предоставляет средств для решения этой задачи. Также следует знать, что приложения, рекламируемые в магазине Google Play как «уничтожители задач», в действительности являются уничтожителями процессов. Такие приложения уничтожают

конкретный процесс, что может привести к уничтожению activity, на которые ссылаются задачи других приложений.

Для любознательных: параллельные документы

Запуская приложения с экрана «Обзор приложений» и передавая данные другим приложениям, вы заметите один интересный аспект поведения в отношении CriminalIntent. При отправке отчета о преступлении из CriminalIntent activity приложения, выбранного на экране, добавляется в отдельную задачу, а не в задачу CriminalIntent (рис. 23.14).

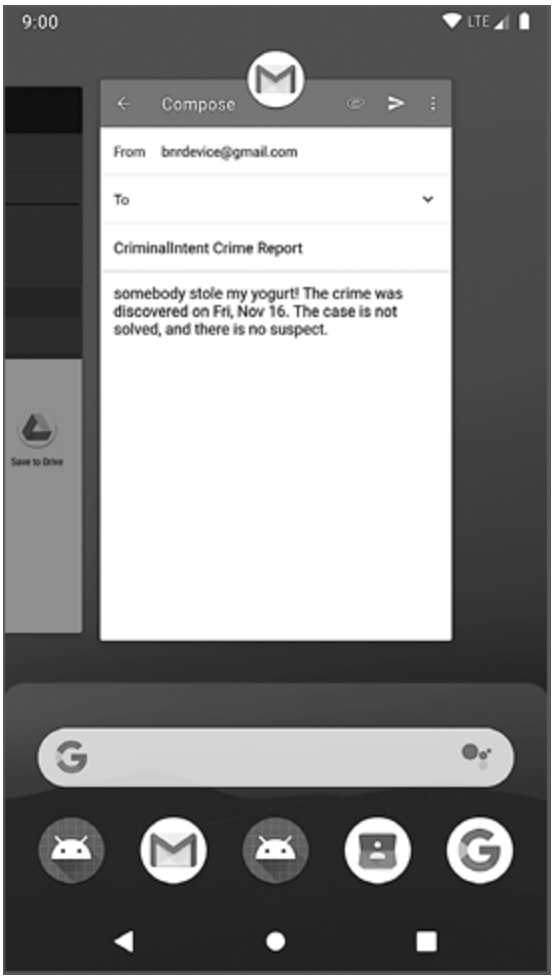


Рис. 23.14. Gmail запускается отдельной задачей

Для activity, запускаемых действиями `android.intent.action.SEND` и `action.intent.action.SEND_MULTIPLE`, создаются новые отдельные задачи.

В этом поведении используется концепция *параллельных документов* (*concurrentdocuments*). Механизм параллельных документов позволяет динамически создать для приложения любое количество задач во время выполнения. Обратите внимание, что данный функционал впервые появился в версии Android Lollipop (API уровня 21), поэтому, если вы протестируете его на более старом устройстве, то не увидите параллельные документы на экране «Обзор приложений». До версии Lollipop приложения могли использовать только заранее определенный набор задач, имена которых должны быть указаны в манифесте.

Типичный пример практического использования параллельных документов — приложение Google Drive. Вы можете открывать и редактировать сразу несколько документов, каждый из которых получает собственную задачу на экране «Обзор приложений» (рис. 23.15).

Рис. 23.15. Задачи Google Drive на экране «Обзор приложений»

Чтобы запустить несколько «документов» (задач) из приложения, либо добавьте флаг `Intent.FLAG_ACTIVITY_NEW_DOCUMENT` в интент перед вызовом `startActivity(...)`, либо присвойте activity в манифесте атрибут `documentLaunchMode`:

```
<activity
    android:name=".CrimePagerActivity"
    android:label="@string/app_name"
    android:parentActivityName=".MainActivity"
        android:documentLaunchMode="intoExistin
g" />
```

При использовании этого способа для каждого документа будет создаваться только одна задача (и при отправке интента с теми же данными, что у существующей задачи, новая задача создана не будет). Также можно включить режим принудительного создания новой задачи даже в том случае, если она уже существует для данного документа: либо добавьте перед отправкой интента флаг `Intent.FLAG_ACTIVITY_MULTIPLE_TASK` вместе с `Intent.FLAG_ACTIVITY_NEW_DOCUMENT`, либо используйте в манифесте атрибут `documentLaunchMode` со значением `always`.

Упражнение. Значки

В этой главе мы использовали функцию `ResolveInfo.loadLabel(PackageManager)` для отображения содержательных имен в лаунчере. Класс `ResolveInfo` предоставляет аналогичную функцию `loadIcon()` для получения значка, отображаемого для каждого приложения. Вам предлагается несложное упражнение: снабдить каждое приложение в NerdLauncher значком.

24. HTTP и фоновые задачи

В умах пользователей господствуют сетевые приложения. Чем заняты эти люди, которые за обедом возятся со своими телефонами вместо дружеской беседы? Они маниакально обновляют новостную ленту, отвечают на текстовые сообщения или играют в сетевые игры.

Для экспериментов с сетевыми возможностями Android мы создадим новое приложение PhotoGallery. Это клиент для веб-фотохостинга Flickr, который будет загружать и отображать самые интересные общедоступные фото, отправленные на Flickr. Рис. 24.1 дает примерное представление о внешнем виде приложения.

(Мы добавили в свою реализацию PhotoGallery фильтр, с которым отображаются только фотографии, опубликованные на Flickr «без известных ограничений авторского права». За дополнительной информацией об использовании такого контента обращайтесь по адресу **www.flickr.com/commons/usage/**. Все остальные фотографии на сайте Flickr являются собственностью человека, опубликовавшего их, и не могут повторно использоваться без разрешения владельца. Дополнительная информация об использовании независимого контента, загруженного с Flickr, доступна по адресу **www.flickr.com/creativecommons/**.)

Мы будем работать с PhotoGallery на протяжении нескольких глав. В этой главе вы узнаете, как использовать библиотеку Retrofit для выполнения веб-запросов в REST API и библиотеку Gson для десериализации ответов на эти запросы из формата JSON в объекты Kotlin. Почти все программирование веб-сервисов в наши дни основано на сетевом протоколе HTTP.

Модернизация обеспечивает легкий доступ к веб-службам HTTP и HTTP/2 из Android-приложений.

К концу главы мы научимся собирать, парсить и отображать заголовки фотографий с помощью Flickr (рис. 24.2). Поиск и отображение фотографий реализуется в главе 25.

Рис. 24.1. Готовое приложение PhotoGallery

Рис. 24.2. Приложение PhotoGallery в конце этой главы

Создание приложения PhotoGallery

Создайте новый проект Android-приложения. Как и прежде, выберите целевой форм-фактор **PhoneandTablet** и отключите добавление activity. Назовите проект **PhotoGallery**. Убедитесь,

что указано название пакета — **com.bignerdranch.android.photogallery**, язык — **Kotlin**, а минимальный уровень API — **API21:Android5.0(Lollipop)**. Установите флажок **UseAndroidXartifacts**.

После инициализации проекта щелкните мышью по приложению на панели **Project** и добавьте пустую activity (**File⇒New⇒Activity⇒EmptyActivity**) с именем `PhotoGalleryActivity`. Установите флажок, чтобы `PhotoGalleryActivity` была запускающей activity.

В `PhotoGalleryActivity` будет размещен фрагмент `PhotoGalleryFragment`, который вы вскоре создадите. Откройте файл `res/layout/activity_photo_gallery.xml` и замените автоматически сгенерированное содержимое на один `FrameLayout`. Этот макет будет служить контейнером для размещенного фрагмента. Назначьте макету идентификатор `fragmentContainer`. Когда вы закончите, содержимое файла `activity_photo_gallery.xml` должно быть таким, как показано в листинге 24.1.

Листинг 24.1. Добавление контейнера для фрагмента (res/layout/activity_photo_gallery.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:id="@+id/fragmentContainer"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".PhotoGalleryActivity"/>
```

В файле PhotoGalleryActivity.kt измените код функции onCreate(...), чтобы проверить, разместился ли фрагмент в контейнере. Если нет, создайте экземпляр PhotoGalleryFragment и добавьте его в контейнер (на ошибку пока внимания не обращаем. Она исчезнет после создания класса PhotoGalleryFragment).

Листинг 24.2. Настройка activity (PhotoGalleryActivity.kt)

```
class PhotoGalleryActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_photo_gallery)

        val isFragmentContainerEmpty = savedInstanceState == null
        if (isFragmentContainerEmpty) {
            supportFragmentManager
                .beginTransaction()
                .add(R.id.fragmentContainer, PhotoGalleryFragment.newInstance())
                .commit()
        }
    }
}
```

В приложении CriminalIntent мы проверяли размещение фрагмента в контейнере путем вызова функции

`findFragmentById(...)` на идентификаторе контейнера фрагмента. Эта проверка была необходима, так как менеджер фрагментов автоматически создает и добавляет размещаемые фрагменты обратно в activity после изменения конфигурации или после завершения процесса системой. Добавление фрагмента в контейнер необходимо только в том случае, если фрагмента там еще нет.

В приложении PhotoGalleryActivity используется другой способ определения, размещен ли фрагмент: проверка на не-`null` значения пакета `saveInstanceState` в `onCreate(...)`. Напомним, что если пакет равен `null`, то это новый запуск activity, и можно смело предположить, что ни один фрагмент еще не был автоматически восстановлен и переустановлен. Если пакет не `null`, это означает, что activity восстанавливается после уничтожения системы (например, после поворота или уничтожения процесса), и все фрагменты, которые были размещены до уничтожения, были восстановлены и добавлены обратно в соответствующие контейнеры.

Если надо выяснить, находится ли фрагмент в единственном контейнере activity, работает любой из подходов. Какой из них использовать — вопрос личных предпочтений.

Проверка `saveInstanceState` позволяет использовать информацию, которая у вас уже есть, чтобы определить, размещен ли фрагмент. Однако в этом случае предполагается, что читатель вашего кода понимает, как работает `saveInstanceState` и восстановление фрагмента после изменения конфигурации.

Проверка существования фрагмента с помощью `supportFragmentManager.findFragmentById(R.id.fragment_container)` для новичков в Android более понятна и проста для чтения. Однако если фрагмент уже существует в

контейнере, предполагается неоправданное взаимодействие с менеджером фрагментов.

Теперь надо настроить представление фрагмента. Приложение PhotoGallery будет отображать свои результаты в `RecyclerView`, используя встроенный `GridLayoutManager`, который расположит элементы в сетке. Сначала нужно добавить библиотеку `RecyclerView` в качестве зависимости, как это было сделано в главе 9. Откройте файл `build.gradle` модуля приложения и добавьте зависимость от Gradle для представления утилизатора (листинг 24.3). После внесения этих изменений синхронизируйте файл Gradle с помощью подсказки Android Studio.

Листинг 24.3. Добавление зависимости RecyclerView (app/build.gradle)

```
dependencies {
    ...
                                    implementation
'androidx.constraintlayout:constraintlayout:2.0
.0-alpha2'
                                    implementation
'androidx.recyclerview:recyclerview:1.0.0'
    ...
}
```

Далее щелкните правой кнопкой мыши по папке `res/layout` на панели **Project** и выберите команду **New⇒Layoutresourcefile** в контекстном меню. Присвойте файлу имя **fragment_photo_gallery.xml** и задайте `androidx.recyclingerview.widget.RecyclerView` в качестве корневого элемента. Когда файл будет создан,

присвойте свойству утилизатора `android:id` значение `@+id/photo_recycler_view`. Закройте открывающий тег закрывающим, так как между ними ничего не будет. Когда вы закончите, содержимое файла `res/layout/ftgment_photo_gallery.xml` должно соответствовать коду, показанному в листинге 24.4.

Листинг 24.4. Добавление утилизатора в макет фрагмента (res/layout/fragment_photo_gallery.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/photo_recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
```

Наконец, создайте класс `PhotoGalleryFragment`. Заполните вновь созданный макет и задайте переменную-член, ссылающуюся на `RecyclerView`. Установите `layoutManager` утилизатора на новый экземпляр `GridLayoutManager`. Пока что просто зададим количество столбцов равным 3 (в разделе «Упражнение: динамическая настройка количества столбцов» в конце этой главы вам будет поставлена задача адаптировать количество столбцов под ширину экрана). Когда вы закончите, код фрагмента должен оказаться следующим:

Листинг 24.5. Настройка фрагмента (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
```

```
      private lateinit var photoRecyclerView:
RecyclerView

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
                                  val    view    =
inflater.inflate(R.layout.fragment_photo_galler
y, container, false)

                               photoRecyclerView   =
view.findViewById(R.id.photo_recycler_view)
                photoRecyclerView.layoutManager  =
GridLayoutManager(context, 3)

        return view
    }

    companion object {
                            fun   newInstance()   =
PhotoGalleryFragment()
    }
}
```

Прежде чем двигаться дальше, запустите приложение PhotoGallery, чтобы убедиться, что все подключено правильно. Если все в порядке, вы увидите красивый пустой экран.

Основы работы с сетью при помощи Retrofit

Retrofit — это библиотека с открытым исходным кодом компании Square (**square.github.io/retrofit**). В качестве HTTP-клиента в ней используется библиотека OkHttp (**square.github.io/okhttp**).

Retrofit помогает построить класс шлюза HTTP. Вы пишете интерфейс с аннотированными методами экземпляров, а Retrofit создает реализацию. Реализация Retrofit управляет созданием HTTP-запроса и разбором HTTP-ответа в OkHttp.ResponseBody. Но тут есть ограничения: было бы лучше, если бы можно было работать с типами данных вашего приложения. Для этого Retrofit позволяет вам зарегистрировать преобразователь ответов, который затем используется для преобразования ваших типов данных в запрос и удаления ваших типов данных из ответа.

Добавьте зависимость от Retrofit в файл `build.gradle` вашего модуля приложения. Синхронизируйте свой файл Gradle после внесения изменений.

Листинг 24.6. Добавление зависимости Retrofit (app/build.gradle)

```
dependencies {
        implementation fileTree(dir: 'libs',
include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-
stdlib-jdk7:$kotlin_version"
    ...
                                    implementation
'com.squareup.retrofit2:retrofit:2.5.0'
}
```

Перед тем как модернизировать Flickr REST API, сначала нужно настроить Retrofit на извлечение и запись в журнал URL

веб-страницы — в частности, главной страницы Flickr. В Retrofit много всего. Начав с простого, посмотрим, как все это устроено. Позже на основе этой базовой реализации мы построим запросы Flickr и *десериализуем* ответы — то есть преобразуем линейные, сериализованные данные в несериализованные фрагменты данных. Именно эти несериализованные данные будут вашими модельными объектами.

Определение интерфейса API

Пришло время определить вызовы API, которые ваше приложение должно научиться выполнять. Сначала создайте новый пакет для вашего связанного с API кода. На панели **Project** щелкните правой кнопкой мыши по папке `com.bignerdranch.android.photogallery` и выберите команду **New⇒Package** в контекстном меню. Присвойте новому пакету имя **api**.

Далее добавьте в ваш новый пакет интерфейс Retrofit API. Это стандартный интерфейс Kotlin, который использует аннотации Retrofit для определения вызовов API. Щелкните правой кнопкой мыши по папке `api` на панели **Project**. Выберите команду **New⇒KotlinFile/Class** в контекстном меню и присвойте файлу имя **FlickrApi**. В раскрывающемся списке **Kind** выберите пункт **File**. В новом файле определите интерфейс под именем `FlickrApi` и добавьте один GET-запрос.

Листинг 24.7. Добавление интерфейса Retrofit API (api/FlickrApi.kt)

```
interface FlickrApi {

    @GET("/")
```

```
        fun fetchContents(): Call<String>
    }
```

Если при импорте `Call` у вас появляется выбор вариантов, выберите `retrofit2.Call`.

Каждая функция в интерфейсе привязывается к конкретному HTTP-запросу и должна быть аннотирована *аннотацией метода HTTP-запроса*. Аннотация метода HTTP-запроса сообщает Retrofit тип HTTP-запроса (его называют HTTP verb), с которым сопоставляется функция в вашем API интерфейса. Наиболее распространенные типы запросов: `@GET`, `@POST`, `@PUT`, `@DELETE` и `@HEAD` (полный список запросов есть в документации API на сайте **square.github.io/retrofit/2.x/retrofit**).

Аннотация `@GET("/")` в приведенном выше коде настраивает `Call`, возвращаемый функцией `fetchContents()`, на выполнение GET-запроса. «/» — это *относительный путь* — строка пути, представляющая относительный URL-адрес от базового URL-адреса конечной точки API. В большинстве аннотаций методов HTTP-запросов используется именно относительный путь. В этом случае относительный путь «/» означает, что запрос будет отправлен на базовый URL-адрес, который мы тоже скоро зададим.

По умолчанию все веб-запросы Retrofit возвращают объект `retrofit2.Call`. Объект `Call` представляет собой один веб-запрос, который вы можете выполнить. При выполнении вызова генерируется один соответствующий веб-отклик. (Вы также можете настроить Retrofit на возврат объекта RxJava `Observables`, но это выходит за рамки данной книги.)

Тип, который вы используете в качестве общего параметра типа `Call`, определяет тип данных, на которые Retrofit раскладывает HTTP-ответ. По умолчанию Retrofit десериализует ответ на `OkHttp.ResponseBody`. Указание типа

`Call<String>` инструктирует Retrofit, чтобы ответ десериализовался в объект `String`.

Сборка объекта Retrofit и создание экземпляра API

Экземпляр Retrofit отвечает за реализацию и создание экземпляров вашего интерфейса API. Для выполнения веб-запросов через интерфейс API, который вы определили, вам необходимо, чтобы Retrofit реализовывал и создавал экземпляр интерфейса `FlickrApi`.

Сначала создайте и настройте экземпляр Retrofit. Откройте файл `PhotoGalleryFragment.kt`. В коде функции `onCreate(...)` соберите объект Retrofit и используйте его для создания реализации вашего интерфейса `FlickrApi`.

Листинг 24.8. Использование объекта Retrofit для создания экземпляра API (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {

      private lateinit var photoRecyclerView:
RecyclerView

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

                  val retrofit: Retrofit =
Retrofit.Builder()
            .baseUrl("https://www.flickr.com/")
            .build()
```

```
                val flickrApi: FlickrApi =
retrofit.create(FlickrApi::class.java)
    }
    ...
}
```

`Retrofit.Builder()` — это интерфейс, выполняющий настройку и сборку вашего экземпляра Retrofit. Базовый URL для вашей конечной точки задается с помощью функции `baseUrl(...)`. Здесь надо указать главную страницу Flickr: `"https://www.flickr.com/"`. Убедитесь, что в URL-адресе указан соответствующий протокол (здесь: `https://`). Также всегда включайте трейлинг `/`, чтобы убедиться, что Retrofit правильно добавляет относительные пути, задаваемые в интерфейсе API, к базовому URL.

Вызов функции `build()` возвращает экземпляр Retrofit, у которого появляются настройки, заданные с помощью объекта `builder`. Получив объект Retrofit, вы используете его для создания экземпляра вашего интерфейса API. Обратите внимание, что Retrofit не генерирует код во время компиляции, а делает всю работу во время выполнения. При вызове функции `retrofit.create(...)` Retrofit использует информацию в указанном вами интерфейсе API наряду с информацией, указанной вами при сборке экземпляра Retrofit, для создания экземпляров анонимного класса, реализующего интерфейс «на лету».

Добавление строкового конвертера

По умолчанию Retrofit десериализует веб-ответы в объекты `okhttp3.ResponseBody`. Однако для протоколирования содержимого веб-страницы гораздо проще работать с традиционными (объектами) `String`. Чтобы заставить Retrofit

десериализовать ответ в строки, при сборке объекта Retrofit нужно указать *конвертер*.

Конвертер знает, как декодировать объект `ResponseBody` в другой тип объекта. Вы можете создать собственный конвертер, хотя это и необязательно. К счастью для вас, команда Square создала конвертер с открытым исходным кодом под названием «скалярный конвертер», который может преобразовывать ответ в строку. Мы будем использовать его для десериализации ответов Flickr в строковые объекты.

Чтобы использовать скалярный конвертер, сначала добавьте зависимость в файл `build.gradle` вашего модуля приложения. Не забудьте синхронизировать файл после добавления зависимости.

Листинг 24.9. Добавление зависимости от скалярного конвертера (app/build.gradle)

```
dependencies {
    ...
                                    implementation
'com.squareup.retrofit2:retrofit:2.5.0'
                                    implementation
'com.squareup.retrofit2:converter-
scalars:2.5.0'
}
```

Теперь создайте экземпляр скалярного конвертера и добавьте его в свой объект Retrofit.

Листинг 24.10. Добавление конвертера в объект Retrofit (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
```

```
      private lateinit var photoRecyclerView:
RecyclerView

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

                  val retrofit: Retrofit =
Retrofit.Builder()
            .baseUrl("https://www.flickr.com/")
            .addConverterFactory(ScalarsConvert
erFactory.create())
            .build()

                 val flickrApi: FlickrApi =
retrofit.create(FlickrApi::class.java)
    }
    ...
}
```

Функция `addConverterFactory(...)` в объекте `Retrofit.Builder` ожидает экземпляр `Converter.Factory`.

Фабрика конвертеров умеет создавать и возвращать экземпляры конкретного конвертера. Функция `ScalarsConverterFactory.create()` возвращает экземпляр скалярного конвертера (`retrofit2.converter.scalars.ScalarsConverterFacto ry`), который, в свою очередь, возвращает экземпляры скалярного конвертера, когда Retrofit это нужно.

Поскольку в качестве возвращаемого типа для `FlickrApi.fetchContents()` был указан `Call<String>`,

фабрика создает экземпляр скалярного конвертера (`retrofit2.converter.scalars.StringResponseBodyConverter`). В свою очередь, ваш объект Retrofit будет использовать преобразователь строк для преобразования объекта `ResponseBody` в `String` перед возвращением результата вызова.

Команда Square создала и другие удобные конвертеры с открытым кодом для Retrofit. Позднее в этой главе вы будете использовать конвертер Gson. Ознакомиться с конвертерами и с информацией о создании собственного пользовательского конвертера вы можете на сайте **square.github.io/retrofit**.

Выполнение веб-запроса

До этого момента вы писали код для настройки вашего сетевого запроса. Настал момент, которого вы долго ждали: выполнение веб-запросов и протоколирование результата. Сначала вызываем функцию `fetchContents()` для генерации объекта `retrofit2.Call`, представляющего собой исполняемый веб-запрос.

Листинг 24.11. Получение объекта Call, выполняющего запрос (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {

      private lateinit var photoRecyclerView:
RecyclerView

     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
```

```
                val flickrApi: FlickrApi =
retrofit.create(FlickrApi::class.java)

        val flickrHomePageRequest: Call<String>
= flickrApi.fetchContents()
    }
    ...
}
```

Обратите внимание, что вызов функции `fetchContents()` на экземпляре `FlickrApi` не приводит к созданию сети. Вместо этого функция `fetchContents()` возвращает объект `Call<String>`, представляющий собой веб-запрос. Вы можете выполнить `Call` в любой момент в будущем. Retrofit определяет подробности объекта вызова по данным интерфейса API (`FlickrApi`) и созданного вами объекта Retrofit.

Для выполнения веб-запроса, содержащегося в объекте `Call`, необходимо вызвать функцию `enqueue(...)` в `onCreate(savedInstanceState:Bundle?)` и передать экземпляр `retrofit2.Callback`. Пока вы этим занимаетесь, добавьте константу `TAG`.

Листинг 24.12. Выполнение запроса асинхронно (PhotoGalleryFragment.kt)

```
private const val TAG = "PhotoGalleryFragment"

class PhotoGalleryFragment : Fragment() {

      private lateinit var photoRecyclerView:
RecyclerView
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        ...
                val flickrHomePageRequest :
Call<String> = bnrInterface.fetchContents()

        flickrHomePageRequest.enqueue(object :
Callback<String> {
                override fun onFailure(call:
Call<String>, t: Throwable) {
                    Log.e(TAG, "Failed to fetch
photos", t)
            }

            override fun onResponse(
                call: Call<String>,
                response: Response<String>
            ) {
                Log.d(TAG, "Response received:
${response.body()}")
            }
        })
    }
}
```

Retrofit позволяет легко соблюдать два наиболее важных правила потоков в Android:

1. Выполнять долгосрочные операции только в фоновом потоке, а не в основном.

2. Обновлять пользовательский интерфейс только из главного потока, а не из фонового.

Функция `Call.enqueue(...)` выполняет веб-запрос, находящийся в объекте `Call`. Самое главное, что запрос выполняется в фоновом потоке. Retrofit управляет фоновым потоком самостоятельно, и вам не нужно об этом думать.

В потоке ведется очередь задач, которые необходимо выполнить. При вызове функции `Call.enqueue(...)` Retrofit добавляет запрос в свою очередь задач. Вы можете вызывать несколько запросов, и Retrofit будет обрабатывать их один за другим до тех пор, пока очередь не станет пустой (подробнее о создании и управлении фоновыми потоками поговорим в главе 25).

Объект `Callback`, который вы передаете в `enqueue(...)`, позволяет определить, что вы хотите сделать после того, как будет получен ответ на запрос. Когда запрос, выполняемый в фоновом потоке, будет завершен, Retrofit вызовет одну из функций обратного вызова, которую вы предоставили в основном потоке (UI): если ответ от сервера получен, вызывается функция `Callback.onResponse(...)`, а если нет, то `onFailure(...)`.

Передача `Response` Retrofit в `onResponse()` содержит в своем теле содержимое результата. Тип результата будет соответствовать типу возвращаемого объекта, который вы указали в соответствующей функции в интерфейсе API. В данном случае `fetchContents()` возвращает `Call<String>`, поэтому `response.body()` возвращает строку.

Объект `Call`, переданный функциям `onResponse()` и `onFailure()`, — это исходный объект вызова, используемый для инициирования запроса.

Вы можете организовать синхронное выполнение запросов с помощью вызова `Call.execute()`. Просто убедитесь, что выполнение происходит в фоновом, а не в основном потоке пользовательского интерфейса. Как вы узнали из главы 11, Android запрещает все сетевые взаимодействия в основном потоке. Если вы попытаетесь это сделать, Android выдаст исключение `NetworkOnMainThreadException`.

Запрос разрешения на работу с сетью

Для создания сети необходима еще одна вещь: нужно спросить разрешение. Пользователи не хотят, чтобы вы тайно фотографировали их, и точно так же они не хотят, чтобы вы тайно скачивали информацию и ASCII-картинки животных.

Чтобы запросить разрешение на использование в сети, добавьте следующее разрешение в файл `manifests/AndroidManifest.xml`.

Листинг 24.13. Запрос на использование сети (manifests/AndroidManifest.xml)

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.bignerdranch.android.photogallery" >

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

    <application>
        ...
```

```
    </application>

</manifest>
```

Android считает разрешение `INTERNET` «обычным», так как доступ к сети требуется многим приложениям. Поэтому для реализации разрешения достаточно объявить его в своем манифесте. Более опасные разрешения (например, разрешение на местоположение устройства) дополнительно требуют выполнения запроса.

Запустите свой код, и вы увидите главную страницу Flickr на панели **Logcat**, как показано на рис. 24.3. (Поиск записей на панели **Logcat** — это непросто. Проще искать что-то конкретное. В этом случае введите **PhotoGalleryFragment** в поле поиска на панели **Logcat**, как показано на рисунке.)

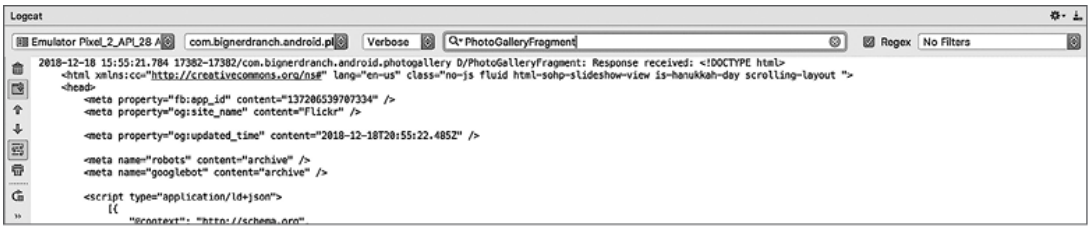


Рис. 24.3. HTML-фрагмент Flickr.com в журнале на панели Logcat

Переход к шаблону репозитория

В данный момент код для работы с сетью встроен в ваш фрагмент. Перед тем как двигаться дальше, переместите код конфигурации Retrofit и прямой доступ к API в новый класс.

Создайте новый файл Kotlin под названием `FlickrFetchr.kt`. Добавьте свойство для хранения экземпляра `FlickrApi`. Уберите код конфигурации Retrofit и создания экземпляра API интерфейса из `PhotoGalleryFragment` и вставьте его в блок `init` в новом

классе (это две строки, которые начинаются с `valretrofit:Retrofit=...` и `flickrApi=...` в листинге 24.14). Разделите объявление и присвоение `flickrApi` на две отдельные строки, чтобы объявить `flickrApi` как приватное свойство `FlickrFetchr`. Это делается для того, чтобы вы могли получить доступ к нему в другом месте класса (вне блока `init`), но не вне класса.

Когда вы закончите, код `FlickrFetchr` должен выглядеть так, как показано в листинге 24.14.

Листинг 24.14. Создание FlickrFetchr (FlickrFetchr.kt)

```
private const val TAG = "FlickrFetchr"

class FlickrFetchr {

    private val flickrApi: FlickrApi

    init {
                    val retrofit: Retrofit =
Retrofit.Builder()
            .baseUrl("https://www.flickr.com/")
            .addConverterFactory(ScalarsConvert
erFactory.create())
            .build()

                                    flickrApi    =
retrofit.create(FlickrApi::class.java)
    }
}
```

Если вы этого еще не сделали, уберите код конфигурации Retrofit из PhotoGalleryFragment (листинг 24.15). Это вызовет ошибку, но мы исправим ее, когда доделаем FlickrFetchr.

Листинг 24.15. Удаление настроек Retrofit из фрагмента (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {

      private lateinit var photoRecyclerView:
RecyclerView

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

                    ~~val retrofit: Retrofit =~~
~~Retrofit.Builder()~~
                ~~.baseUrl("https://www.flickr.co~~
~~m/")~~
                ~~.addConverterFactory(ScalarsCon~~
~~verterFactory.create())~~
                ~~.build()~~

                    ~~val flickrApi: FlickrApi =~~
~~retrofit.create(FlickrApi::class.java)~~

                  val flickrHomePageRequest :
Call<String> = flickrApi.fetchContents()

        ...
```

```
    }
    ...
}
```

Затем добавьте функцию `fetchContents()` во `FlickrFetchr`, чтобы обернуть функцию API Retrofit, которую вы определили для загрузки главной страницы Flickr. (Вы можете скопировать большую часть кода из `PhotoGalleryFragment`, но убедитесь, что вставленный результат соответствует коду в листинге 24.16.)

Листинг 24.16. Добавление функции fetchContents() во FlickrFetchr (FlickrFetchr.kt)

```
private const val TAG = "FlickrFetchr"

class FlickrFetchr {

    private val flickrApi: FlickrApi

    init {
        ...
    }

    fun fetchContents(): LiveData<String> {
                        val responseLiveData:
MutableLiveData<String> = MutableLiveData()
            val flickrRequest: Call<String> =
flickrApi.fetchContents()

                flickrRequest.enqueue(object :
Callback<String> {
```

```
                override fun onFailure(call:
Call<String>, t: Throwable) {
                    Log.e(TAG, "Failed to fetch
photos", t)
            }

            override fun onResponse(
                call: Call<String>,
                response: Response<String>
            ) {
                Log.d(TAG, "Response received")
                    responseLiveData.value =
response.body()
            }
        })

        return responseLiveData
    }
}
```

В функции `fetchContents()` вы приписываете значение `responseLiveData` пустому объекту `MutableLiveData<String>`. Затем вы ставите в очередь веб-запрос для получения страницы Flickr и возвращаете `responseLiveData` (до завершения запроса). После успешного завершения результат становится публичным путем установки значения `responseLiveData.value`. Таким образом, другие компоненты, такие как `PhotoGalleryFragment`, могут наблюдать объект `LiveData`, возвращенный из `fetchContents()`, чтобы в конечном итоге получить результаты веб-запроса.

Обратите внимание, что возвращаемый тип `fetchContents()` является неизменяемым `LiveData<String>`. Вам следует избегать выставления на публику объектов «живых» данных, если это возможно, чтобы другие компоненты не могли изменять содержимое «живых» данных. Данные через `LiveData` должны идти в одном направлении.

`FlickrFetchr` оборачивает большую часть сетевого кода в PhotoGallery (сейчас обертка простая, но мы усложним ее в следующих главах). Функция `fetchContents()` ставит в очередь сетевой запрос и обертывает результат в `LiveData`. Теперь другие компоненты вашего приложения, такие как `PhotoGalleryFragment` (или `ViewModel` или activity и т.д.), могут создавать экземпляр `FlickrFetchr` и запрашивать данные фотографии без необходимости знать что-то о Retrofit или об источнике, из которого исходят данные.

Измените код `PhotoGalleryFragment`, чтобы использовать `FlickrFetchr` и увидеть магию в действии (листинг 24.17).

Листинг 24.17. Использование FlickrFetchr в PhotoGalleryFragment (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {

      private lateinit var photoRecyclerView:
RecyclerView

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
```

~~val flickrHomePageRequest : Call<String> = flickrApi.fetchContents()~~

~~flickrHomePageRequest.enqueue(object : Callback<String> {~~
~~override fun onFailure(call: Call<String>, t: Throwable) {~~
~~Log.e(TAG, "Failed to fetch photos", t)~~
~~}~~

~~override fun onResponse(~~
~~call: Call<String>,~~
~~response: Response<String>~~
~~) {~~
~~Log.d(TAG, "Response received: ${response.body()}")~~
~~}~~
~~})~~

```
        val flickrLiveData: LiveData<String> =
FlickrFetchr().fetchContents()
        flickrLiveData.observe(
            this,
            Observer { responseString ->
                Log.d(TAG, "Response received:
$responseString")
            })
    }
    ...
}
```

Теперь ваше приложение стало похожим на шаблон репозитория, рекомендованный Google в документации (**developer.android.com/jetpack/docs/guide**). `FlickrFetchr` — это своего рода простой репозиторий. Класс репозитория инкапсулирует логику доступа к данным из одного источника или набора источников. Он определяет, как получать и хранить определенный набор данных, будь то в локальной базе данных или на удаленном сервере. Ваш код пользовательского интерфейса будет запрашивать все данные из репозитория, потому что ему неважно, как данные хранятся или извлекаются на самом деле. Это детали реализации самого репозитория.

Сейчас все данные вашего приложения поступают непосредственно с веб-сервера Flickr. Однако позднее мы научимся кэшировать эти данные в локальной базе данных. В этом случае репозиторий будет управлять получением данных из нужного места. Другие компоненты вашего приложения могут использовать репозиторий для получения данных без необходимости знать, откуда поступают данные.

Запустите приложение и убедитесь, что оно все еще работает правильно. Вы должны увидеть в Logcat содержимое главной страницы Flickr, как показано на рис. 24.3.

Получение JSON-данных от Flickr

JSON расшифровывается как JavaScript Object Notation (нотация объектов JavaScript). Это популярный формат данных, особенно для веб-сервисов. Более подробно о формате JSON можно почитать на сайте **json.org**.

На сайте Flickr доступен прекрасный JSON API. Все подробности описаны в документации на сайте **flickr.com/services/api**. Откройте ее в веб-браузере и найдите список форматов запроса (Request Formats). Вы будете

использовать самый простой формат — REST. Рабочая станция REST API доступна по адресу **api.flickr.com/services/rest**, и с помощью нее вы сможете вызывать функции Flickr.

Вернувшись на главную страницу документации API, найдите список методов API. Доберитесь до раздела **interestingness** и выберите пункт **flickr.interestingness.getList**. В документации говорится, что этот метод «возвращает список интересных фотографий за последний день или указанную пользователем дату». Именно это нам и нужно.

Единственный требуемый параметр для метода `getList` — это ключ API. Чтобы получить ключ API, вернитесь на сайт **flickr.com/services/api** и перейдите по ссылке на **APIKeys**. Для входа в систему вам понадобится Yahoo ID. После входа в систему запросите новый некоммерческий API-ключ. Эта процедура займет всего минуту. Ваш ключ API будет иметь вид `4f721bgafa75bf6d2cb9af54f937bb70`. (Вам не нужен «секрет», который используется только тогда, когда приложение получает доступ к специфической для пользователя информации или изображениям.)

Как только у вас есть ключ, вам останется всего лишь сделать запрос на веб-сервис Flickr. URL-адрес GET-запроса будет выглядеть примерно так:

```
https://api.flickr.com/services/rest/?
method=flickr.interestingness.getList
  &api_key=вашApiКлюч&format=json&nojsoncallbac
k=1&extras=url_s
```

Flickr по умолчанию дает ответ в формате XML. Чтобы получить ответ в формате JSON, необходимо указать значения как для формата, так и для параметров `nojsoncallback`. Установка значения `nojsoncallback` равным `1` инструктирует

Flickr убрать из ответа круглые скобки. Это позволяет вашему коду Kotlin легче разобрать ответ.

Указание параметра `extras` со значением `url_s` инструктирует Flickr добавить URL-адрес мини-версии изображения, если таковая есть.

Скопируйте URL-адрес примера в ваш браузер, заменив значение *вашApiКлюч* на ваш реальный ключ API. Вы увидите нечто похожее на рис. 24.4.

Пора научить код запрашивать у Flickr REST API недавние интересные фото, а не содержимое главной страницы Flickr. Сначала добавьте функцию в ваш интерфейс `FlickrApi` API. Замените *вашApiКлюч* на ваш ключ API. Пока что параметры запроса URL будут жестко закодированы в программе. (Позже вы обобщите эти параметры запроса и будете добавлять их программно.)

Рис. 24.4. Пример вывода в формате JSON

Листинг 24.18. Определение запроса «получить недавние интересные фотографии» (api/FlickrApi.kt)

```
interface FlickrApi {

    @GET("/")
    fun fetchContents() : Call<String>
```

```
    @GET(
                                    "services/rest/?
method=flickr.interestingness.getList" +
"&api_key=вашApiКлюч" +
                "&format=json" +
                "&nojsoncallback=1" +
                "&extras=url_s"
    )
    fun fetchPhotos(): Call<String>
}
```

Обратите внимание, что вы добавили значения параметров `method`, `api_key`, `format`, `nojsoncallback` и `extras`.

Теперь нужно изменить код конфигурации экземпляра Retrofit во `FlickrFetchr`. Измените базовый URL-адрес с главной страницы Flickr на конечную точку базового API. Переименуйте функцию `fetchContents()` в `fetchPhotos()` и вызовите новую функцию `fetchPhotos()` на интерфейсе API.

Листинг 24.19. Обновление базового URL (FlickrFetchr.kt)

```
class FlickrFetchr {

    private val flickrApi: FlickrApi

    init {
                    val retrofit: Retrofit =
Retrofit.Builder()
            .baseUrl("https://   www.api.flickr
.com/")
```

```
            .addConverterFactory(ScalarsConvert
erFactory.create())
            .build()

                                    flickrApi    =
retrofit.create(FlickrApi::class.java)
    }

          fun     fetchContents()fetchPhotos():
LiveData<String> {
                          val   responseLiveData:
MutableLiveData<String> = MutableLiveData()
             val flickrRequest: Call<String> =
flickrApi.   fetchContents()fetchPhotos()
         ...
    }
}
```

Обратите внимание, что основной URL-адрес, который мы установили, — **api.flickr.com**, но конечные точки находятся по адресу **api.flickr.com/services/rest**. Это связано с тем, что вы указали службы и остальную часть пути в аннотации `@GET` во `FlickrApi`. Путь и другая информация, которую вы включили в аннотацию `@GET`, будет добавлена в URL-адрес Retrofit до выдачи веб-запроса.

Теперь нужно научить `PhotoGalleryFragment` выполнять веб-запрос, чтобы он получал интересные фотографии вместо содержимого главной страницы Flickr. Замените вызов `fetchContents()` вызовом новой функции `fetchPhotos()`. Пока что, как и раньше, выполним сериализацию ответа в строку.

Листинг 24.20. Выполнение запроса «получить недавние интересные фотографии» (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {

      private lateinit var photoRecyclerView:
RecyclerView

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

         val flickrLiveData: LiveData<String> =
FlickrFetchr().fetch   ~~Contents~~Photos()
        ...
    }
    ...
}
```

Эти мелкие изменения существующего кода делают ваше приложение готовым к извлечению и записи данных Flickr в журнал. Запустите приложение PhotoGallery, и вы увидите на панели **Logcat** JSON-данные с сайта Flickr, как показано на рис. 24.5. (Это поможет найти `PhotoGalleryFragment` на панели **Logcat**.)

(Панель **Logcat** бывает привередлива. Не паникуйте, если увидите не то же самое, что у нас. Иногда подключение к эмулятору выполняется некорректно, и сообщения журналов не выводятся. Обычно со временем все проясняется, но иногда приходится перезапускать приложение или даже эмулятор.)

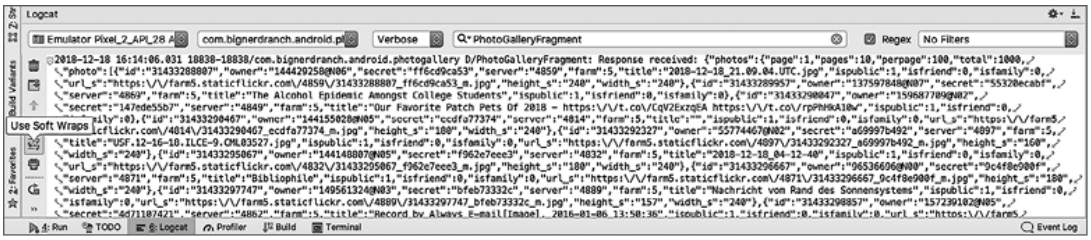


Рис. 24.5. JSON-данные с сайта Flickr на панели Logcat

На момент написания книги панель **Logcat** в Android Studio не переносит по словам вывод автоматически. Прокрутите вправо, чтобы увидеть длинную строку ответа в формате JSON, либо выровняйте строки содержимого `Logcat`, нажав **кнопкуUseSoftWraps**, показанную на рис. 24.5.

Теперь, когда у вас есть такие JSON-данные с сайта Flickr, что с ними делать? Ответ: то же, что и со всеми данными — помещать их в один или несколько объектов модели. Класс модели, который вы собираетесь создать для PhotoGallery, называется `GalleryItem`. В элементе галереи хранится метаинформация одной фотографии: название, идентификатор и URL-адрес для загрузки изображения.

Создайте класс данных `GalleryItem` и добавьте следующий код:

Листинг 24.21. Создание класса объекта модели (GalleryItem.kt)

```
data class GalleryItem(
    var title: String = "",
    var id: String = "",
    var url: String = ""
)
```

Теперь, когда вы определили объект модели, пришло время создать и заполнить экземпляры этого объекта данными из JSON-вывода, полученного с Flickr.

Десериализация JSON-текста в объекты модели

Ответ в формате JSON, отображаемый в окне браузера и на панели **Logcat**, очень трудно считывается. Если вы *красиво вывели* ответ (с пробелами), он будет выглядеть как текст слева на рис. 24.6.

JSON-объект представляет собой набор пар «имя — значение», заключенных между фигурными скобками, `{ }`. JSON-массив — это разделенный запятыми список JSON-объектов, заключенный в квадратные скобки, `[ ]`. Объекты могут быть вложены друг в друга, в результате чего образуется иерархия.

У Android имеется стандартный пакет `org.json`, в котором есть классы, предоставляющие доступ к созданию и разбору JSON-текста (такие как `JSONObject` и `JSONArray`). Тем не менее многие умные люди создали библиотеки для упрощения процесса преобразования JSON-текста в объекты Java и обратно.

Одна из таких библиотек — Gson (**github.com/google/gson**). Она автоматически сопоставляет JSON-данные с объектами Kotlin. Это означает, что вам не нужно самостоятельно писать код для парсинга. Достаточно лишь определить классы Kotlin, которые сопоставляют JSON-данные с иерархией объектов, а Gson сделает все остальное.

Команда Square создала конвертер Gson для Retrofit, который позволяет легко подключить Gson к вашей реализации Retrofit. Сначала добавьте зависимости библиотек Gson и Retrofit Gson конвертера в Gradle вашего модуля приложения. Как всегда, не забудьте синхронизировать файл, когда вы закончите.

Листинг 24.22. Добавление зависимостей Gson (app/build.gradle)

```
dependencies {
    ...
                                    implementation
'com.squareup.retrofit2:retrofit:2.5.0'
                                    implementation
'com.squareup.retrofit2:converter-
scalars:2.5.0'
                                    implementation
'com.google.code.gson:gson:2.8.5'
                                    implementation
'com.squareup.retrofit2:converter-gson:2.4.0'
}
```

Затем создадим объекты модели, которые сопоставляются с JSON-данными в ответе с Flickr. У вас уже есть `GalleryItem`, который практически напрямую сопоставляется с отдельным объектом в JSON-массиве — `"photo"`. По умолчанию Gson сопоставляет имена JSON-объектов с именами свойств. Если имена ваших свойств совпадают с именами JSON-объектов, вы можете оставить их как есть.

Однако имена свойств не всегда совпадают с именами JSON-объектов. Возьмите свойство `GalleryItem.url` и поле данных `"url_s"`. Имя `GalleryItem.url` более значимо в контексте вашей кодовой базы, поэтому лучше оставить его. В этом случае вы можете добавить в свойство `@SerializedNameannotation`, чтобы сообщить Gson, к какому JSON-полю относится свойство.

Измените код `GalleryItem`, как показано ниже.

Листинг 24.23. Переопределение сопоставления имен-свойств по умолчанию (GalleryItem.kt)

```
data class GalleryItem(
    var title: String = "",
    var id: String = "",
     @SerializedName("url_s") var url: String =
""
)
```

Теперь создадим класс `PhotoResponse` для сопоставления с объектом `"photos"` JSON-данных. Поместите новый класс в пакет `api`, так как этот класс является побочным эффектом вашей реализации десериализации Flickr API, а не объектом модели, с которым работает остальная часть вашего приложения.

Добавьте свойство `galleryItems` для хранения списка галерейных объектов и примечаний к нему с помощью `@SerializedName("photo")`. Gson автоматически создаст список и заполнит его объектами элементов галереи на основе JSON-массива `"photo"`. (На данный момент единственные данные, которые нам нужны в этом объекте, — это массив данных фотографий в JSON-объекте `"photo"`. Позже в этой главе нам потребуется захватить данные вывода страниц, если вы соберетесь выполнить упражнение «Навигация по страницам» в конце главы).

Листинг 24.24. Добавление PhotoResponse (PhotoResponse.kt)

```
class PhotoResponse {
    @SerializedName("photo")
                lateinit var galleryItems:
List<GalleryItem>
```

```
}
```

Наконец, добавим в пакет `api` класс `FlickrResponse`. Этот класс будет сопоставлен с крайним объектом в JSON-данных (тот, который находится в верхней части иерархии JSON-объектов с соответствующим обозначением `{}`). Добавьте свойство для сопоставления с полем `"photo"`.

Листинг 24.25. Добавление FlickrResponse (FlickrResponse.kt)

```
class FlickrResponse {
    lateinit var photos: PhotoResponse
}
```

Диаграмма на рис. 24.6 показывает, как созданные вами объекты сопоставляются с JSON-данными.

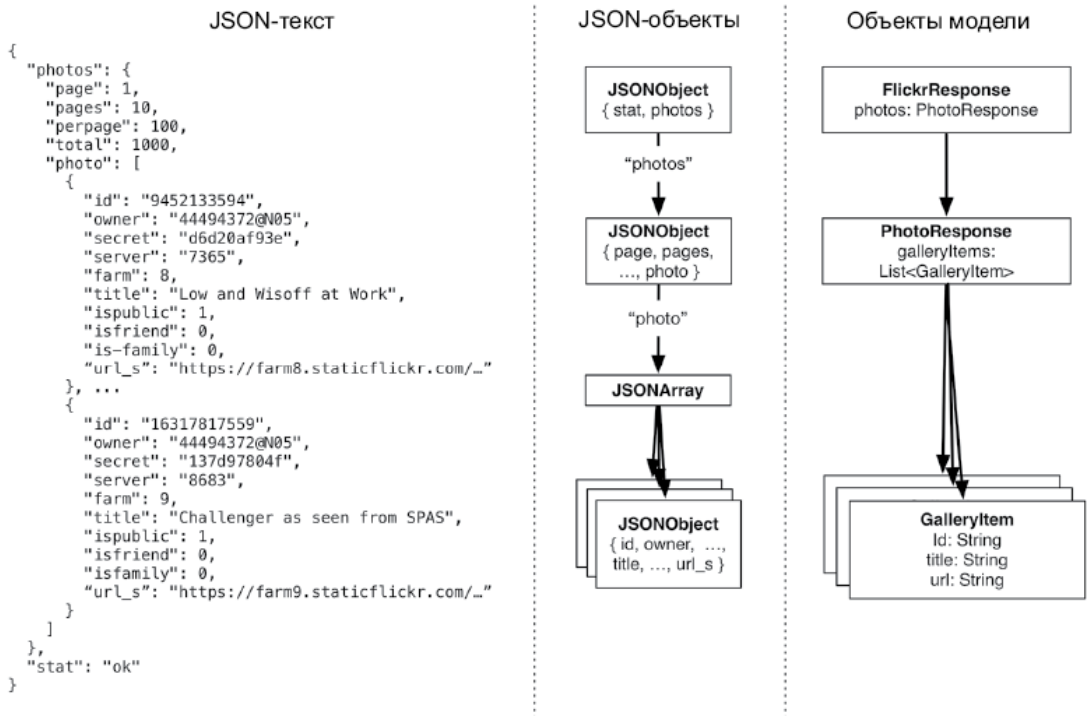


Рис. 24.6. Текст в формате JSON, иерархия JSON и соответствующие объекты модели

Теперь пришло время магии: настроим Retrofit на использование Gson для десериализации ваших данных в объекты модели, которые вы только что определили. Сначала изменим возвращаемый тип, указанный в интерфейсе API Retrofit, соответствующий объекту модели, который вы определили для сопоставления с крайним JSON-объектом. Теперь Gson будет использовать `FlickrResponse` для десериализации JSON-данных в ответе.

Листинг 24.26. Изменение возвращаемого типа fetchPhoto() (api/FlickrApi.kt)

```
interface FlickrApi {

    @GET(...)
                        fun      fetchPhotos():
Call<   String>FlickrResponse>
}
```

Теперь изменим код `FlickrFetchr`. Замените фабрику скалярного конвертера на фабрику конвертера Gson. Функция `fetchPhotos()` теперь должна возвращать объект `LiveData`, обертывающий список элементов галереи. Измените спецификаторы типа `LiveData` и `MutableLiveData` со `String` на `List<GalleryItem>`. Измените спецификаторы типа `Call` и `Callback` со `String` на `FlickrResponse`. Наконец, измените код функции `onResponse(...)`, чтобы выделить список элементов галереи из ответа и обновить `LiveData`.

Листинг 24.27. Изменение кода FlickrFetchr для Gson (FlickrFetchr.kt)

```
class FlickrFetchr {

    private val flickrApi: FlickrApi

    init {
                    val retrofit: Retrofit =
Retrofit.Builder()
            .baseUrl("https://api.flickr.com/")
             .addConverterFactory( ~~ScalarsConverterFactory~~GsonConverterFactory.create())
            .build()

                                    flickrApi =
retrofit.create(FlickrApi::class.java)
    }

                          fun fetchPhotos():
LiveData< ~~String~~List<GalleryItem>> {
                          ~~val responseLiveData:
MutableLiveData<String> = MutableLiveData()~~
                          val responseLiveData:
MutableLiveData<List<GalleryItem>> =
MutableLiveData()
                             val flickrRequest:
Call< ~~String~~FlickrResponse> =
flickrApi.fetchPhotos()

                flickrRequest.enqueue(object :
Callback< ~~String~~FlickrResponse> {
```

```
                override fun onFailure(call:
Call<   StringFlickrResponse>, t: Throwable) {
                    Log.e(TAG, "Failed to fetch
photos", t)
            }

            override fun onResponse(
                                            call:
Call<   StringFlickrResponse>,
                                        response:
Response<   StringFlickrResponse>
            ) {
                Log.d(TAG, "Response received")
                    responseLiveData.value =
response.body()
                        val flickrResponse:
FlickrResponse? = response.body()
                         val photoResponse:
PhotoResponse? = flickrResponse?.photos
                          var galleryItems:
List<GalleryItem> = photoResponse?.galleryItems
                    ?: mutableListOf()
                            galleryItems =
galleryItems.filterNot {
                    it.url.isBlank()
                }
                       responseLiveData.value =
galleryItems
            }
        })

        return responseLiveData
```

```
    }
}
```

Обратите внимание, что Flickr не всегда возвращает для изображения правильный `url_s`. Код выше отфильтровывает элементы галереи с пустыми значениями URL-адреса, используя `filterNot{...}`.

Наконец, измените код спецификатора типа `LiveData` в `PhotoGalleryFragment`.

Листинг 24.28. Новые спецификаторы типов в PhotoGalleryFragment (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {

      private lateinit var photoRecyclerView:
RecyclerView

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

                              ~~val flickrLiveData:~~
~~LiveData<String> = FlickrFetchr().fetchPhotos()~~
                             val flickrLiveData:
LiveData<List<GalleryItem>>                    =
FlickrFetchr().fetchPhotos()
        flickrLiveData.observe(
            this,
                                      Observer {
   ~~responseString~~galleryItems ->
```

```
                    Log.d(TAG, "Response received:
$    ~~responseString~~galleryItems")
                })
        }
        ...
}
```

Запустите приложение PhotoGallery, чтобы протестировать ваш парсер JSON. Вы должны увидеть вывод списка элементов галереи `ToString()` на панели **Logcat**. Если вы хотите изучить результаты более подробно, установите точку останова на строке журнала `Observer` и используйте отладчик для проверки `galleryItems` (рис. 24.7).

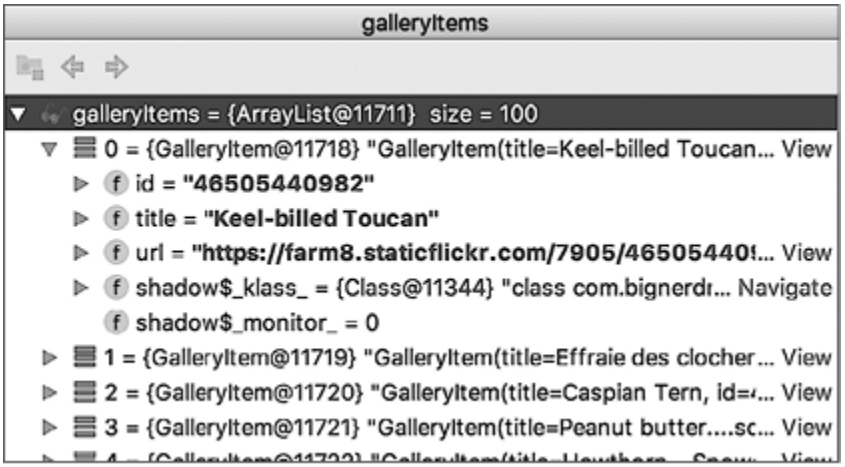


Рис. 24.7. Исследование ответа Flickr

Если выброшено исключение `UninitializedPropertyAccessException`, убедитесь, что ваш веб-запрос отформатирован правильно. В некоторых случаях (например, когда ключ API недействителен) Flickr API возвращает правильный код ответа (`200`), но пустое тело ответа, и Gson не сможет инициализировать ваши поздно инициализированные модели.

Работа с сетью после изменения конфигурации

Теперь ваше приложение умеет десериализовать данные JSON в объекты модели. Посмотрите, как ведет себя ваша реализация при изменении конфигурации. Запустите приложение, убедитесь, что у устройства или эмулятора включен автоповорот, а затем быстро поверните устройство пять-шесть раз.

Посмотрите на данные на панели **Logcat**, используя фильтр с запросом `PhotoGalleryFragment` и отключив программный перенос строк (рис. 24.8).

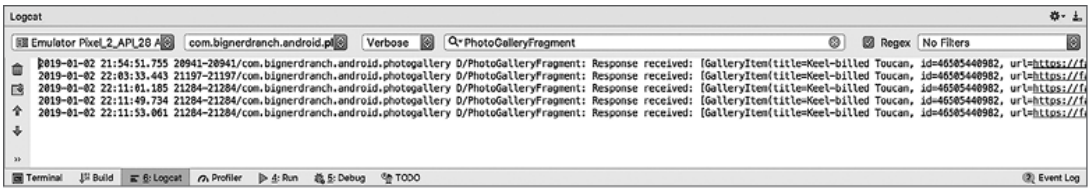


Рис. 24.8. Вывод Logcat после нескольких поворотов

Что здесь происходит? Каждый раз при повороте устройства выполняется новый сетевой запрос. Это связано с тем, что запрос выполняется в функции `onCreate(...)`. Поскольку каждый раз при вращении фрагмент уничтожается и воссоздается заново, создается новый лишний запрос на повторную загрузку данных. Это неправильно, так как выполняется лишняя работа. Запрос (и полученные на него данные) должен сохраняться после поворота, чтобы обеспечить оперативность работы пользователя (и избежать избыточного использования данных пользователя, если он не подключен к Wi-Fi).

Вместо того чтобы выдавать новый веб-запрос каждый раз, когда происходит изменение конфигурации, нужно получить данные фотографии лишь раз при запуске и отображении фрагмента на экране. Тогда вы можете разрешить веб-запросу

продолжить выполнение при изменении конфигурации путем кэширования результатов в памяти. Наконец, вы можете использовать результаты кэширования по мере их доступности, вместо того чтобы делать новый запрос.

ViewModel — это как раз то, что поможет вам с этой задачей. (Если вам нужно освежить знания о ViewModel, обратитесь к главе 4.)

Во-первых, добавьте зависимость lifecycle-extensions в файл app/build.gradle.

Листинг 24.29. Добавление зависимости lifecycle-extensions (app/build.gradle)

```
dependencies {
    ...
                                    implementation
'androidx.appcompat:appcompat:1.0.2'
                                    implementation
'androidx.lifecycle:lifecycle-extensions:2.0.0'
    ...
}
```

Далее создадим класс ViewModel с именем PhotoGalleryViewModel. Добавим свойство для хранения объекта «живых» данных, содержащего список элементов галереи. Выполним веб-запрос для получения данных фото при первой инициализации ViewModel и сохраним полученные данные в созданное нами свойство. Код должен получиться таким, как показано в листинге 24.30.

Листинг 24.30. Новый код ViewModel (PhotoGalleryViewModel.kt)

```
class PhotoGalleryViewModel : ViewModel() {

    val galleryItemLiveData: LiveData<List<GalleryItem>>

    init {
        galleryItemLiveData = FlickrFetchr().fetchPhotos()
    }
}
```

Функция `FlickrFetchr().fetchPhotos()` вызывается в блоке `init{}` модели `PhotoGalleryViewModel`. Она запускает запрос данных фото при первом создании `ViewModel`. Так как `ViewModel` создается только один рз в течение жизненного цикла владельца (при первом запросе из класса `ViewModelProviders`), запрос выполнится только один раз (при запуске пользователем `PhotoGalleryFragment`). Когда пользователь поворачивает устройство или иным образом инициирует изменение конфигурации, `ViewModel` останется в памяти, а воссозданная версия фрагмента сможет получить доступ к результатам оригинального запроса через `ViewModel`.

При таком устройстве репозиторий `FlickrFetchr` будет продолжать выполнять запрос даже в том случае, если пользователь заблаговременно откажется от хост-activity фрагмента. В вашем приложении результат запроса будет просто проигнорирован. В реальном же приложении вы можете кэшировать результаты в базе данных или другом локальном хранилище, поэтому имеет смысл позволить фрагменту продолжить выполнение запроса.

Если вы хотите вместо этого остановить запрос FlickrFetchr при выходе пользователя из фрагмента, можно научить FlickrFetchr сохранять объект Call, представляющий веб-запрос, и отменять запрос, когда ViewModel будет удален из памяти. Подробнее об этом рассказывается в разделе «Для любознательных: отмена запросов» далее в этой главе.

Измените код функции PhotoGalleryFragment.onCreate(...), чтобы получить доступ к ViewModel. Сохраните ссылку на ViewModel в свойстве. Удалите код, который взаимодействует с FlickrFetchr, так как теперь эту задачу решает PhotoGalleryViewModel.

Листинг 24.31. Получение экземпляра ViewModel от провайдера (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {

    private lateinit var photoGalleryViewModel:
PhotoGalleryViewModel
      private lateinit var photoRecyclerView:
RecyclerView

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

                                ~~val flickrLiveData:~~
~~LiveData<List<GalleryItem>> =~~
~~FlickrFetchr().fetchPhotos()~~
            ~~flickrLiveData.observe(~~
```

```
                this,
                Observer { galleryItems ->
                        Log.d(TAG, "Response
received: $galleryItems")
                })

        photoGalleryViewModel =
                ViewModelProviders.of(this).get
(PhotoGalleryViewModel::class.java)
    }

    ...
}
```

Как вы можете помнить, при первом запросе `ViewModel` для данного владельца жизненного цикла создается новый экземпляр `ViewModel`. Когда `PhotoGalleryFragment` уничтожается и создается заново из-за изменения конфигурации, например вращения, существующий `ViewModel` сохраняется. Следующие запросы к `ViewModel` возвращают тот же экземпляр, который был создан изначально.

Теперь научим `PhotoGalleryFragment` наблюдать за объектом `LiveData` в `PhotoGalleryViewModel` после создания представления фрагмента. А пока запишем, что данные были получены. Позднее мы будем использовать эти результаты для обновления содержимого утилизатора.

Листинг 24.32. Наблюдение за «живыми» данными ViewModel (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
```

```
    override fun onCreateView(
        ...
    ): View {
        ...
    }

       override fun onViewCreated(view: View,
savedInstanceState: Bundle?) {
                        super.onViewCreated(view,
savedInstanceState)
        photoGalleryViewModel.galleryItemLiveDa
ta.observe(
            viewLifecycleOwner,
            Observer { galleryItems ->
                Log.d(TAG, "Have gallery items
from ViewModel $galleryItems")
                            // Обновить данные,
поддерживающие представление утилизатора
            })
    }
    ...
}
```

Позже мы будем обновлять пользовательский интерфейс (например, представление утилизатора) в ответ на изменение данных. Внедрение наблюдения в функцию `onViewCreated(...)` гарантирует, что виджеты пользовательского интерфейса и другие объекты будут к этому готовы. Это также гарантирует, что будет правильно обрабатываться сценарий уничтожения фрагмента. В этом сценарии при повторном присоединении фрагмента

представление будет создано заново, и при создании в новое представление будет добавлено наблюдение за LiveData.

(Мы могли также увидеть, как началось наблюдение в функции onCreateView(...) или onActivityCreated(...). Код выполняется нормально, но менее явно показывает связь между наблюдаемыми «живыми» данными и жизненным циклом представления.)

Передача ViewLifecycleOwner в качестве параметра LifecycleOwner функции LiveData.observe(LifecycleOwner,Observer) гарантирует, что объект LiveData удалит наблюдателя при уничтожении представления фрагмента.

Запустите приложение. Отфильтруйте содержимое панели Logcat с помощью фильтра FlickrFetchr. Поверните эмулятор несколько раз. Вы должны увидеть на панели Logcat одно сообщение: FlickrFetchr: Response received, независимо от того, сколько раз повернули экран.

Отображение результатов в RecyclerView

Нашей последней задачей в этой главе будет перейти к представлению и получить RecyclerView фрагмента PhotoGalleryFragment, чтобы отобразить заголовки.

Сначала определите класс ViewHolder в PhotoGalleryFragment.

Листинг 24.33. Добавление реализации ViewHolder (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
```

```
        override fun onViewCreated(view: View,
savedInstanceState: Bundle?) {
            ...
        }

        private class PhotoHolder(itemTextView:
TextView)
            : RecyclerView.ViewHolder(itemTextView)
{

            val bindTitle: (CharSequence) -> Unit =
itemTextView::setText
        }
        ...
}
```

Далее добавьте `RecyclerView.Adapter`, чтобы он выдавал `PhotoHolder` из галереи.

Листинг 24.34. Добавление реализации RecyclerView.Adapter (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
        private class PhotoHolder(itemTextView:
TextView)
            : RecyclerView.ViewHolder(itemTextView)
{

            val bindTitle: (CharSequence) -> Unit =
itemTextView::setText
        }
```

```
        private class PhotoAdapter(private val
    galleryItems: List<GalleryItem>)
            : RecyclerView.Adapter<PhotoHolder>() {

            override fun onCreateViewHolder(
                    parent: ViewGroup,
                    viewType: Int
            ): PhotoHolder {
                                    val  textView  =
    TextView(parent.context)
                return PhotoHolder(textView)
            }

               override fun getItemCount(): Int =
    galleryItems.size

              override fun onBindViewHolder(holder:
    PhotoHolder, position: Int) {
                                  val  galleryItem  =
    galleryItems[position]
                holder.bindTitle(galleryItem.title)
            }
        }
        ...
    }
```

Теперь, когда мы закончили с `RecyclerView`, добавьте код для прикрепления адаптера с обновленными данными элементов галереи в момент обратного вызова наблюдателя `LiveData`.

Листинг 24.35. Добавление адаптера для наблюдения за доступностью и изменением данных (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
       override fun onViewCreated(view: View,
savedInstanceState: Bundle?) {
                        super.onViewCreated(view,
savedInstanceState)
        photoGalleryViewModel.galleryItemLiveDa
ta.observe(
            this,
            Observer { galleryItems ->
                        Log.d(TAG, "Have gallery
items from ViewModel $galleryItems")
                       // Eventually, update data
backing the recycler view
                    photoRecyclerView.adapter =
PhotoAdapter(galleryItems)
            })

        return view
    }
    ...
}
```

На этом завершаем главу. Запустите PhotoGallery, и вы увидите текст для каждого элемента `GalleryItem` (см. рис. 24.2).

Для любознательных: альтернативные парсеры и форматы данных

Gson — это популярный, но это не единственный парсер для JSON. Moshi компании Square (**github.com/square/moshi**) — еще одна популярная библиотека, которая также обрабатывает данные JSON. Moshi некоторые идеи заимствовала у Gson, но налицо попытка сделать их более быстрыми и эффективными. Компания Square создала конвертер для Moshi, чтобы он мог работать непосредственно с Retrofit.

Retrofit не ограничивает вас одним только форматом JSON. Вы можете использовать XML или даже Protobufs. Для работы с Retrofit существует множество библиотек сериализации, так что вы можете использовать ту, которая лучше всего подходит для вашей конфигурации передачи данных.

Для любознательных: отмена запросов

В текущей реализации `PhotoGalleryFragment` осуществляет к своей `ViewModel`, `PhotoGalleryViewModel`, запрос на запуск веб-запроса на загрузку фотоданных. Если пользователь достаточно быстро нажмет кнопку «Назад» после запуска приложения, возможно, что веб-запрос продолжится и после того, как пользователь закроет activity. Это не вызовет утечки памяти, так как FlickrFetchr не содержит ссылок на компоненты, связанные с пользовательским интерфейсом, или на `ViewModel`.

Однако, поскольку вы игнорируете результаты, продолжение запроса приведет к очень маленькому расходу заряда батареи и ресурсов системы и, возможно, использованию платных данных, если пользователь находится не в безлимитной сети. Большого вреда нанесено не будет, так как размер данных, которые вы получаете, очень мал.

В большинстве реальных приложений вы, скорее всего, позволили бы запросу выполняться дальше, как и здесь. Но

вместо того чтобы игнорировать результат, вы будете где-нибудь кэшировать его, например в базе данных.

Так как в вашей текущей реализации вы не кэшируете результаты, вы могли бы вместо этого отменить запрос в момент уничтожения `ViewModel`. Для этого нужно было бы сохранить объект `Call`, представляющий собой веб-запрос. Затем можно отменить веб-запрос, вызвав `Call.cancel()` на сохраненные объекты вызова:

```
class SomeRespositoryClass {

            private lateinit var someCall:
Call<SomeResponseType>
    ...
    fun cancelRequestInFlight() {
        if (::someCall.isInitialized) {
            someCall.cancel()
        }
    }
}
```

При отмене объекта `Call` будет вызвана соответствующая функция `Callback.onFailure(...)`. Вы можете проверить значение `Call.isCancelled`, чтобы определить, появилась ли из-за отмены запроса ошибка (значение `true` означает, что запрос действительно был отменен).

Для подключения к жизненному циклу `ViewModel`, а точнее для отмены вызовов при уничтожении `ViewModel`, нужно переопределить функцию `ViewModel.onCleared()`. Эта функция вызывается тогда, когда `ViewModel` вот-вот будет уничтожен (например, когда пользователь выключает activity с помощью кнопки «Назад»).

```
class SomeViewModel : ViewModel() {

            private val someRepository =
SomeRespositoryClass()
    ...
    override fun onCleared() {
        super.onCleared()
        someRepository.cancelRequestInFlight()
    }
    ...
}
```

Для любознательных: управление зависимостями

`FlickrFetchr` предоставляет слой абстракции над источником метаданных фото Flickr. Другие компоненты (такие как `PhotoGalleryFragment`) используют эту абстракцию для получения данных Flickr, не думая о том, откуда поступают эти данные.

`FlickrFetchr` сам по себе не знает, как загружать данные JSON с помощью Flickr. Вместо этого `FlickrFetchr` берет данные у `FlickrApi`, подключается к этой конечной точке и выполняет фактическую работу по загрузке данных JSON. Говорят, что `FlickrFetchr`*зависит* от `FlickrApi`.

Вы инициализируете `FlickrApi` внутри блока `init` во `FlickrFetchr`:

```
class FlickrFetchr {
    ...
    private val flickrApi: FlickrApi

    init {
```

```
                    val retrofit: Retrofit =
Retrofit.Builder()
                .baseUrl("https://www.flickr.co
m/")
                .addConverterFactory(ScalarsCon
verterFactory.create())
                .build()

                                    flickrApi    =
retrofit.create(FlickrApi::class.java)
    }

    fun fetchContents(): LiveData<String> {
        ...
    }
}
```

Это прекрасно работает для простых приложений, но есть несколько вопросов, требующих рассмотрения.

Во-первых, сложно проводить с `FlickrFetchr` модульное тестирование. Из главы 20 нам известно, что целью модульного теста является проверка поведения класса и его взаимодействия с другими классами. Чтобы правильно провести модульный тест `FlickrFetchr`, вам нужно изолировать его от реального `FlickrApi`. А это сложно, если вообще возможно, поскольку `FlickrApi` инициализируется внутри блока инициализации `FlickrFetchr`.

Следовательно, нет никакой возможности предоставить `FlickrApi` во `FlickrFetchr` для тестирования. Это проблема, поскольку любой тест, запущенный с функцией `fetchContents()`, будет создавать сетевые запросы. Успех

ваших тестов будет зависеть от состояния сети и доступности внутреннего API Flickr на момент запуска теста.

Вторая проблема заключается в том, что создавать экземпляры `FlickrApi` сложно. Вы должны создать и настроить экземпляр `Retrofit`, прежде чем сможете создать экземпляр `FlickrApi`. Эта реализация требует от вас дублирования пяти строк кода конфигурации Retrofit в любом месте, где вы хотите создать экземпляр `FlickrApi`.

Наконец, создание нового экземпляра `FlickrApi` везде, где вы хотите его использовать, приводит к появлению чрезмерного числа объектов. Создание объекта стоит дорого, особенно с учетом ограниченности ресурсов мобильных устройств. Везде, где это целесообразно, вы должны совместно использовать экземпляры класса в вашем приложении и избегать ненужного выделения объектов. `FlickrApi` является идеальным кандидатом для совместного использования, так как у нее нет переменного состояния экземпляров.

Инъекция зависимостей (или DI) — это шаблон проектирования, который решает эти проблемы путем централизации логики создания зависимостей, таких как `FlickrApi`, и предоставления зависимостей классам, которые в них нуждаются. Применив этот подход к PhotoGallery, вы можете легко передать экземпляр `FlickrApi` во `FlickrFetchr` каждый раз при создании нового экземпляра `FlickrFetchr`. Использование DI позволит вам:

- инкапсулировать логику инициализации `FlickrApi` в общем месте за пределами `FlickrFetchr`;

- использовать синглтон `FlickrApi` во всем приложении;

- заменить имитационную версию `FlickrApi` при модульном тестировании.

Применение шаблона DI к `FlickrFetchr` может выглядеть примерно так:

```
class FlickrFetchr(flickrApi: FlickrApi) {
    fun fetchContents(): LiveData<String> {
        ...
    }
}
```

Обратите внимание, что в DI синглтоны используются не для всех зависимостей. Во `FlickrFetchr` передается экземпляр `FlickrApi`. Этот механизм построения `FlickrFetchr` дает гибкость для предоставления нового экземпляра или общего экземпляра `FlickrApi`, основанного на вашем случае использования.

DI — это обширная тема с множеством аспектов, и разговор о ней лежит далеко за пределами Android. Здесь мы лишь затронули ее. Концепции DI посвящены целые книги, и существует множество библиотек, облегчающих реализацию DI. Если вы хотите использовать DI в своем приложении, стоит подумать об использовании одной из этих библиотек. Это поможет вам сократить количество кода, который придется писать для реализации шаблона.

На момент написания этой книги официальной рекомендуемой Google библиотекой для реализации DI на Android является Dagger 2. Подробную документацию, примеры кода и обучающие материалы по DI с использованием Dagger 2 вы можете найти по адресу **google.github.io/dagger**.

Упражнение. Добавление пользовательского десериализатора Gson

Ответ JSON с сайта Flickr содержит несколько слоев вложенных данных (рис. 24.6). В разделе «Десериализация JSON-текста в объекты модели» вы создавали объекты модели для сопоставления непосредственно в иерархию JSON. Но что, если вам не нужны данные во внешних слоях? Круто было бы не загромождать свою кодовую базу ненужными объектами модели?

По умолчанию Gson сопоставляет все данные JSON непосредственно с объектами вашей модели, сопоставляя имена свойств Kotlin (или аннотаций `@SerializedName`) с именами полей JSON. Вы можете настроить это поведение, определив пользовательский `com.google.gson.JsonDeserializer`.

Для этой задачи нам потребуется реализовать пользовательский десериализатор для удаления самого внешнего слоя данных JSON (слоя, который сопоставляется во `FlickrResponse`). Десериализатор должен вернуть объект `PhotoResponse`, заполненный на основе данных JSON. Для этого создайте новый класс, который является расширением `com.google.gson.JsonDeserializer`, и переопределите функцию `deserialize(...)`:

```
class PhotoDeserializer : JsonDeserializer<PhotoResponse> {

    override fun deserialize(
        json: JsonElement,
        typeOfT: Type?,
        context: JsonDeserializationContext?
```

```
    ): PhotoResponse {
            // Вытяните объект фотографий из
JsonElement
             // и преобразуйте его в объект
PhotoResponse
    }
}
```

Посмотрите документацию по Gson API, чтобы узнать, как реализовать парсинг с помощью `JsonElement` и преобразовать его в объект модели. (Подсказка: посмотрите документацию по `JsonElement`, `JsonObject` и `Gson`.)

После создания десериализатора измените код инициализации `FlickrFetchr`:

- используйте `GsonBuilder` для создания экземпляра Gson и зарегистрируйте свой пользовательский десериализатор как адаптер типа;
- создайте экземпляр `retrofit2.converter.gson.GsonConverterFactory`, использующий `Gson` в качестве конвертера;
- измените конфигурацию экземпляра Retrofit, чтобы использовать пользовательский экземпляр конвертера Gson в качестве заводского экземпляра.

Наконец, удалите `FlickrResponse` из вашего проекта и измените весь зависимый код.

Упражнение. Навигация по страницам

По умолчанию функция `getRecent` возвращает одну страницу со 100 результатами. При помощи дополнительного параметра `page` можно получить вторую, третью и так далее страницу результатов.

Для этого упражнения посмотрите библиотеку Jetpack Paging Library (**developer.android.com/topic/libraries/architecture/paging**) и используйте ее для реализации страниц в приложении PhotoGallery. В этой библиотеке есть фреймворк для загрузки данных вашего приложения по мере необходимости. Вы можете реализовать эту функцию вручную, но при использовании библиотеки на это уйдет меньше времени и будет меньше ошибок.

Упражнение. Динамическая настройка количества столбцов

В настоящее время количество столбцов в сетке фиксированно (три). Внесите изменения в свой код, чтобы количество столбцов могло динамически изменяться, а в альбомной ориентации и на больших устройствах отображалось больше столбцов.

В простом решении можно было бы предоставить целочисленный ресурс с квалификаторами для разных ориентаций и/или размеров экранов — по аналогии с тем, как мы предоставляли разные макеты для разных размеров экранов в главе 17. Целочисленные ресурсы должны размещаться в папке(-ах) `res/values`. За дополнительной информацией обращайтесь к документации разработчика Android.

Предоставление ресурсов с квалификаторами не отличается динамизмом. Чтобы усложнить задачу (и повысить гибкость реализации), вычисляйте и задавайте количество столбцов каждый раз при создании представления фрагмента.

Количество столбцов должно вычисляться на основании текущей ширины `RecyclerView` и заранее определенной постоянной ширины столбца.

Возникает только одна проблема: количество столбцов не может вычисляться в функции `onCreateView()`, потому что размеры `RecyclerView` еще не определены. Вместо этого реализуйте `ViewTreeObserver.OnGlobalLayoutListener` и разместите код вычисления в `onGlobalLayout()`. Добавьте слушателя к `RecyclerView` функцией `addOnGlobalLayoutListener()`.

25. Классы Looper, Handler и HandlerThread

После загрузки и разбора JSON-контента с сайта Flickr нашей следующей задачей станет загрузка и вывод изображений. В этой главе вы научитесь использовать классы `Looper`, `Handler` и `HandlerThread` для динамической загрузки и вывода фотографий в PhotoGallery. В этой главе вы разберетесь в том, для каких задач предназначен основной поток, а для каких — фоновые потоки. Наконец, вы узнаете, как передавать данные между основным и фоновым потоками.

Подготовка RecyclerView к выводу изображений

Текущая реализация `PhotoHolder` в `PhotoGalleryFragment` просто предоставляет виджеты `TextView` для вывода объектом `GridLayoutManager` компонента `RecyclerView`. В каждом представлении `TextView` выводится содержимое заголовка `GalleryItem`.

Для вывода фотографий внесите изменения в `PhotoHolder`, чтобы вместо текстовых представлений предоставлялись `ImageView`. В конечном итоге `ImageView` выведет фотографию, загруженную в поле `url` экземпляра `GalleryItem`.

Начнем с создания нового файла макета для элементов фотогалереи в файле `list_item_gallery.xml`. Макет будет состоять из единственного виджета `ImageView` (листинг 25.1).

Листинг 25.1. Макет элементов галереи (res/layout/list_item_gallery.xml)

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<ImageView
xmlns:android="http://schemas.android.com/apk/res/android"
            android:layout_width="match_parent"
            android:layout_height="120dp"
            android:layout_gravity="center"
            android:scaleType="centerCrop"/>
```

Эти виджеты ImageView находятся под управлением экземпляра GridLayoutManager компонента RecyclerView, что означает, что их ширина будет величиной переменной. При этом высота будет оставаться фиксированной. Чтобы наиболее эффективно использовать пространство виджета ImageView, следует задать его свойству scaleType значение centerCrop. С этим значением изображение выравнивается по центру и масштабируется, чтобы меньшая сторона была равна размеру представления, а большая усекалась с обеих сторон.

Также измените код класса PhotoHolder, чтобы вместо TextView он содержал ImageView. Замените bindTitle функцией, назначающей объект Drawable виджету ImageView.

Листинг 25.2. Обновление PhotoHolder (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
        ~~private class PhotoHolder(private val itemTextView: TextView)~~
                                                                ÷
~~RecyclerView.ViewHolder(itemTextView) {~~
        private class PhotoHolder(private val itemImageView: ImageView)
```

```
                                                    :
RecyclerView.ViewHolder(itemImageView) {

                ~~val bindTitle: (CharSequence) ->~~
~~Unit = itemTextView::setText~~
        val bindDrawable: (Drawable) -> Unit =
itemImageView::setImageDrawable
    }
    ...
}
```

Ранее конструктор `PhotoHolder` предполагал, что ему будет передаваться просто объект `TextView`. Новая версия рассчитывает получить `ImageView`.

Измените функцию `onCreateViewHolder()` в `PhotoAdapter`, чтобы она заполняла файл `list_item_gallery.xml` и передавала его конструктору `PhotoHolder`. Добавьте ключевое слово `inner`, чтобы у `PhotoAdapter` появился прямой доступ к свойству `layoutInflater` родительской activity. (Вы можете получить заполнитель из `parent.context`, но позже вам понадобится доступ к другим свойствам и функциям в родительской activity, а использование `inner` это упрощает.)

Листинг 25.3. Обновление функции onCreateViewHolder() класса PhotoAdapter (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
      private inner class PhotoAdapter(private
val galleryItems: List<GalleryItem>)
        : RecyclerView.Adapter<PhotoHolder>() {
```

```
        override fun onCreateViewHolder(
            parent: ViewGroup,
            viewType: Int
        ): PhotoHolder {
                                ~~val textView =~~
~~TextView(activity)~~
                ~~return PhotoHolder(textView)~~
            val view = layoutInflater.inflate(
                R.layout.list_item_gallery,
                parent,
                false
            ) as ImageView
            return PhotoHolder(view)
        }
        ...
    }
    ...
}
```

Также нам понадобится временное изображение (плейсхолдер) для каждого виджета `ImageView`, которое будет отображаться до завершения загрузки изображения. Найдите файл `bill_up_close.png` в файле решений (**www.bignerdranch.com/solutions/AndroidProgramming4e.zip**) и поместите его в каталог `res/drawable`.

Внесите изменения в код функции `onBindViewHolder()` класса `PhotoAdapter`, чтобы временное изображение назначалось объектом `Drawable` виджета `ImageView`.

Листинг 25.4. Назначение временного изображения (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
      private inner class PhotoAdapter(private
val galleryItems: List<GalleryItem>)
        : RecyclerView.Adapter<PhotoHolder>() {
        ...
          override fun onBindViewHolder(holder:
PhotoHolder, position: Int) {
                              val galleryItem =
galleryItems[position]
                ~~holder.bindTitle(galleryItem.ti~~
~~tle)~~
                  val placeholder: Drawable =
ContextCompat.getDrawable(
                requireContext(),
                R.drawable.bill_up_close
            ) ?: ColorDrawable()
            holder.bindDrawable(placeholder)
        }
    }
    ...
}
```

Обратите внимание, что вы передаете пустой объект `ColorDrawable`, если `ContextCompat.getDrawable(...)` возвращает `null`.

Запустив приложение PhotoGallery, вы увидите набор увеличенных фотографий Билла (рис. 25.1).

Рис. 25.1. Билл повсюду

Подготовка к загрузке через URL

Ваш интерфейс AP-Retrofit пока не поддерживает загрузку изображения. Это нужно исправить прямо сейчас. Добавьте новую функцию, которая принимает на вход строку с URL-адресом и возвращает исполняемый объект вызова (`retrofit2.Call`), результатом которого будет `okhttp3.ResponseBody`.

Листинг 25.5. Изменение кода FlickrApi (api/FlickrApi.kt)

```
interface FlickrApi {
```

```
    ...
    @GET
        fun fetchUrlBytes(@Url url: String):
Call<ResponseBody>
}
```

Новая функция API выглядит слегка иначе. На вход ей подается URL-адрес, который используется для определения того, откуда загружать данные. Использование беспараметрической аннотации `@GET` в сочетании с аннотацией первого параметра в `fetchUrlBytes(...)` с `@Url` приводит к тому, что Retrofit полностью переопределяет базовый URL. Вместо этого Retrofit будет использовать URL, переданный в функцию `fetchUrlBytes(...)`.

Добавьте функцию в `FlickrFetchr`, чтобы загружать данные по заданномуURL-адресу и декодировать их в изображение `Bitmap` (листинг 25.6).

Листинг 25.6. Добавление загрузки изображения во FlickrFetchr (FlickrFetchr.kt)

```
class FlickrFetchr {
    ...
    @WorkerThread
    fun fetchPhoto(url: String): Bitmap? {
        val response: Response<ResponseBody> =
flickrApi.fetchUrlBytes(url).execute()
                                val bitmap =
response.body()?.byteStream()?.use(BitmapFactor
y::decodeStream)
        Log.i(TAG, "Decoded bitmap=$bitmap from
Response=$response")
```

```
            return bitmap
        }
    }
```

Здесь используется функция `Call.execute()`, которая синхронно выполняет веб-запрос. Как вы уже знаете, работа с сетью в основном потоке не разрешена. Аннотация `@WorkerThread` указывает, что эта функция должна вызываться только в фоновом потоке.

Однако аннотация только дает указания, но не создает фоновый поток и не перемещает туда задачу. Это уже ваша работа. (В аннотации `@WorkerThread` появится ошибка Lint, если вызываемая функция будет аннотирована с помощью `@MainThread` или `@UiThread`. Однако на момент написания этой книги функции жизненного цикла Android не аннотируются с помощью `@MainThread` или `@UiThread`, даже если все функции жизненного цикла выполняются в основном потоке.) В результате вы будете вызывать функцию `fetchPhoto(String)` из создаваемого фонового потока.

Объект `java.io.InputStream` извлекается из тела ответа с помощью функции `ResponseBody.byteStream()`. Получив поток байтов, мы передаем его функции `BitmapFactory.decodeStream(InputStream)`, которая создаст `Bitmap` из данных в потоке.

Ответный и байтовый потоки должны быть закрытыми. Так как `InputStream` реализует атрибут `Closeable`, то стандартная функция библиотеки Kotlin `use(...)` выполнит чистку при возвращении `BitmapFactory.decodeStream(...)`.

Наконец мы возвращаем растровое изображение, которое построила наша `BitmapFactory`. Благодаря этому интерфейс API и репозиторий теперь готовы к загрузке изображений.

Но основная работа еще впереди.

Множественные загрузки

Сейчас работа с сетью у приложения PhotoGalletry организована так: `PhotoGalleryViewModel` вызывает функцию `FlickrFetchr().fetchPhotos()` для загрузки JSON-данных с сайта Flickr. `FlickrFetchr` сразу же возвращает пустой объект `LiveData<List<GalleryItem>>` и выполняет асинхронный запрос Retrofit для получения данных с Flickr. Этот сетевой запрос выполняется в фоновом потоке.

По завершении загрузки данных `FlickrFetchr` разбирает данные JSON в списке `GalleryItem` и публикует список возвращаемому объекту `LiveData`. У каждого элемента `GalleryItem` есть URL, с которого загружается миниатюра фотографии.

Следующим шагом должна стать загрузка этих миниатюр. Как это сделать? По умолчанию `FlickrFetchr` запрашивает лишь 100 URL-адресов, поэтому в списке `GalleryItem` будет не более 100 URL-адресов. Тогда, как вариант, можно загружать изображения одно за другим, пока их не станет 100, затем уведомлять `ViewModel`, и в конечном счете выводить изображения в `RecyclerView`.

Однако единовременная загрузка всех миниатюр создает две проблемы. Во-первых, она займет довольно много времени, а пользовательский интерфейс не будет обновляться до момента ее завершения. На медленном подключении пользователям придется долго рассматривать стену из фотографий Билла.

Во-вторых, хранение полного набора изображений требует ресурсов. Сотня миниатюр легко уместится в памяти. Но что,

если их будет 1000? Что если вы захотите реализовать бесконечную прокрутку? Со временем свободная память будет исчерпана.

С учетом этих проблем реальные приложения часто загружают изображения только тогда, когда они должны выводиться на экране. Загрузка по мере надобности предъявляет дополнительные требования к `RecyclerView` и его адаптеру. Адаптер инициирует загрузку изображения как часть реализации `onBindViewHolder(...)`.

Ну и *как* это сделать? Вы можете выдавать отдельный асинхронный запрос на Retrofit для каждой загрузки изображения. Однако вам придется отслеживать все объекты `Call` и управлять их жизненным циклом по отношению к каждому контейнеру представления и самому фрагменту.

Вместо этого мы создадим фоновый поток. Этот поток будет получать и поочередно обрабатывать запросы на загрузку, а также предоставлять результирующее изображение для каждого отдельного запроса по мере завершения загрузки. Поскольку все запросы управляются фоновым потоком, без проблем можно удалять запросы или даже останавливать поток, вместо того чтобы управлять целой кучей отдельных запросов.

Создание фонового потока

Создайте новый класс `ThumbnailDownloader`, расширяющий `HandlerThread`. Определите для него конструктор, заглушку реализации функции `queueThumbnail()` и переопределение функции `quit()`, которая говорит о завершении потока (в конце главы эта информация нам пригодится).

Листинг 25.7. Исходная версия кода потока (ThumbnailDownloader.kt)

```
private const val TAG = "ThumbnailDownloader"

class ThumbnailDownloader<in T>
    : HandlerThread(TAG) {

    private var hasQuit = false

    override fun quit(): Boolean {
        hasQuit = true
        return super.quit()
    }

     fun queueThumbnail(target: T, url: String)
{
        Log.i(TAG, "Got a URL: $url")
    }
}
```

Классу передается один обобщенный аргумент <T>. Пользователю `ThumbnailDownloader` понадобится объект для идентификации каждой загрузки и определения элемента пользовательского интерфейса, который должен обновляться после завершения загрузки. Вместо того чтобы ограничивать пользователя одним конкретным типом объекта, мы используем обобщенный параметр и сделаем реализацию более гибкой.

Функция `queueThumbnail()` ожидает получить объект типа `T`, выполняющий функции идентификатора загрузки, и `String`

с URL-адресом для загрузки. Эта функция будет вызываться PhotoAdapter в его реализации onBindViewHolder(...).

Передача информации о жизненном цикле в поток

Так как единственной целью ThumbnailDownloader является загрузка и передача изображений в PhotoGalleryFragment, жизненный цикл потока привязывается к предполагаемой пользователем *продолжительности жизни* фрагмента. Иными словами, поток запускается, когда пользователь впервые открывает экран. Поток завершается, когда пользователь заканчивает работу с экраном (например, нажимая кнопку «Назад» или завершая приложение). Не нужно уничтожать, а затем воссоздавать поток, когда пользователь поворачивает устройство, так как экземпляр потока должен сохраняться после изменения конфигурации.

Жизненный цикл ViewModel соответствует циклу жизни фрагмента. Однако управление потоком в PhotoGalleryViewModel сделает реализацию более сложной, чем необходимо, а также приведет к проблемам с утечкой представлений. Было бы разумнее передать управление потоком какому-либо другому компоненту, например репозиторию (FlickrFetchr). В реальном приложении мы, скорее всего, сделали бы именно так. Но в данном случае — нет, так как это отвлечет вас от цели освоения HandlerThread.

В целях данной главы привяжите экземпляр ThumbnailDownloader непосредственно к вашему PhotoGalleryFragment. Сначала сохраните PhotoGalleryFragment таким образом, чтобы жизнь экземпляра фрагмента соответствовала предполагаемой пользователем жизни фрагмента.

Листинг 25.8. Сохранение PhotoGalleryFragment (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

        retainInstance = true
        ...
    }
    ...
}
```

(Обычно следует избегать сохранения фрагментов. Мы будем делать это только здесь, потому что сохранение фрагмента упрощает реализацию и позволяет нам сконцентрироваться на изучении того, как работает `HandlerThread`. Подробнее о последствиях сохранения фрагмента вы узнаете позже в разделе «Сохраненные фрагменты».)

Теперь, когда фрагмент сохранен, нужно запускать поток при вызове функции `PhotoGalleryFragment.onCreate(...)` и завершать его при вызове функции `PhotoGalleryFragment.onDestroy()`. Этого можно добиться, добавив код непосредственно в функции жизненного цикла `PhotoGalleryFragment`, но это избыточно усложнит класс. Вместо этого абстрагируйте код в `ThumbnailDownloader`, сделав жизненный цикл `ThumbnailDownloader`*публичным*.

Компонент, наблюдающий за жизненным циклом — *наблюдатель*, — наблюдает за жизненным циклом *владельца жизненного цикла*. Activity и фрагмент являются владельцами жизненного цикла — у них есть жизненный цикл и реализуется интерфейс `LifecycleOwner`.

Добавьте в `ThumbnailDownloader` реализацию интерфейса `LifecycleObserver` и наблюдение за функциями `onCreate(...)` и `onDestroy` владельца жизненного цикла. Пусть загрузчик эскизов запускает сам себя при вызове `onCreate(...)` и останавливается при вызове функции `onDestroy()`.

Листинг 25.9. Связь ThumbnailDownloader с жизненным циклом (ThumbnailDownloader.kt)

```
private const val TAG = "ThumbnailDownloader"

class ThumbnailDownloader<in T>
    : HandlerThread(TAG), LifecycleObserver {

    private var hasQuit = false

    override fun quit(): Boolean {
        hasQuit = true
        return super.quit()
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE
)
    fun setup() {
            Log.i(TAG, "Starting background
thread")
```

```
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_DESTRO
Y)
    fun tearDown() {
            Log.i(TAG, "Destroying background
thread")
    }

     fun queueThumbnail(target: T, url: String)
{
        Log.i(TAG, "Got a URL: $url")
    }
}
```

Реализация `LifecycleObserver` означает, что вы можете подписать `ThumbnailDownloader` на получение обратных вызовов жизненного цикла от любого владельца `LifecycleOwner`. Для этого используется аннотация `@OnLifecycleEvent(Lifecycle.Event)`, позволяющая ассоциировать функцию в вашем классе с обратным вызовом жизненного цикла. `Lifecycle.Event.ON_CREATE` регистрирует вызов функции `ThumbnailDownloader.setup()` при вызове функции `LifecycleOwner.onCreate(...)`. `Lifecycle.Event.ON_DESTROY` регистрирует вызов функции `ThumbnailDownloader.tearDown()`при вызове функции `LifecycleOwner.onDestroy()`.

Список доступных констант `Lifecycle.Event` можно посмотреть на странице API (**developer.android.com/reference/android/arch/lifecycle/Lifecycle.Event**).

(Кстати, LifecycleObserver, Lifecycle.Event и OnLifecycleEvent являются частью Jetpack, а именно пакета android.arch.lifecycle. Вы уже имеете доступ к этим классам, потому что добавили зависимость lifecycle-extensions в файл Gradle в главе 24.)

Далее вам нужно создать экземпляр ThumbnailDownloader и подписать его на получение обратных вызовов жизненного цикла из PhotoGalleryFragment. Выполните это в файле PhotoGalleryFragment.kt.

Листинг 25.10. Создание ThumbnailDownloader (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {

    private lateinit var photoGalleryViewModel:
PhotoGalleryViewModel
      private lateinit var photoRecyclerView:
RecyclerView
     private lateinit var thumbnailDownloader:
ThumbnailDownloader<PhotoHolder>

     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

        retainInstance = true

        photoGalleryViewModel =
                ViewModelProviders.of(this).get
(PhotoGalleryViewModel::class.java)
```

```
                        thumbnailDownloader =
ThumbnailDownloader()
        lifecycle.addObserver(thumbnailDownload
er)
    }
    ...
       override fun onViewCreated(view: View,
savedInstanceState: Bundle?) {
        ...
    }

    override fun onDestroy() {
        super.onDestroy()
        lifecycle.removeObserver(
            thumbnailDownloader
        )
    }
    ...
}
```

Вы можете указать любой тип для общего аргумента `ThumbnailDownloader`. Однако помните, что это должен быть тип объекта, который будет использоваться в качестве идентификатора для вашей загрузки. В нашем случае удобно использовать `PhotoHolder`, так как он также является местом, куда в конечном итоге будут направлены загруженные изображения.

Так как `Fragment` реализует `LifecycleOwner`, у него есть свойство `lifecycle`. Это свойство мы будем использовать для добавления наблюдателя в жизненный цикл фрагмента. Вызов `lifecycle.addObserver(thumbnailDownloader)` подписывает экземпляр загрузчика эскизов на получение

обратных вызовов жизненного цикла фрагмента. Теперь при вызове функции `PhotoGalleryFragment.onCreate(...)` вызывается функция `ThumbnailDownloader.setup()`. При вызове функции `PhotoGalleryFragment.onDestroy()` вызывается функция `ThumbnailDownloader.tearDown()`.

Функция `lifecycle.removeObserver(thumbnailDownloader)` во `Fragment.onDestroy()` вызывается для снятия `thumbnailDownloader` с роли наблюдателя за жизненным циклом при уничтожении экземпляра фрагмента. Вообще, можно было бы оставить эту работу сборщику мусора. Однако мы предпочитаем управлять ресурсами точно, а не ждать, когда произойдет уборка мусора. Это помогает легче отслеживать ошибки.

Запустите ваше приложение. Перед вами должна быть стена фотографий Билла. Проверьте записи в своем журнале на наличие сообщения `ThumbnailDownloader:Startingbackgroundthread`, чтобы убедиться, что функция `ThumbnailDownloader.setup()` была выполнена один раз. Нажмите кнопку «Назад», чтобы выйти из приложения и закончить (и уничтожить) `PhotoGalleryActivity` (и ее `PhotoGalleryFragment`). Проверьте отчеты журнала еще раз: сообщение `Thumbnail-Downloader:Destroyingbackgroundthread` говорит о том, что функция `ThumbnailDownloader.tearDown()` была выполнена один раз.

Запуск и остановка HandlerThread

Теперь, когда `ThumbnailDownloader` наблюдает за жизненным циклом `PhotoGalleryFragment`, добавьте в `ThumbnailDownloader` запуск при вызове функции

PhotoGalleryFragment.onCreate(...) и остановку при вызове функции PhotoGalleryFragment.onDestroy().

Листинг 25.11. Запуск и остановка потока ThumbnailDownloader (ThumbnailDownloader.kt)

```
class ThumbnailDownloader<in T>
    : HandlerThread(TAG), LifecycleObserver {
    ...
    @OnLifecycleEvent(Lifecycle.Event.ON_CREATE
)
    fun setup() {
            Log.i(TAG, "Starting background
thread")
        start()
        looper
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_DESTRO
Y)
    fun tearDown() {
          Log.i(TAG, "Destroying background
thread")
        quit()
    }

     fun queueThumbnail(target: T, url: String)
{
        Log.i(TAG, "Got a URL: $url")
    }
}
```

Парочка указаний по безопасности. Первое: обратите внимание, что вы получаете доступ к `looper` после вызова функции `start()` на `ThumbnailDownloader` (подробнее о `Looper` через мгновение). Это способ убедиться, что поток готов к выполнению, чтобы избежать потенциального (хотя и редко встречающегося) состояния гонки. До первого обращения к `looper` нет никаких гарантий, что была вызвана функция `onLooperPrepared()`, так что существует вероятность того, что вызовы `queueThumbnail(...)` будут неудачными из-за нулевого обработчика.

Второе: нужно вызывать функцию `quit()` для завершения потока. Это важно. Если вы не выйдете из `HandlerThreads`, он будет жить вечно. Как зомби. Или рок-н-ролл.

Наконец, в функции `PhotoAdapter.onBindViewHolder(...)` вызываем функцию `queueThumbnail()` потока и передаем целевую папку `PhotoHolder`, где в конечном итоге будет размещено изображение и URL-адрес `GalleryItem` для скачивания.

Листинг 25.12. Подключение ThumbnailDownloader (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
      private inner class PhotoAdapter(private
val galleryItems: List<GalleryItem>)
        : RecyclerView.Adapter<PhotoHolder>() {
        ...
          override fun onBindViewHolder(holder:
PhotoHolder, position: Int) {
                            val galleryItem =
galleryItems[position]
```

```
                ...
                thumbnailDownloader.queueThumbnail(
holder, galleryItem.url)
            }
        }
    }
```

Еще раз запустите PhotoGallery и проверьте содержимое панели **Logcat**. При прокрутке `RecyclerView` вы должны увидеть на панели **Logcat** строки, сигнализирующие о том, что `ThumbnailDownloader` получает каждый ваш запрос на загрузку. Стена фотографий Билла пока никуда не делась.

Теперь, когда у вас запущен `HandlerThread`, следующим шагом будет общение между основным потоком вашего приложения и вашим новым фоновым потоком.

Сообщения и обработчики сообщений

Специализированный поток будет загружать фотографии, но как он будет взаимодействовать с адаптером `RecyclerView` для их отображения, если не может напрямую обращаться к главному потоку? (Не забывайте, что фоновым потокам не разрешается выполнять код, в результате которого изменяется выводимый на экран контент, так как это разрешено только основному потоку. А основной поток не может выполнять долгосрочные задачи — это разрешено только фоновому потоку.)

Вспомните пример с обувным магазином и двумя продавцами-Флэшами. Фоновый Флэш завершил свой телефонный разговор с поставщиком, и теперь ему нужно сообщить Главному Флэшу о том, что обувь была заказана. Если Главный Флэш занят, Фоновый Флэш не может сделать это

немедленно. Ему придется подождать у стойки и перехватить Главного Флэша в свободный момент. Такая схема работает, но не слишком эффективно.

Лучше дать каждому Флэшу по почтовому ящику. Фоновый Флэш пишет сообщение о том, что обувь заказана, и кладет его в ящик Главного Флэша. Главный Флэш делает то же самое, когда он хочет сообщить Фоновому Флэшу о том, что какой-то товар закончился.

Идея почтового ящика чрезвычайно полезна. Возможно, у продавца имеется задача, которая должна быть выполнена скоро, но не прямо сейчас. В таком случае он кладет сообщение в свой почтовый ящик и обрабатывает его в свободное время.

В Android такой «почтовый ящик», используемый потоками, называется *очередью сообщений* (message queue). Поток, работающий с использованием очереди сообщений, называется *циклом сообщений* (message loop); он снова и снова проверяет новые сообщения, которые могли появиться в очереди (рис. 25.2).

Цикл сообщений состоит из потока и *объекта* `Looper`, управляющего очередью сообщений потока.

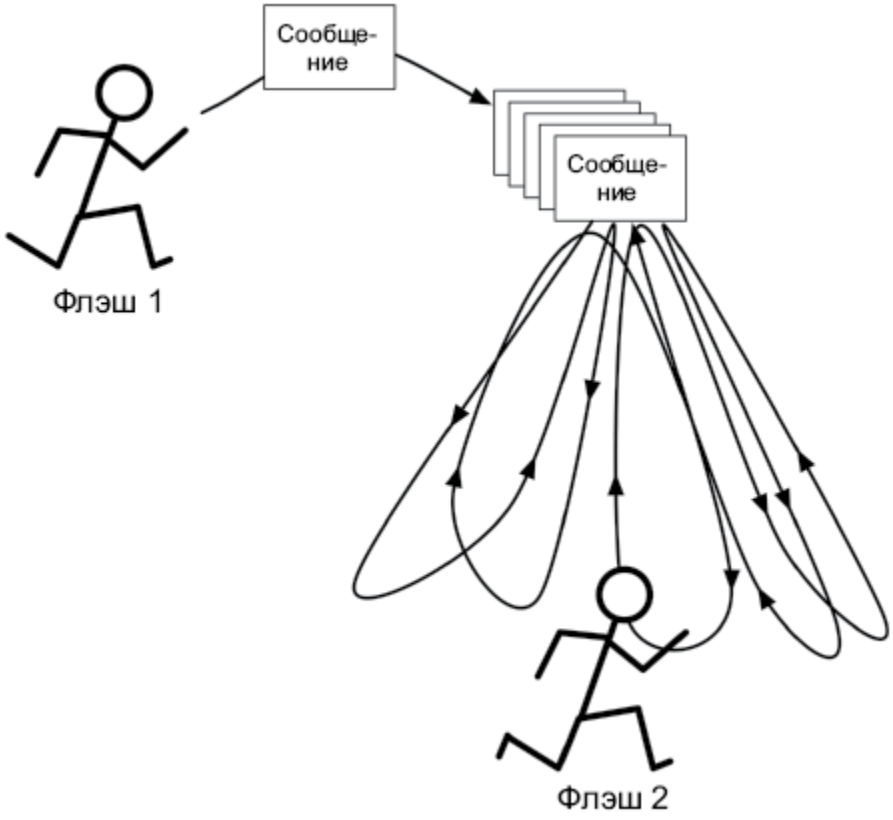


Рис. 25.2. Цикл сообщений

Главный поток представляет собой цикл сообщений, и у него есть управляющий объект, который извлекает сообщения из очереди сообщений и выполняет задачу, описанную в сообщении (рис. 25.3).

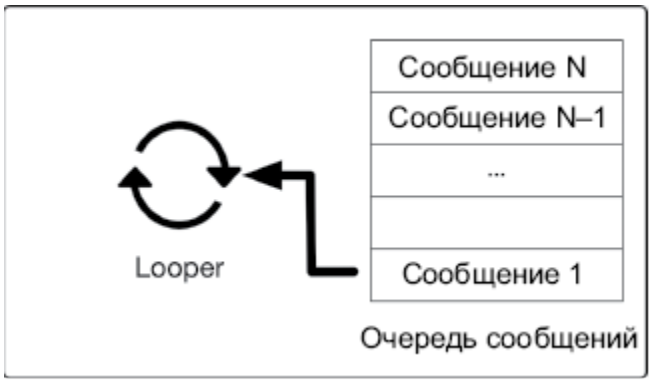


Рис. 25.3. Основной поток — это HandlerThread

Мы создадим фоновый поток, который тоже использует цикл сообщений. При этом будет использоваться класс `HandlerThread`, который предоставляет готовый объект `Looper`.

Основной и фоновый потоки будут взаимодействовать друг с другом, помещая сообщения в очередь друг друга с помощью обработчиков (рис. 25.4).

Прежде чем создавать сообщение, необходимо сначала понять, что оно собой представляет и какие отношения связывают его с *обработчиком сообщения* (message handler).

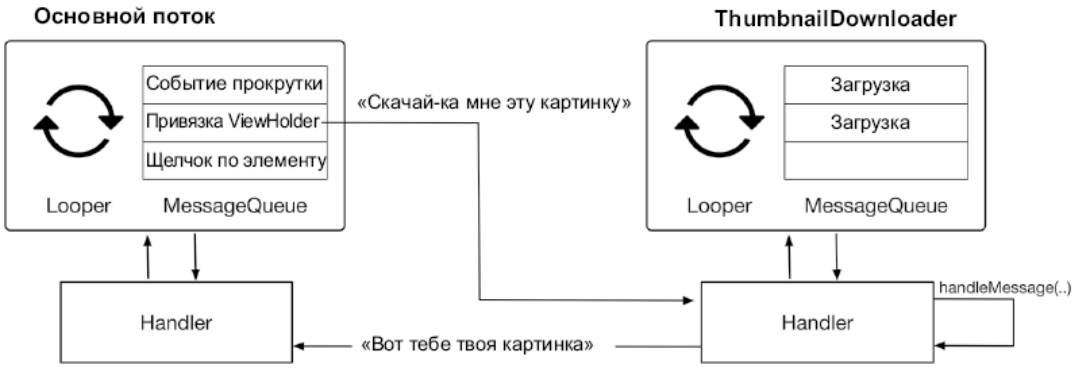


Рис. 25.4. Взаимодействие с обработчиками

Структура сообщения

Начнем с сообщений. Сообщения, которые Флэш кладет в почтовый ящик (свой собственный или принадлежащий другому продавцу), содержат не ободряющие записки типа «Ты бегаешь очень быстро», а описания задач, которые необходимо выполнить.

Сообщение является экземпляром класса `Message` и состоит из нескольких полей.

Для нашей реализации важны три поля:

- what — определяемое пользователем значение int, описывающее сообщение;
- obj — заданный пользователем объект, передаваемый с сообщением;
- target — приемник, то есть объект Handler, который будет обрабатывать сообщение.

Приемником сообщения Message является экземпляр Handler. Когда вы создаете сообщение, оно автоматически присоединяется к Handler. А когда ваше сообщение будет готово к обработке, именно Handler становится объектом, отвечающим за эту обработку.

Структура обработчика

Итак, для выполнения реальной работы с сообщениями необходимо иметь экземпляр Handler. Объект Handler — не просто приемник для обработки сообщений; он также предоставляет интерфейс для создания и отправки сообщений. Взгляните на рис. 25.5.

Сообщения Message отправляются и потребляются объектом Looper, потому что Looper является владельцем почтового ящика объектов Message. Соответственно, Handler всегда содержит ссылку на своего коллегу Looper.

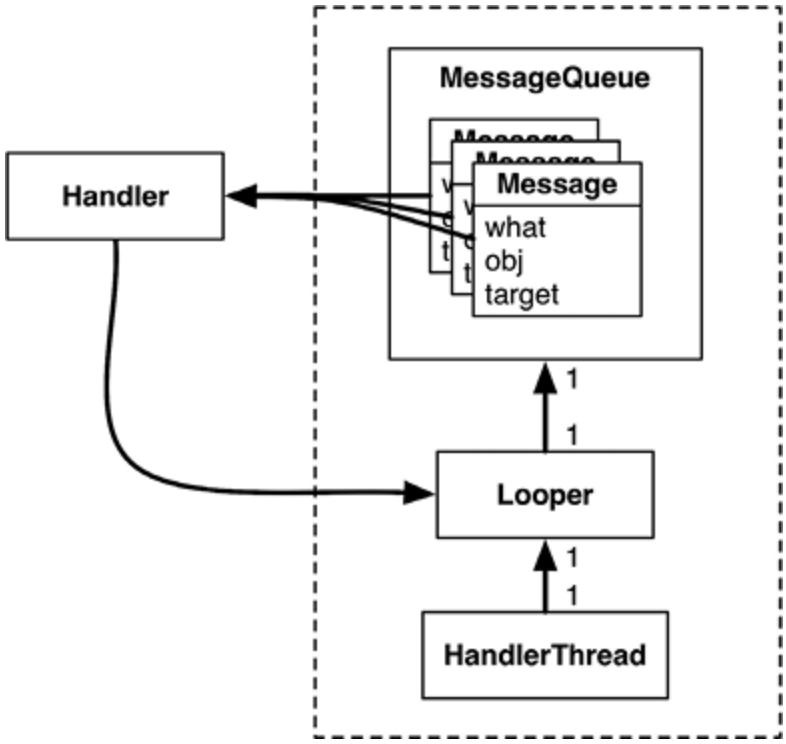


Рис. 25.5. Looper, Handler, HandlerThread и Messages

`Handler` всегда присоединяется ровно к одному объекту `Looper`, а `Message` присоединяется ровно к одному объекту `Handler`, называемому его *приемником*. Объект `Looper` обслуживает целую очередь сообщений `Message`. Многие сообщения `Message` могут содержать ссылку на один целевой объект `Handler` (рис. 25.5).

К одному объекту `Looper` могут быть присоединены несколько объектов `Handler` (рис. 25.6). Это означает, что сообщения `Message` объекта `Handler` могут сосуществовать с сообщениями другого объекта `Handler`.

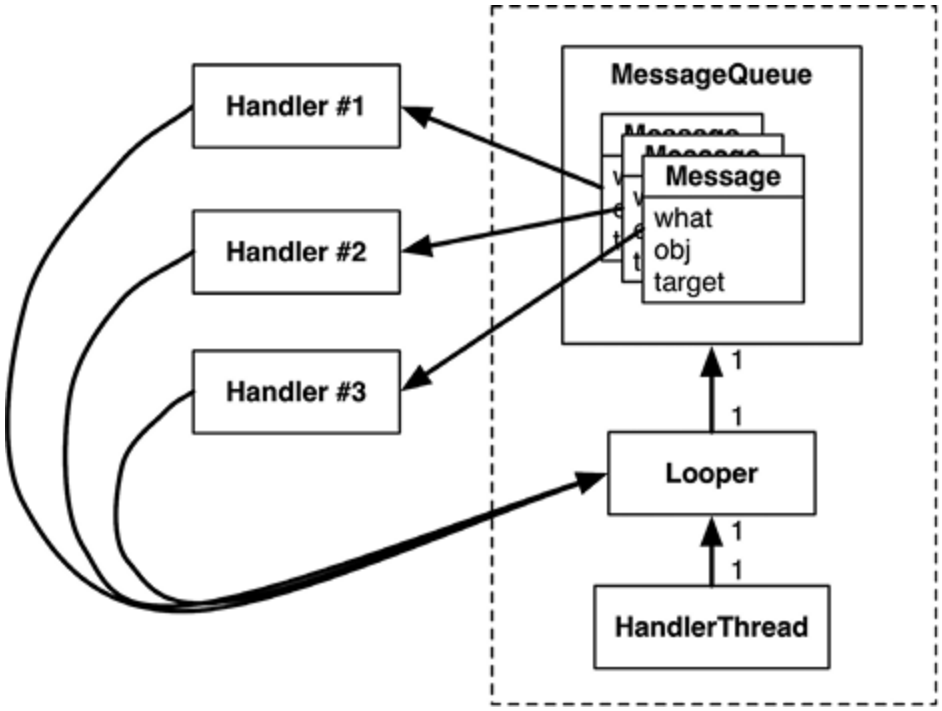


Рис. 25.6. Несколько объектов Handler, один объект Looper

Использование обработчиков

Обычно приемные объекты `Handler` сообщений не задаются вручную. Лучше построить сообщение вызовом функции `Handler.obtainMessage(...)`. Вы передаете функции другие поля сообщения, а она автоматически назначает приемник.

Функция `Handler.obtainMessage(...)` использует общий пул объектов, чтобы избежать создания новых объектов `Message`, поэтому он также работает более эффективно, чем простое создание новых экземпляров.

Когда объект `Message` будет получен, вы можете вызвать функцию `sendToTarget()`, чтобы отправить сообщение его обработчику. Обработчик помещает сообщение в конец очереди сообщений объекта `Looper`.

Мы собираемся получить сообщение и отправить его приемнику в реализации `queueThumbnail()`. В поле `what` сообщения будет содержаться константа, определяемая под именем `MESSAGE_DOWNLOAD`. В поле `obj` будет содержаться объект типа `T`, предназначенный для идентификации загрузки. В нашем случае это экземпляр `PhotoHolder`, переданный адаптером функции `queueThumbnail()`.

Когда объект `Looper` добирается до конкретного сообщения в очереди, он передает сообщение приемнику сообщения для обработки. Как правило, сообщение обрабатывается в реализации `Handler.handleMessage(...)` приемника.

Схема отношений между объектами изображена на рис. 25.7.

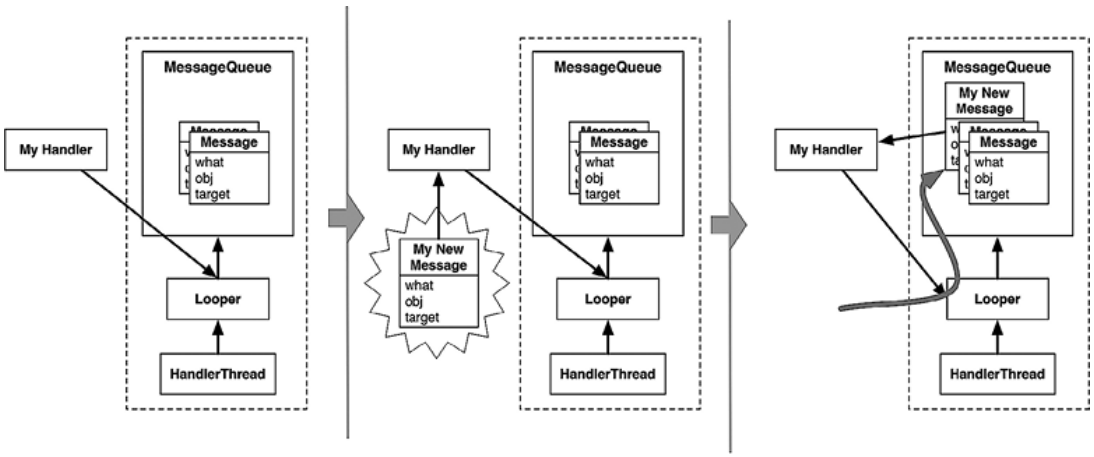


Рис. 25.7. Создание сообщения и его отправка

В нашем случае реализация `handleMessage(...)` будет использовать `FlickrFetchr` для загрузки байтов по URL-адресу и их преобразования в растровое изображение.

Добавьте константы и переменные из листинга 25.13.

Листинг 25.13. Добавление констант и переменных (ThumbnailDownloader.kt)

```
private const val TAG = "ThumbnailDownloader"
private const val MESSAGE_DOWNLOAD = 0

class ThumbnailDownloader<in T>
    : HandlerThread(TAG), LifecycleObserver {

    private var hasQuit = false
        private lateinit var requestHandler:
Handler
                private val requestMap =
ConcurrentHashMap<T, String>()
    private val flickrFetchr = FlickrFetchr()
    ...
}
```

Значение MESSAGE_DOWNLOAD будет использоваться для идентификации сообщений как запросов на загрузку. (ThumbnailDownloader присваивает его полю what создаваемых сообщений загрузки.)

В переменной requestHandler будет храниться ссылка на объект Handler, отвечающий за постановку в очередь запросов на загрузку в фоновом потоке ThumbnailDownloader. Этот объект также будет отвечать за обработку сообщений запросов на загрузку при извлечении их из очереди.

Переменная requestMap содержит ConcurrentHashMap — разновидность HashMap, безопасную по отношению к потокам. В данном случае использование объекта-идентификатора типа T запроса на загрузку в качестве ключа позволяет хранить и загружать URL-адрес, связанный с конкретным запросом. (Здесь объектом-идентификатором является PhotoHolder, так что по ответу на запрос можно легко вернуться к элементу

пользовательского интерфейса, в котором должно находиться загруженное изображение.)

В свойстве `flickrFetchr` хранится ссылка на экземпляр `FlickrFetchr`. Таким образом, весь код установки Retrofit будет выполняться только один раз в течение жизни потока. (Если создавать новый экземпляр Retrofit каждый раз, когда вы делаете веб-запрос, ваше приложение замедлится, особенно если вы делаете много запросов.)

Затем добавьте в `queueThumbnail(...)` код обновления переменной `requestMap` и постановки нового сообщения в очередь сообщений фонового потока.

Листинг 25.14. Отправка сообщения (ThumbnailDownloader.kt)

```
class ThumbnailDownloader<in T>
    : HandlerThread(TAG), LifecycleObserver {
    ...
     fun queueThumbnail(target: T, url: String)
{
        Log.i(TAG, "Got a URL: $url")
        requestMap[target] = url
        requestHandler.obtainMessage(MESSAGE_DO
WNLOAD, target)
            .sendToTarget()
    }
}
```

Сообщение берется непосредственно из переменной `requestHandler`, в результате чего поле `target` нового объекта `Message` немедленно заполняется переменной `requestHandler`. Это означает, что переменная `requestHandler` будет отвечать за обработку сообщения при

его извлечении из очереди сообщений. Поле what сообщения заполняется значением MESSAGE_DOWNLOAD. В поле obj заносится значение Ttarget (PhotoHolder в данном случае), переданное функцией queueThumbnail(...).

Новое сообщение представляет запрос на загрузку заданного Ttarget (PhotoHolder из RecyclerView). Вспомните, что реализация адаптера RecyclerView из PhotoGalleryFragment вызывает функцию queueThumbnail(...) из функции onBindViewHolder(...), передавая объект PhotoHolder, для которого загружается изображение, и URL-адрес загружаемого изображения.

Обратите внимание: в само сообщение URL-адрес не входит. Вместо этого переменная requestMap обновляется связью между идентификатором запроса (PhotoHolder) и URL-адресом запроса. Позднее мы получим URL-адрес из переменной requestMap, чтобы гарантировать, что для заданного экземпляра PhotoHolder всегда загружается последний из запрашивавшихся URL-адресов. (Это важно, потому что объекты ViewHolder в RecyclerView перерабатываются и используются повторно.)

Наконец, инициализируйте переменную requestHandler и определите, что будет делать объект Handler, когда сообщения извлекаются из очереди и передаются ему.

Листинг 25.15. Обработка сообщения (ThumbnailDownloader.kt)

```
class ThumbnailDownloader<in T>
    : HandlerThread(TAG), LifecycleObserver {
    ...
                private    val    requestMap    =
ConcurrentHashMap<T, String>()
    private val flickrFetchr = FlickrFetchr()
```

```
    @Suppress("UNCHECKED_CAST")
    @SuppressLint("HandlerLeak")
    override fun onLooperPrepared() {
        requestHandler = object : Handler() {
                override fun handleMessage(msg:
Message) {
                                if (msg.what ==
MESSAGE_DOWNLOAD) {
                    val target = msg.obj as T
                      Log.i(TAG, "Got a request
for URL: ${requestMap[target]}")
                    handleRequest(target)
                }
            }
        }
    }
    ...
     fun queueThumbnail(target: T, url: String)
{
        ...
    }

    private fun handleRequest(target: T) {
        val url = requestMap[target] ?: return
                              val   bitmap   =
flickrFetchr.fetchPhoto(url) ?: return
    }
}
```

При импорте библиотеки `Message` обязательно выберите из предложенных опций `android.os.Message`.

Функция Handler.handleMessage(...) реализуется в подклассе Handler внутри onLooperPrepared(). Функция HandlerThread.onLooperPrepared() вызывается до того, как Looper впервые проверит очередь, поэтому она хорошо подходит для создания реализации Handler.

В коде Handler.handleMessage(...) мы проверяем тип сообщения, читаем значение obj (которое имеет тип T и служит идентификатором для запроса) и передаем его функции handleRequest(...). (Вспомните, что Handler.handleMessage(...) будет вызываться, когда сообщение загрузки извлечено из очереди и готово к обработке.)

Вся загрузка осуществляется в функции handleRequest(). Мы проверяем существование URL-адреса, после чего передаем его новому экземпляру знакомого класса FlickrFetchr. При этом используется функция FlickrFetchr.getUrlBytes(...), которую мы так предусмотрительно создали в этой главе.

Аннотация @Suppress("UNCHECKED_CAST") при проверке сообщает Lint, что вы приводите msg.obj к типу T без предварительной проверки того, относится ли msg.obj к этому типу на самом деле. Это нормально, потому что вы единственный разработчик, работающий с кодом PhotoGallery. Вы сами управляете сообщениями, добавленными в очередь, и знаете, что на данный момент все сообщения в очереди имеют объектное поле, установленное в экземпляр PhotoHolder (которое соответствует T, указанному в ThumbnailDownloader).

Реализация обработчика, описанная выше, технически создает внутренний класс. Внутренние классы содержат ссылку на свой внешний класс (в данном случае

ThumbnailDownloader), что в свою очередь может привести к утечке внешнего класса, если время жизни внутреннего класса больше, чем предполагаемое время жизни внешнего класса.

Проблемы тут получаются только в том случае, если обработчик прикреплен к объекту Looper основного потока. Предупреждение HandlerLeak убирается аннотацией @SuppressLint("HandlerLeak"), так как создаваемый обработчик прикреплен к looper фонового потока. Если вместо этого обработчик был прикреплен к looper основного потока, то он может и не собирать мусор. Если бы произошла утечка, так как он также содержит ссылку на ThumbnailDownloader, ваше приложение также утеряло бы экземпляр ThumbnailDownloader.

Блокировать предупреждения Lint нужно только в том случае, если вы действительно понимаете, откуда взялось предупреждение и почему его блокирование в данном сценарии безопасно.

Запустите приложение PhotoGallery и проверьте на панели **LogCat** ваши подтверждающие команды регистрации.

Разумеется, запрос не будет полностью обработан до момента назначения изображения в объекте PhotoHolder, поступившем от PhotoAdapter. Однако эта операция относится к пользовательскому интерфейсу, поэтому она должна выполняться в главном потоке.

До настоящего момента мы ограничивались использованием обработчиков и сообщений в одном потоке — помещением сообщений в собственный почтовый ящик ThumbnailDownloader. В следующем разделе вы увидите, как ThumbnailDownloader использует Handler для отправки запросов главному потоку.

Передача обработчиков

Итак, вы знаете, как спланировать выполнение работы в фоновом потоке из главного потока с использованием значения переменной `requestHandler` объекта `ThumbnailDownloader`. Соответствующая схема изображена на рис. 25.8.

Аналогичным образом можно планировать операции в главном потоке из фонового потока с использованием экземпляра `Handler`, присоединенного к главному потоку (рис. 25.9).

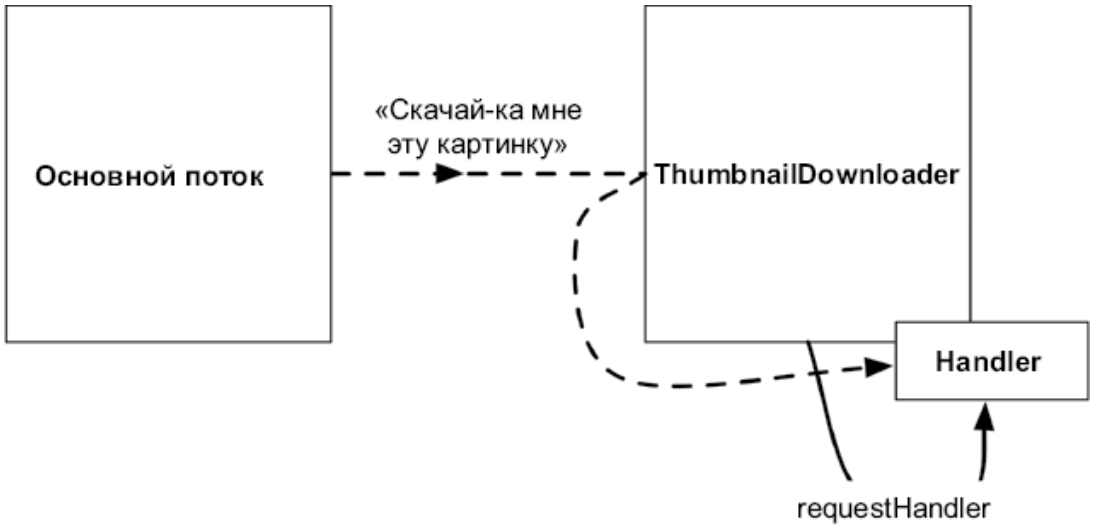


Рис. 25.8. Планирование операций в ThumbnailDownloader из главного потока

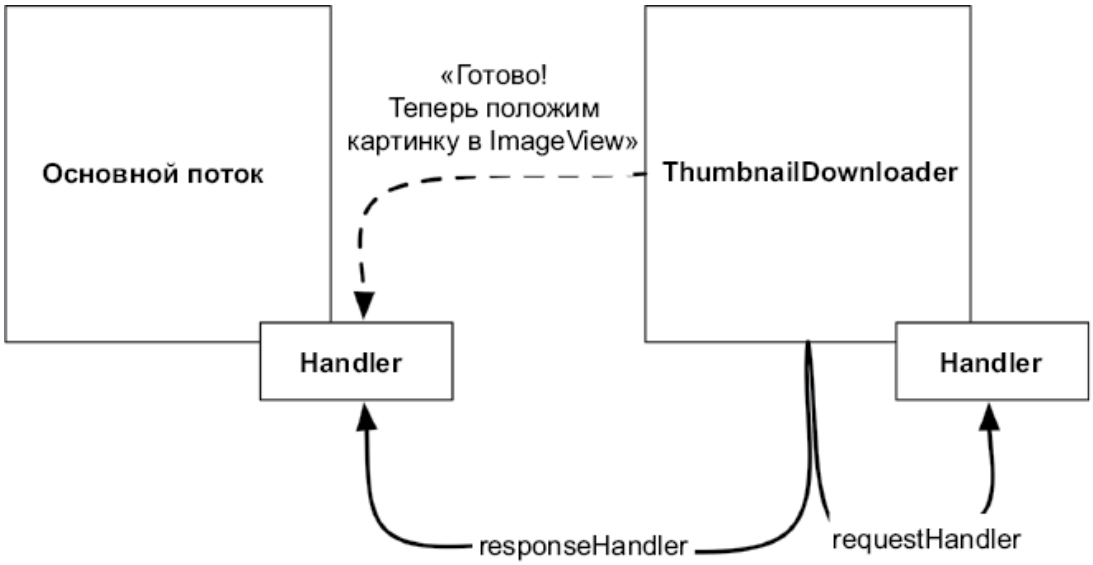


Рис. 25.9. Планирование операций в главном потоке из ThumbnailDownloader

Главный поток представляет собой цикл сообщений с обработчиками и `Looper`. При создании экземпляра `Handler` в главном потоке он ассоциируется с экземпляром `Looper` главного потока. Затем этот экземпляр `Handler` можно передать другому потоку. Переданный экземпляр `Handler` сохраняет связь с `Looper` потока-создателя. Все сообщения, за которые отвечает `Handler`, будут обрабатываться в очереди главного потока.

В файле `ThumbnailDownloader.kt` добавьте упоминавшуюся выше переменную `responseHandler` для хранения экземпляра `Handler`, переданного из главного потока. Затем замените конструктор другим, который получает `Handler` и задает переменную, и добавьте интерфейс слушателя для передачи ответов (загруженных изображений) запрашивающей стороне (главному потоку).

Листинг 25.16. Добавление responseHandler (ThumbnailDownloader.kt)

```
class ThumbnailDownloader<in T>(
    private val responseHandler: Handler,
       private val onThumbnailDownloaded: (T,
Bitmap) -> Unit
) : HandlerThread(TAG), LifecycleObserver {
    ...
}
```

Свойство типа функции, определенное в новом конструкторе, будет рано или поздно использовано, когда полностью загруженное изображение появится в интерфейсе. Использование слушателя передает ответственность за обработку загруженного изображения другому классу, а не `ThumbnailDownloader` (в данном случае `PhotoGalleryFragment`). Тем самым задача загрузки отделяется от задачи обновления пользовательского интерфейса (связывания изображений с `ImageView`), чтобы класс `ThumbnailDownloader` при необходимости мог использоваться для загрузки данных других разновидностей объектов `View`.

Затем измените класс `PhotoGalleryFragment` так, чтобы он передавал классу `ThumbnailDownloader` объект `Handler`, присоединенный к главному потоку. Также назначьте анонимную функцию для обработки загруженного изображения после завершения загрузки.

Листинг 25.17. Подключение к обработчику ответа (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
```

```
    override fun onCreate(savedInstanceState:
Bundle?) {
        ...
                            thumbnailDownloader =
ThumbnailDownloader()
        val responseHandler = Handler()
        thumbnailDownloader =
                ThumbnailDownloader(responseHan
dler) { photoHolder, bitmap ->
                                    val drawable =
BitmapDrawable(resources, bitmap)
                    photoHolder.bindDrawable(dr
awable)
                }
        lifecycle.addObserver(thumbnailDownload
er)
    }
    ...
}
```

Вспомните, что по умолчанию `Handler` присоединяется к объекту `Looper` для текущего потока. Поскольку объект `Handler` создан в `onCreate(...)`, он будет присоединен к объекту `Looper` главного потока.

Теперь `ThumbnailDownloader` имеет доступ к экземпляру `Handler`, связанному с экземпляром `Looper` главного потока, через поле `responseHandler`. В нем также есть реализация типа функции для реализации интерфейса при возврате `Bitmap`. В частности, функция, переданная в функцию высшего порядка `onThumbnailDownloaded`, устанавливает `Drawable` запрошенного `PhotoHolder` на только что загруженный `Bitmap`.

Аналогичным образом можно отправить главному потоку нестандартный объект Messsage, запрашивающий добавление изображения в пользовательский интерфейс, по аналогии с тем, как мы ставили в очередь запрос к фоновому потоку на загрузку изображения. Для этого потребуется другой подкласс Handler с переопределением функции handleMessage(...).

Однако вместо этого мы используем другую удобную функцию Handler — post (Runnable).

Handler.post(Runnable)— вспомогательная функция для отправки сообщений следующего вида:

```
var myRunnable: Runnable = object : Runnable {
    override fun run() {
        // Здесь ваш код
    }
}
var msg: Message = Message.obtain(someHandler,
myRunnable)
// Устанавливает msg.callback в myRunnable
```

Если у Message задано поле callback, то вместо передачи приемнику Handler при извлечении из очереди сообщений выполняется объект Runnable из поля callback. В функции ThumbnailDownloader.handleRequest() запишите Runnable в очередь основного потока через responseHandler (листинг 25.18).

Листинг 25.18. Загрузка и вывод изображений (ThumbnailDownloader.kt)

```
class ThumbnailDownloader<in T>(
    private val responseHandler: Handler,
```

```
        private val onThumbnailDownloaded: (T,
Bitmap) -> Unit
) : HandlerThread(TAG), LifecycleObserver {
    ...
    private fun handleRequest(target: T) {
        val url = requestMap[target] ?: return
                                val bitmap =
flickrFetchr.fetchPhoto(url) ?: return

        responseHandler.post(Runnable {
              if (requestMap[target] != url ||
hasQuit) {
                return@Runnable
            }

            requestMap.remove(target)
                onThumbnailDownloaded(target,
bitmap)
        })
    }
}
```

А поскольку `responseHandler` связывается с `Looper` главного потока, весь код функции `run()` в `Runnable` будет выполнен в главном потоке.

Что делает этот код? Сначала он проверяет `requestMap`. Такая проверка необходима, потому что `RecyclerView` заново использует свои представления. К тому времени, когда `ThumbnailDownloader` завершит загрузку `Bitmap`, может оказаться, что виджет `RecyclerView` уже переработал `ImageView` и запросил для него изображение с другого URL-адреса. Эта проверка гарантирует, что каждый объект

PhotoHolder получит правильное изображение, даже если за прошедшее время был сделан другой запрос.

Затем проверяется hasQuit. Если выполнение ThumbnailDownloader уже завершилось, выполнение каких-либо обратных вызовов небезопасно.

Наконец, мы удаляем из requestMap связь «PhotoHolder—URL» и назначаем изображение для PhotoHolder.

Прослушивание жизненного цикла представления

Перед запуском приложения PhotoGallery и просмотром снимков нужно обратить внимание на одну опасность. Если пользователь поворачивает экран, ThumbnailDownloader может повиснуть на недействительном PhotoHolder. Ваше приложение может аварийно завершить работу, если программа ThumbnailDownloader попытается отправить растровое изображение PhotoHolder, которое было частью представления, уничтоженного во время вращения.

Исправьте утечку, удалите все запросы из очереди при уничтожении представления фрагмента. Для этого ThumbnailDownloader должен иметь информацию о жизненном цикле представления фрагмента. (Вспомните, что жизненные циклы фрагмента и его представления расходятся, так как вы сохраняете фрагмент: представление будет уничтожено при повороте, но сам экземпляр фрагмента сохранится.)

Во-первых, выполните рефакторинг кода наблюдателя за жизненным циклом фрагмента, чтобы освободить место для реализации второго наблюдателя за жизненным циклом.

Листинг 25.19. Рефакторинг наблюдателя за жизненным циклом фрагмента (ThumbnailDownloader.kt)

```
class ThumbnailDownloader<in T>(
    private val responseHandler: Handler,
       private val onThumbnailDownloaded: (T,
Bitmap) -> Unit
) : HandlerThread(TAG), LifecycleObserver {

              val fragmentLifecycleObserver:
LifecycleObserver =
        object : LifecycleObserver {

            @OnLifecycleEvent(Lifecycle.Event.O
N_CREATE)
            fun setup() {
                Log.i(TAG, "Starting background
thread")
                start()
                looper
            }

            @OnLifecycleEvent(Lifecycle.Event.O
N_DESTROY)
            fun tearDown() {
                        Log.i(TAG, "Destroying
background thread")
                quit()
            }
        }

    private var hasQuit = false
```

```
    ...
        @OnLifecycleEvent(Lifecycle.Event.ON_CR
EATE)
        fun setup() {
            start()
            looper
        }

        @OnLifecycleEvent(Lifecycle.Event.ON_DE
STROY)
        fun tearDown() {
                Log.i(TAG, "Background thread
destroyed")
            quit()
        }
    ...
}
```

Определите нового наблюдателя, который в конце концов будет слушать обратные вызовы жизненного цикла с точки зрения фрагмента.

Листинг 25.20. Добавление наблюдателя жизненного цикла представления (ThumbnailDownloader.kt)

```
class ThumbnailDownloader<in T>(
    private val responseHandler: Handler,
       private val onThumbnailDownloaded: (T,
Bitmap) -> Unit
) : HandlerThread(TAG) {
```

```
                val    fragmentLifecycleObserver:
LifecycleObserver =
        object : LifecycleObserver {
        ...
    }

                  val    viewLifecycleObserver:
LifecycleObserver =
        object : LifecycleObserver {
           @OnLifecycleEvent(Lifecycle.Event.ON
_DESTROY)
           fun clearQueue() {
                     Log.i(TAG, "Clearing all
requests from queue")
                requestHandler.removeMessages(ME
SSAGE_DOWNLOAD)
                requestMap.clear()
           }
        }
    ...
}
```

При наблюдении за жизненным циклом представления фрагмента `Lifecycle.Event.ON_DestROY` сопоставляет его с функцией `Fragment.onDestroyView()`. Чтобы узнать, как константы `Lifecycle.Event` сопоставляются с обратными вызовами при наблюдении жизненного цикла представления фрагмента, загляните в раздел `getViewLifecycleOwner` для `Fragment` по адресу **developer.android.com/reference/kotlin/androidx/fragment/app/Fragment**.

Теперь научим PhotoGalleryFragment регистрировать восстановленного наблюдателя за фрагментом. Также зарегистрируйте вновь добавленного наблюдателя жизненного цикла, чтобы отслеживать жизненный цикл представления фрагмента.

Листинг 25.21. Регистрация наблюдателя жизненного цикла представления (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        thumbnailDownloader =
            ThumbnailDownloader(responseHandler
) {
            ...
        }
        lifecycle.addObserver(thumbnailDownload
er.fragmentLifecycleObserver)
    }

    override fun onCreateView(
            inflater: LayoutInflater,
            container: ViewGroup?,
            savedInstanceState: Bundle?
    ): View {
        viewLifecycleOwner.lifecycle.addObserve
r(
            thumbnailDownloader.viewLifecycleOb
server
```

```
            )
            ...
        }
        ...
    }
```

Жизненный цикл представления можно безопасно наблюдать в функции `Fragment.onCreateView(...)`, до тех пор пока в конце функции `onCreateView(...)` не будет возвращено ненулевое представление. Смотрите упражнение в конце этой главы, чтобы узнать больше о наблюдении за `viewLifecycleOwner`, что дает большую гибкость в настройке наблюдения жизненного цикла представления.

Наконец, отменим регистрацию `thumbnailDownloader` в качестве наблюдателя за жизненным циклом фрагмента представления в функции `Fragment.onDestroyView()`. Обновим существующий код, который отменяет регистрацию `thumbnailDownloader` в качестве наблюдателя за жизненным циклом фрагментов.

Листинг 25.22. Отказ от регистрации наблюдателя жизненного цикла представления (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
    override fun onDestroyView() {
        super.onDestroyView()
        viewLifecycleOwner.lifecycle.removeObse
rver(
            thumbnailDownloader.viewLifecycleOb
server
        )
```

```
        }

        override fun onDestroy() {
            super.onDestroy()
            lifecycle.removeObserver(
                thumbnailDownloader.fragmentLifecyc
leObserver
            )
        }
        ...
    }
```

Как и в случае с наблюдателями жизненного цикла фрагмента, вам не нужно явно отменять регистрацию фрагмента `viewLifecycleOwner.lifecycle` наблюдателей. Регистр представления жизненного цикла фрагмента, в котором отслеживаются все наблюдатели жизненного цикла представления, опустошается при уничтожении представления фрагмента. Однако, как и раньше, мы предпочитаем явно снимать с подписки наблюдателей, а не полагаться на сбор мусора.

На этом мы завершаем главу. Запустите приложение PhotoGallery. Покрутите экран, чтобы увидеть динамическую загрузку изображений (рис. 25.10).

Рис. 25.10. Изображения на экране

PhotoGallery выполняет свою цель — выводит изображения с Flickr. В следующих нескольких главах мы добавим дополнительные функциональные возможности, такие как поиск фотографий и открытие каждой страницы Flickr в веб-браузере.

Сохраненные фрагменты

По умолчанию свойство `retainInstance` фрагмента имеет значение `false`. Это означает, что он не сохраняется, а уничтожается и воссоздается при повороте вместе с activity, на которой он расположен. Вызов функции

`setRetainInstance(true)` сохраняет фрагмент. При сохранении фрагмента он не удаляется. Вместо этого он сохраняется и передается новой activity в целости и сохранности.

При сохранении фрагмента можно рассчитывать на то, что все его экземпляры будут иметь одинаковые значения и всегда будут на месте.

Поворот и сохранение фрагментов

Давайте подробнее рассмотрим, как работают сохраненные фрагменты. Сохраненные фрагменты работают за счет того, что представление фрагмента можно уничтожить и воссоздать заново, не уничтожая сам фрагмент.

При смене конфигурации `FragmentManager` сначала уничтожает представления фрагментов в списке. Представления фрагментов всегда уничтожаются и воссоздаются при изменении конфигурации по тем же причинам, что и представления activity: если у вас есть новая конфигурация, то вам могут понадобиться новые ресурсы. Если они есть, вы восстанавливаете представление с нуля.

Далее `FragmentManager` проверяет свойство `remainInstance` каждого фрагмента. Если оно ложное, что по умолчанию так и есть, то `FragmentManager` уничтожает экземпляр фрагмента. Фрагмент и его представление будут воссозданы новым `FragmentManager` новой activity «с другой стороны» (рис. 25.11).

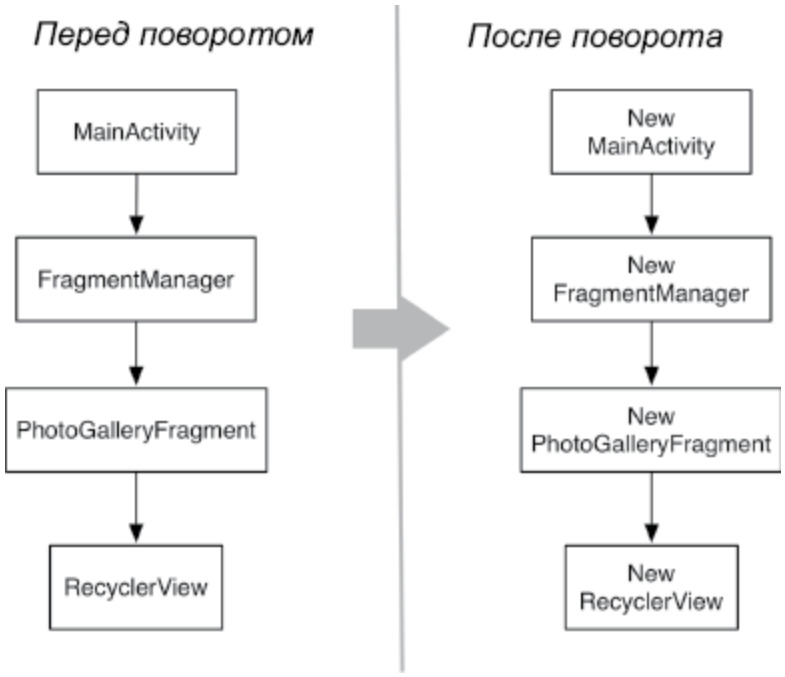


Рис. 25.11. Поворот по умолчанию с фрагментом пользовательского интерфейса

С другой стороны, если `retainInstance` имеет значение `true`, то представление фрагмента уничтожается, а сам фрагмент — нет. Когда создается новая activity, новый `FragmentManager` находит сохраненный фрагмент и воссоздает его (рис. 25.12).

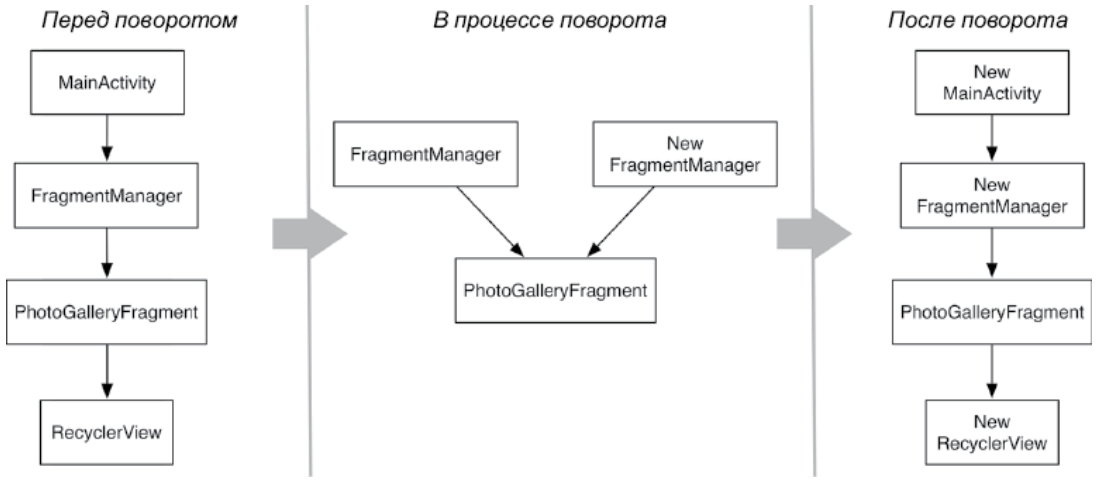


Рис. 25.12. Поворот с сохранением фрагмента пользовательского интерфейса

Сохраненный фрагмент не уничтожается, а *отделяется* от завершаемой activity. Это переводит фрагмент в сохраненное состояние. Фрагмент все еще существует, но он не размещен внутри activity (рис. 25.13).

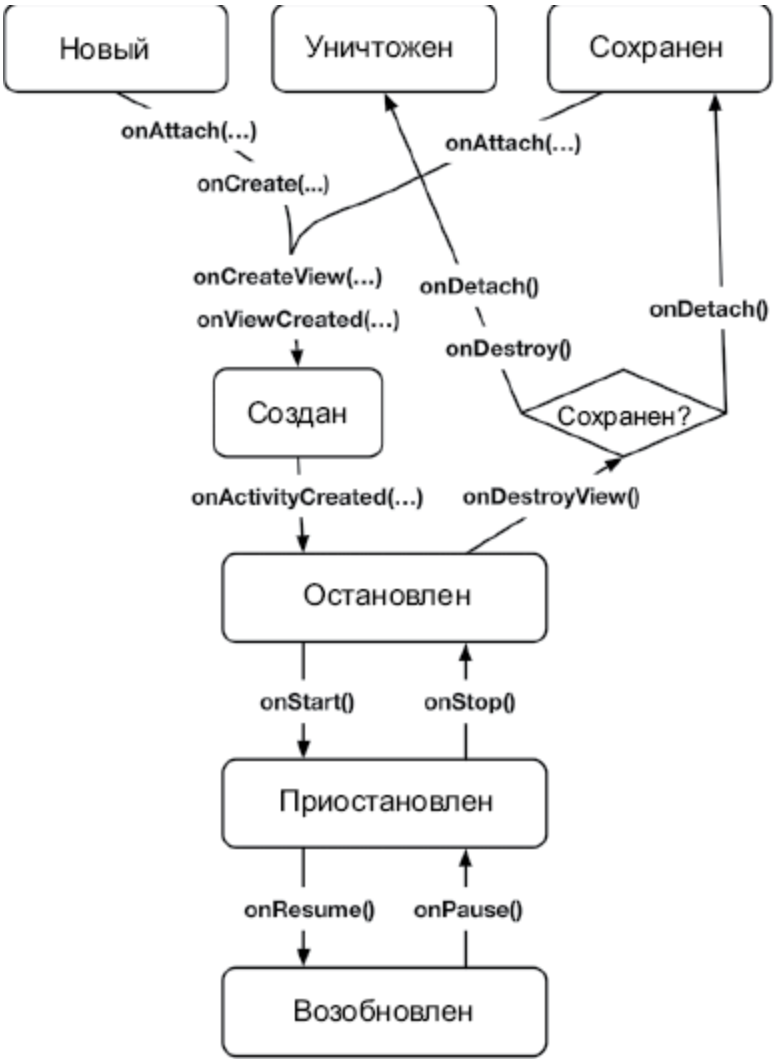


Рис. 25.13. Жизненный цикл фрагмента

Сохраненное состояние вводится при выполнении двух условий:

- была вызвана функция `setRetainInstance(true)`;

- хост-activity уничтожается для изменения конфигурации (обычно это поворот).

Фрагмент находится в сохраненном состоянии в течение очень короткого промежутка времени — между отсоединением от старой activity и повторным присоединением к новой activity, которая создается сразу же.

Выполнять ли сохранение

Сохраненные фрагменты: изящное решение, да? Да! Похоже, такой подход решает все наши проблемы с уничтожением при повороте. Когда меняется конфигурация устройства, вы получаете наиболее подходящие ресурсы, создавая совершенно новое представление, и у вас есть простой способ сохранить данные и объекты.

Но почему мы тогда не сохраняем каждый фрагмент или почему фрагменты не сохраняются по умолчанию? В целом мы не рекомендуем использовать этот механизм (если нет реальной необходимости) по нескольким причинам.

Первая причина заключается в том, что сохраняемые фрагменты сложнее, чем не сохраненные. Когда с ними что-то не так, то докопаться до сути того, что пошло не так, сложно. Программы всегда сложнее, чем вы хотите, так что если вы сможете обойтись без этих осложнений, то так и надо.

Вторая причина заключается в том, что фрагменты, обрабатывающие поворот с использованием сохраненного состояния экземпляра, обрабатывают все ситуации жизненного цикла, но сохраненные фрагменты обрабатывают только тот случай, когда activity уничтожается для изменения конфигурации. Если же ваша activity уничтожена из-за того, что операционной системе требуется память, то уничтожаются и

все сохраненные вами фрагменты, что может означать потерю некоторых данных.

Третья причина заключается в том, что класс `ViewModel` в большинстве случаев устраняет необходимость сохранения фрагментов. Используйте `ViewModel` вместо сохраненного фрагмента для сохранения состояния UI при изменении конфигурации. `ViewModel` дает те же преимущества без недостатков введения более сложного жизненного цикла для фрагмента.

Для любознательных: решение задачи загрузки изображений

Эта книга призвана научить вас пользоваться инструментами из стандартной библиотеки Android. Но если вы не боитесь пользоваться сторонними библиотеками, то знайте, что они способны сэкономить уйму времени в различных ситуациях, в том числе и при загрузке изображений, реализованной в PhotoGallery.

Откровенно говоря, решение, представленное в этой главе, далеко не идеально. Когда вам приходится решать проблемы кэширования, преобразований и повышения быстродействия, естественно спросить себя: а не занимался ли кто-нибудь решением этой задачи до вас? Ответ: конечно, занимался. Существует несколько готовых библиотек, решающих задачу загрузки изображений. В своих коммерческих приложениях мы предпочитаем использовать для загрузки изображений библиотеку Picasso (**square.github.io/picasso/**).

С Picasso все, что мы делали в этой главе, делается в одной строке:

```
private class PhotoHolder(private val
itemImageView: ImageView)
    : RecyclerView.ViewHolder(itemView) {
        ...
              fun bindGalleryItem(galleryItem:
GalleryItem) {
            Picasso.get()
                .load(galleryItem.url)
                .placeholder(R.drawable.bill_up
_close)
                .into(itemImageView)
        }
        ...
}
```

Динамичный интерфейс требует задания контекста конструкцией `get()`. URL-адрес загружаемого изображения задается конструкцией `load(String)`, а объект `ImageView` для загрузки результатов — конструкцией `into(ImageView)`. Также поддерживается много других настраиваемых параметров: например, назначение изображения, которое должно выводиться до полной загрузки запрошенного изображения (`placeholder(Int)` или `placeholder(drawable)`).

В функции `PhotoAdapter.onBindViewHolder(...)` существующий код заменяется сквозным вызовом новой функции `bindGalleryItem(...)`.

Picasso выполняет всю работу `ThumbnailDownloader` (вместе с обратным вызовом `ThumbnailDownloader.ThumbnailDownloadListener<T>`) и всю работу `FlickrFetchr`, связанную с графикой. Это означает, что при использовании Picasso класс

`ThumbnailDownloader` можно удалить (хотя класс `FlickrFetchr` еще понадобится для загрузки данных JSON). Кроме упрощения кода, Picasso поддерживает такие дополнительные возможности, как преобразование изображений и организация кэширования с минимальными усилиями с вашей стороны.

Библиотека Picasso добавляется в проект как зависимость на панели структуры проекта; это же делалось для других зависимостей (например, `RecyclerView`).

К недостаткам Picasso можно отнести намеренное ограничение функциональности для сокращения ее размера. В результате Picasso не может загружать и отображать анимированные изображения. Если у вас возникнет такая необходимость, проверьте библиотеку Google Glide или библиотеку Facebook Fresco. Из этих двух библиотек Glide занимает меньше памяти, а Fresco обладает лучшим быстродействием.

Для любознательных: StrictMode

Есть некоторые вещи, которые просто не следует делать в приложениях Android — ошибки, которые напрямую приводят к сбоям и дефектам безопасности. Например, выполнение сетевого запроса в главном потоке при плохом состоянии сети с большой вероятностью приведет к ошибке ANR.

Вместо того чтобы спокойно разрешить выполнение сетевого запроса в главном потоке приложения, Android выдает исключение `NetworkOnMainThread` и сохраняет сообщение в журнале. Это обусловлено действием режима `StrictMode`: он замечает вашу ошибку и любезно сообщает вам о ней. Режим `StrictMode` был создан для того, чтобы помочь разработчику в

обнаружении таких и многих других ошибок и дефектов безопасности в коде.

Сетевые операции в главном потоке запрещаются без какой-либо дополнительной конфигурации. Также StrictMode поможет обнаруживать другие ошибки, способные ухудшить быстродействие приложения. Чтобы включить все рекомендованные политики StrictMode, вызовите функцию `StrictMode.enableDefaults()` (**developer.android.com/reference/android/os/StrictMode.html#enableDefaults()**).

После вызова функции `StrictMode.enableDefaults()` на панели **Logcat** будет выводиться информация о следующих нарушениях:

- сетевые операции в главном потоке;
- операции чтения и записи на диск в главном потоке;
- продолжающееся существование activity за пределами их естественного жизненного цикла (так называемая «утечка activity»);
- незакрытые курсоры баз данных SQLite;
- передача незашифрованного текста в сетевом трафике без защиты SSL/TLS.

Классы `ThreadPolicy.Builder` и `VmPolicy.Builder` предоставляют средства для расширенного управления тем, что должно происходить при нарушении политик: выдача исключения, появление диалогового окна или просто сохранение в журнале информации, уведомляющей вас о происходящем.

Упражнение. Наблюдение LiveData у LifecycleOwner

PhotoGalleryFragment вызывает функцию viewLifecycleOwner.lifecycle.observe(...) во Fragment.onCreateView(...). Это прекрасно работает до тех пор, пока представление, которое вы возвращаете из функции onCreateView(...), не является null.

Непосредственно наблюдать жизненный цикл представления в функции Fragment.onCreate(...) нельзя, так как жизненный цикл действителен только с момента вызова функции Fragment.onCreateView(...) до момента вызова функции Fragment.onDestroyView(...). Тем не менее вы можете наблюдать за жизненным циклом представления фрагмента с помощью вызова функции Fragment.getViewLifecycleOwnerLiveData(), которая возвращает LiveData<LifecycleOwner>. viewLifecycleOwner фрагмента публикуется на LiveData после возврата ненулевого представления из Fragment.onCreateView(...) и устанавливается в null после вызова Fragment.onDestroyView(...).

Для этой задачи надо научить код наблюдать за viewLifecycleOwner фрагмента через LiveData<LifecycleOwner>, возвращаемый из функции Fragment.getViewLifecycleOwnerLiveData(). Вновь добавленное отношение наблюдателя должно быть привязано к жизни экземпляра фрагмента.

Когда публикуется значение, отличное от null, наблюдайте за жизненным циклом вида фрагмента, доступного через viewLifecycleOwner.lifeecycle.

Упражнение. Больше информации о жизненном цикле ThumbnailDownloader

`LiveData` — это компонент жизненного цикла, который автоматически удаляется в качестве наблюдателя в зависимости от событий, происходящих с владельцем его жизненного цикла. Измените `ThumbnailDownloader`, чтобы он удалял себя в качестве наблюдателя автоматически, когда жизненный цикл, который он наблюдает, испускает событие `ON_DESTROY`. Для этого загрузчику эскизов понадобится ссылка на каждый из наблюдаемых им жизненных циклов.

Тот факт, что `ThumbnailDownloader` наблюдает и жизненный цикл фрагмента, и жизненный цикл представления фрагмента, плотно соединяет `ThumbnailDownloader` с фрагментом. Существует много разных способов, чтобы убрать это, но здесь мы просим вас сделать один маленький шаг: научить `ThumbnailDownloader` наблюдать только за жизненным циклом фрагментов. Затем при вызове функции `Fragment.onDestroyView()` уберите фрагмент из очереди. Таким образом, ваш загрузчик можно будет использовать более легко и с activity.

Упражнение. Предварительная загрузка и кэширование

Пользователи понимают, что не все происходит мгновенно (или по крайней мере большинство пользователей). Но, несмотря на это, программисты стремятся к совершенству.

Для достижения моментального отклика в большинстве реальных приложений приведенный код расширяется в двух направлениях: добавление кэширования и предварительная загрузка изображений.

Кэш в нашем приложении представляет собой место для хранения определенного количества объектов `Bitmap`, чтобы они оставались в памяти даже после завершения использования. Объем кэша ограничен, поэтому вам понадобится стратегия выбора сохраняемых объектов при исчерпании свободного места. Многие кэши используют стратегию LRU (Least Recently Used): при нехватке свободного места из кэша удаляется элемент, который дольше всего не использовался.

Библиотека поддержки Android содержит класс `LruCache`, реализующий стратегию LRU. В первом упражнении используйте `LruCache` для добавления простейшего кэширования `ThumbnailDownloader`. Каждый раз, когда для URL-адреса загружается объект `Bitmap`, вы помещаете его в кэш. Затем, когда требуется загрузить новое изображение, вы сначала проверяете содержимое кэша и смотрите, нет ли его в кэше.

После того как в программе будет создан кэш, он может использоваться для предварительной загрузки, то есть загрузки данных в кэш еще до того, как они фактически потребуются программе. Тем самым предотвращается задержка для загрузки объектов `Bitmap` до их вывода.

Качественно реализовать предварительную загрузку непросто, но она существенно меняет восприятие приложения пользователем. Во втором, более сложном упражнении для каждого выводимого элемента `GalleryItem` выполните предварительную загрузку 10 предшествующих и 10 следующих элементов `GalleryItem`.

26. Поиск: SearchView и SharedPreferences

Следующим шагом в работе над приложением PhotoGallery станет поиск фотографий на Flickr. В этой главе вы узнаете, как правильно интегрировать поиск в приложение с использованием `SearchView`. `SearchView` — класс *представления действия* (*action view*) — представления, которое может быть встроено прямо в `Toolbar`.

Пользователь нажимает на `SearchView`, вводит запрос и отправляет его. Flickr проводит поиск введенной строки с использованием поискового API и заполняет `RecyclerView` полученными результатами (рис. 26.1). Отправленная строка запроса сохраняется в файловой системе. Это означает, что последний запрос пользователя «переживет» перезапуск приложения и даже устройства.

Рис. 26.1. Внешний вид приложения

Поиск во Flickr

Начнем с того, что нужно сделать на стороне Flickr. Для выполнения поиска во Flickr следует вызвать метод `flickr.photos.search`. Вот как выглядит запрос GET для поиска текста «cat»:

```
https://api.flickr.com/services/rest/?
method=flickr.photos.search
&api_key=xxx&format=json&nojsoncallback=1&extra
s=url_s&safe_search=1&text=cat
```

В методе используется значение `flickr.photos.search`. Параметр `text` определяет условия поиска («`cat`» в данном случае). Присваивание свойству `safesearch` значения `1` позволит отфильтровать из поиска оскорбительные результаты.

Некоторые из пар «параметр — значение», такие как `format=json`, являются постоянными как для `flickr.photo.search`, так и для `flickr.interestingness.getList`. Нам нужно абстрагировать эти пары общих параметров-значений в перехватчик. Перехватчик делает то, что следует из названия, — он перехватывает запрос или ответ и позволяет вам манипулировать содержимым или совершать какие-то действия до завершения запроса или ответа.

Создайте новый класс `Interceptor` с именем `PhotoInterceptor` в папке `api`. Переопределите функцию `intercept(chain)` на доступ к запросу, добавьте в нее пары «параметр — значение» и перезапишите исходный URL-адрес. (Не забудьте использовать ваш API-ключ, который вы создали в главе 24, вместо значения *вашApiКлюч*. Вы можете скопировать его из файла `api/FlickrApi.kt`.)

Листинг 26.1. Добавление перехватчика для вставки URL-констант (api/PhotoInterceptor.kt)

```
private const val API_KEY = "вашApiКлюч"

class PhotoInterceptor : Interceptor {

            override fun intercept(chain: Interceptor.Chain): Response {
                val originalRequest: Request = chain.request()

                    val newUrl: HttpUrl = originalRequest.url().newBuilder()
                    .addQueryParameter("api_key", API_KEY)
                    .addQueryParameter("format", "json")
                    .addQueryParameter("nojsoncallback", "1")
                    .addQueryParameter("extras", "url_s")
                    .addQueryParameter("safesearch", "1")
                    .build()

                    val newRequest: Request = originalRequest.newBuilder()
                    .url(newUrl)
                    .build()

        return chain.proceed(newRequest)
    }
```

```
}
```

Android Studio предлагает несколько вариантов при импорте `Request` и `Response`. Выберите вариант `okhttp3` в обоих случаях.

Здесь мы вызываем функцию `chain.request()` для доступа к исходному запросу. Функция `originalRequest.url()` извлекает исходный URL из запроса, а затем используется `HttpUrl.Builder` для добавления параметров запроса.

`HttpUrl.Builder` создает новый запрос на основе оригинального запроса и заменяет исходный URL на новый. Наконец, мы вызываем функцию `chain.continue(newRequest)` для создания ответа. Если вы не вызывали `chain.continue(...)`, сетевой запрос не будет выполнен.

Теперь откройте `FlickrFetchr.kt` и добавьте перехватчик в конфигурацию `Retrofit`.

Листинг 26.2. Добавление перехватчика в конфигурацию Retrofit (FlickrFetchr.kt)

```
class FlickrFetchr {

    private val flickrApi: FlickrApi

    init {
        val client = OkHttpClient.Builder()
            .addInterceptor(PhotoInterceptor())
            .build()
```

```
                    val retrofit: Retrofit =
Retrofit.Builder()
            .baseUrl("https://api.flickr.com/")
            .addConverterFactory(GsonConverterF
actory.create())
            .client(client)
            .build()

                                    flickrApi =
retrofit.create(FlickrApi::class.java)
    }
    ...
}
```

Мы создаем экземпляр `OkHttpClient` и добавляем `PhotoInterceptor` в качестве перехватчика. Затем мы устанавливаем только что настроенный клиент на экземпляр `Retrofit`, который заменяет клиент по умполчанию. Теперь `Retrofit` будет использовать предоставленный вами клиент и, в свою очередь, применять функцию `PhotoInterceptor.intercept(...)` к любым запросам.

Вам больше не нужен URL-адрес `flickr.interestingness.getList`, указанный во `FlickrApi`. Замените это на функцию `searchPhotos()` для определения поискового запроса в настройках Retrofit API.

Листинг 26.3. Добавление функции поиска во FlickrApi (api/FlickrApi.kt)

```
interface FlickrApi {
```

```
                                    @GET("services/rest/?
method=flickr.interestingness.getList" +
        "&api_key=вашApiКлюч" +
                    "&format=json" +
                    "&nojsoncallback=1" +
                    "&extras=url_s")

                                    @GET("services/rest?
method=flickr.interestingness.getList")
    fun fetchPhotos(): Call<FlickrResponse>

    @GET
        fun fetchUrlBytes(@Url url: String):
Call<ResponseBody>

                                    @GET("services/rest?
method=flickr.photos.search")
        fun searchPhotos(@Query("text") query:
String): Call<FlickrResponse>
}
```

Аннотация `@Query` позволяет динамически добавлять к URL параметры запроса. В данном случае мы добавляем параметр запроса `text`. Значение, присваиваемое параметру, зависит от аргумента, переданного в `searchPhotos(String)`. Например, вызов `searchPhotos("robot")` добавит в URL приписку `text=robot`.

Добавьте во `FlickrFetchr` функцию поиска, чтобы обернуть вновь добавленную функцию `FlickrApi.searchPhotos(String)`. Вытяните код, который асинхронно выполняет объект `Call`, и оберните результат в `LiveData` во вспомогательную функцию.

Листинг 26.4. Добавление функции поиска во FlickrFetchr (FlickrFetchr.kt)

```
class FlickrFetchr {

    private val flickrApi: FlickrApi

    init {
        ...
    }

                        fun fetchPhotos():
LiveData<List<GalleryItem>> {
                                    return
fetchPhotoMetadata(flickrApi.fetchPhotos())
    }

          fun searchPhotos(query: String):
LiveData<List<GalleryItem>> {
                                    return
fetchPhotoMetadata(flickrApi.searchPhotos(query
))
    }

                        ~~fun fetchPhotos():~~
~~LiveData<List<GalleryItem>> {~~
                                private fun
fetchPhotoMetadata(flickrRequest:
Call<FlickrResponse>)
          : LiveData<List<GalleryItem>> {
                        val responseLiveData:
MutableLiveData<List<GalleryItem>>          =
MutableLiveData()
```

```
                                ~~val flickrRequest:~~
~~Call<FlickrResponse> = flickrApi.fetchPhotos()~~

                flickrRequest.enqueue(object :
Callback<FlickrResponse> {
            ...
        })

        return responseLiveData
    }
    ...
}
```

Наконец, измените код `PhotoGalleryViewModel`, чтобы начать поиск по Flickr. На данный момент поисковое слово будет жестко задано — `planets`. Жесткое кодирование запроса позволяет вам протестировать ваш новый код поиска, даже если вы еще не предусмотрели способ ввода запроса через пользовательский интерфейс.

Листинг 26.5. Запуск поискового запроса (PhotoGalleryViewModel.kt)

```
class PhotoGalleryViewModel : ViewModel() {

                    val     galleryItemLiveData:
LiveData<List<GalleryItem>>

    init {
                        galleryItemLiveData =
FlickrFetchr().  ~~fetchPhotos()~~searchPhotos("p
lanets")
```

```
        }
    }
```

URL-адрес поискового запроса отличается от того, который вы использовали для запроса интересных фотографий, но формат данных, возвращаемых JSON, остается прежним. Это хорошо, потому что означает, что вы можете использовать ту же самую конфигурацию Gson и тот же код сопоставления модели, который вы уже написали.

Запустите PhotoGallery, чтобы убедиться, что ваш поисковый запрос работает правильно. Надеюсь, вы увидите пару крутых фоток Земли. (Если вы не получите результатов, очевидно связанных с планетами, это не значит, что ваш запрос не работает. Попробуйте другой поисковый термин — например, `bicycle` или `llama` — и запустите ваше приложение еще раз, чтобы убедиться, что вы действительно видите результаты поиска.)

Использование SearchView

Итак, мы обеспечили поддержку поиска во `FlickrFetchr`; теперь необходимо предоставить пользователю средства для ввода запроса и запуска поиска. Для этого мы добавим виджет `SearchView`.

`SearchView` является представлением действия, которое может быть размещено на панели приложения. `SearchView` позволяет вынести весь поисковый интерфейс приложения на панель приложения.

Создайте XML-файл меню `res/menu/fragment_photo_gallery.xml` для `PhotoGalleryFragment` (если забыли, как что делается, вернитесь к главе 14). В этом файле будут определяться

элементы, которые должны располагаться на панели приложения.

Листинг 26.6. Добавление XML-файла меню (res/menu/fragment_photo_gallery.xml)

```
<menu
xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">

    <item android:id="@+id/menu_item_search"
          android:title="@string/search"
          app:actionViewClass="androidx.appcompat.widget.SearchView"
          app:showAsAction="ifRoom" />

    <item android:id="@+id/menu_item_clear"
          android:title="@string/clear_search"
          app:showAsAction="never" />
</menu>
```

Вы получите пару сообщений об ошибках в XML, в которых будет сказано, что строки, используемые в атрибутах `android:title`, еще не определены. Пока не обращайте на них внимания — вскоре мы решим эту проблему.

Первое определение элемента в листинге 26.6 приказывает вывести на панели приложения виджет `SearchView`, для чего атрибуту `app:actionViewClass` присваивается значение `androidx.appcompat.widget.SearchView`. (Обратите внимание на использование пространства имен `app` для

атрибутов showAsAction и actionViewClass. Если вы забыли, для чего оно используется, вернитесь к главе 14.)

Второй элемент в листинге 26.6 добавляет действие Clear Search. Это действие всегда будет отображаться в дополнительном меню, потому что атрибуту app:showAsAction присвоено значение never. Позднее мы настроим это действие так, чтобы при нажатии хранимый запрос пользователя стирался с диска, но пока о нем можно забыть.

Пришло время разобраться с ошибками в разметке меню. Откройте файл res/values/strings.xml и добавьте недостающие строки.

Листинг 26.7. Добавление строк для поиска (res/values/strings.xml)

```
<resources>
    ...
    <string name="search">Search</string>
    <string name="clear_search">Clear Search</string>

</resources>
```

Наконец, откройте файл PhotoGalleryFragment.kt. Добавьте в функцию onCreate(...) вызов setHasOptionsMenu(true), чтобы зарегистрировать фрагмент для получения обратных вызовов меню. Переопределите OnCreateOptionsMenu(...) и заполните созданный файл XML с определением меню. Элементы, перечисленные в разметке меню, добавляются на панель приложения.

Листинг 26.8. Переопределение onCreateOptionsMenu(...) (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

        retainInstance = true
        setHasOptionsMenu(true)
        ...
    }
    ...
    override fun onDestroy() {
        ...
    }

        override fun onCreateOptionsMenu(menu:
Menu, inflater: MenuInflater) {
                super.onCreateOptionsMenu(menu,
inflater)
        inflater.inflate(R.menu.fragment_photo_
gallery, menu)
    }
    ...
}
```

Запустите приложение PhotoGallery и посмотрите, как выглядит SearchView. При нажатии значка поиска открывается представление с текстовым полем для ввода запроса (рис. 26.2).

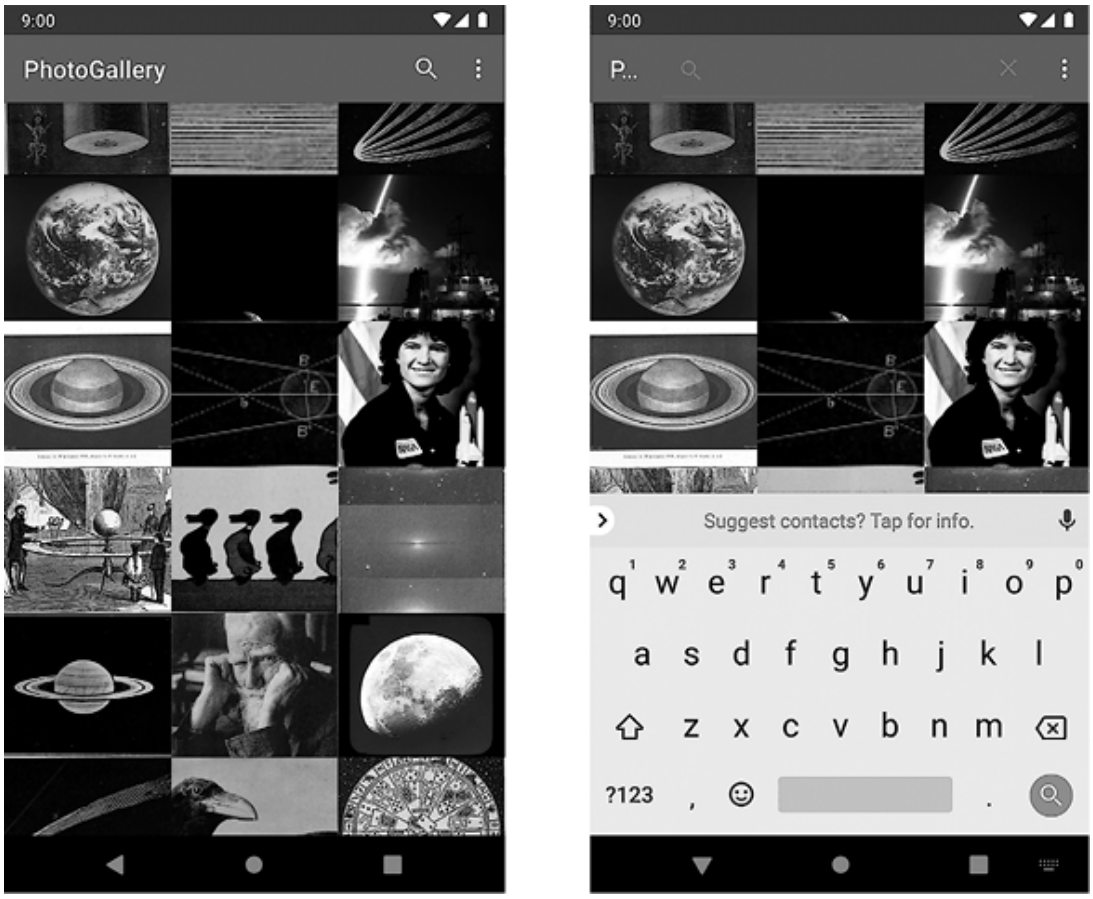


Рис. 26.2. SearchView в свернутом и развернутом состоянии

Когда панель `SearchView` открыта, справа появляется значок ×. Однократное нажатие стирает введенный текст, а повторное нажатие сворачивает `SearchView` в значок.

Если вы попытаетесь отправить запрос, ничего не произойдет. Для беспокойства нет причин — сейчас мы научим `SearchView` делать что-то полезное.

Реакция SearchView на действия пользователя

Когда пользователь отправляет запрос, ваше приложение должно выполнять поиск по веб-сервису Flickr и обновлять изображения, выводимые пользователю в результатах поиска. Добавим в `PhotoGalleryViewModel` код, позволяющий

следить за последним поисковым запросом пользователя и обновлять результаты поиска при изменении запроса.

Листинг 26.9. Хранение последнего запроса в PhotoGalleryViewModel (PhotoGalleryViewModel.kt)

```
class PhotoGalleryViewModel : ViewModel() {

                    val     galleryItemLiveData:
LiveData<List<GalleryItem>>

    private val flickrFetchr = FlickrFetchr()
            private   val   mutableSearchTerm   =
MutableLiveData<String>()

    init {
        mutableSearchTerm.value = "planets"

                          galleryItemLiveData   =
    FlickrFetchr().searchPhotos("planets")
                Transformations.switchMap(mutab
leSearchTerm) { searchTerm ->
                    flickrFetchr.searchPhotos(s
earchTerm)
                }
    }

    fun fetchPhotos(query: String = "") {
        mutableSearchTerm.value = query
    }
}
```

Каждый раз, когда искомое значение изменяется, список элементов галереи тоже должен обновляться и выводить новые результаты. И поисковый элемент, и список объектов галереи обернуты в `LiveData`, поэтому мы будем использовать функцию `Transformations.switchMap(trigger:LiveData<X>,transformFunction:Function<X,LiveData<Y>>)` для реализации этого отношения. (Если вам нужно вспомнить о преобразованиях `LiveData`, см. главу 12.)

Также мы помещаем экземпляр `FlickrFetchr` в свойство. Таким образом, вы создаете экземпляр `FlickrFetchr` всего единожды за время жизни экземпляра `ViewModel`. Повторное использование одного и того же экземпляра `FlickrFetchr` позволяет избежать ненужных затрат, связанных с повторным созданием экземпляра `Retrofit` и `FlickrApi` каждый раз, когда приложение выполняет поиск. Для пользователя это означает ускорение работы приложения.

Теперь научим `PhotoGalleryFragment` изменять значение поискового запроса `PhotoGalleryViewModel` всякий раз, когда пользователь отправляет новый запрос через `SearchView`. К счастью, интерфейс `SearchView.OnQueryTextListener` позволяет получать обратный вызов при отправке запроса.

Измените функцию `OnCreateOptionsMenu(...)`, чтобы добавить `SearchView.OnQueryTextListener` в `SearchView`.

Листинг 26.10. Регистрация событий SearchView.OnQueryTextListener (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
```

```
        override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
                super.onCreateOptionsMenu(menu, inflater)
        inflater.inflate(R.menu.fragment_photo_gallery, menu)

                val searchItem: MenuItem = menu.findItem(R.id.menu_item_search)
         val searchView = searchItem.actionView as SearchView

        searchView.apply {

                setOnQueryTextListener(object : SearchView.OnQueryTextListener {
                                override fun onQueryTextSubmit(queryText: String): Boolean {
                                        Log.d(TAG, "QueryTextSubmit: $queryText")
                                photoGalleryViewModel.fetchPhotos(queryText)
                                return true
                        }

                                override fun onQueryTextChange(queryText: String): Boolean {
                                        Log.d(TAG, "QueryTextChange: $queryText")
                                return false
                        }
                })
```

```
            }
        }
        ...
    }
```

При импорте SearchView выберите строку androidx.appcompat.widget.SearchView из представленных вариантов.

В функции OnCreateOptionsMenu(...) мы берем элемент MenuItem, представляющий собой окно поиска, и сохраняем его в searchItem. Затем мы извлекаем объект SearchView из searchItem с помощью функции getActionView().

Имея ссылку на SearchView, вы можете установить SearchView.OnQueryTextListener, используя функцию setOnQueryTextListener(...). В реализации SearchView.OnQueryTextListener необходимо переопределить две функции: onQueryTextSubmit(String) и onQueryTextChange(String).

Обратный вызов onQueryTextChange(String) выполняется при изменении текста в текстовом поле SearchView. Это означает, что функция вызывается каждый раз, когда меняется один символ. В этом приложении вы не будете ничего делать внутри обратного вызова, кроме как регистрировать входную строку и возвращать false. Возврат false говорит системе о том, что ваше переопределение обратного вызова не обрабатывает изменение текста. Это приказывает системе выполнять действие SearchView по умолчанию (которое заключается в отображении соответствующих предложений, если таковые имеются).

Обратный вызов onQueryTextSubmit(String) выполняется тогда, когда пользователь отправляет запрос. Запрос, заданный пользователем, передается в качестве

входного. Возврат `true` означает, что поисковый запрос был обработан. Этот обратный вызов заставляет `PhotoGalleryViewModel` инициировать загрузку фотографии для поискового запроса.

Запустите приложение и отправьте запрос. Вы должны увидеть в журнале записи, отражающие выполнение функций обратного вызова `SearchView.OnQueryTextListener`. Сами изображения должны измениться в зависимости от запроса (рис. 26.3).

Рис. 26.3. SearchView в процессе работы

Обратите внимание, что если вы используете аппаратную клавиатуру для отправки поискового запроса на эмуляторе (в

отличие от экранной клавиатуры эмулятора), поиск будет выполняться дважды. Это связано с тем, что в `SearchView` есть небольшая ошибка. Вы можете игнорировать такое поведение, потому что это побочный эффект от использования эмулятора, который не будет влиять на ваше приложение, когда оно работает на реальном устройстве Android.

Простое сохранение с помощью SharedPreferences

В нашем приложении в любой момент времени существует только один активный запрос. `PhotoGalleryViewModel` сохраняет запрос на длительность жизненного цикла фрагмента пользователя. Но он также должен сохраняться (запоминаться приложением) между перезапусками, даже если пользователь отключит устройство.

Для этого строка запроса будет записываться в *хранилище общих настроек* (shared preferences). Каждый раз, когда пользователь отправляет запрос, приложение первым делом записывает запрос в общие настройки (заменяя запрос, который хранился до этого). При первом запуске приложения строка запроса читается из общих настроек и используется для поиска во Flickr.

Хранилище общих настроек представляет собой файлы в файловой системе, для чтения и редактирования которых используется класс `SharedPreferences`. Экземпляр `SharedPreferences` работает как хранилище пар «ключ — значение», имеющее много общего с `Bundle`, но с возможностью долгосрочного хранения. Ключами являются строки, а значениями — атомарные типы данных. При ближайшем рассмотрении выясняется, что файлы содержат простую разметку XML, но благодаря классу `SharedPreferences` на эту подробность реализации можно не

обращать внимания. Файлы общих настроек хранятся в песочнице вашего приложения, поэтому в них не следует хранить конфиденциальную информацию (например, пароли).

Создайте файл `QueryPreferences.kt`, который будет предоставлять удобный интерфейс для чтения/записи запроса в хранилище общих настроек.

Листинг 26.11. Добавление класса для работы с хранимым запросом (QueryPreferences.kt)

```
private const val PREF_SEARCH_QUERY = "searchQuery"

object QueryPreferences {

        fun getStoredQuery(context: Context): String {
                                val prefs = PreferenceManager.getDefaultSharedPreferences(context)
                                        return prefs.getString(PREF_SEARCH_QUERY, "")!!
    }

    fun setStoredQuery(context: Context, query: String) {
        PreferenceManager.getDefaultSharedPreferences(context)
                .edit()
                  .putString(PREF_SEARCH_QUERY, query)
                .apply()
```

```
        }
    }
```

Вашему приложению нужен только один экземпляр `QueryPreferences`, который может использоваться всеми другими компонентами. Из-за этого мы используем ключевое слово `object` (вместо `class`), чтобы указать, что `QueryPreferences` — это синглтон. Так тем более в приложении будет создан всего один экземпляр. Это также позволяет вам получить доступ к функциям в объекте, используя синтаксис `ClassName.functionName(...)`, как вы увидите вскоре.

Значение `PREF_SEARCH_QUERY` используется в качестве ключа для хранения запроса. Этот ключ применяется во всех операциях чтения или записи запроса.

Функция `PreferencesManager.getDefaultSharedPreferences(Context)` возвращает экземпляр с именем по умолчанию и приватными разрешениями (так что предпочтения доступны только изнутри вашего приложения). Чтобы получить конкретный экземпляр `SharedPreferences`, вы можете использовать функцию `Context.getSharedPreferences(String,Int)`. Однако на практике нас в большей степени волнует, чтобы экземпляр был общим для всего приложения.

Функция `getStoredQuery(Context)` возвращает значение запроса, хранящееся в общих настройках. Для этого функция сначала получает объект `SharedPreferences` по умолчанию для заданного контекста. (Так как `QueryPreferences` не имеет собственного контекста, вызывающий компонент должен передать свой контекст как входной параметр.)

Получение ранее сохраненного значения сводится к простому вызову SharedPreferences.getString(...), SharedPreferences.getInt(...) или другой функции, соответствующей типу данных. Второй параметр SharedPreferences.getString (String,String) определяет возвращаемое значение по умолчанию, которое должно возвращаться при отсутствии записи с ключом PREF_SEARCH_QUERY.

Возвращаемый тип SharedPreferences.getString(...) определен как тип String, допускающий значение null, так как компилятор не может гарантировать, что значение, связанное с PREF_SEARCH_QUERY, существует и что оно не является null. Но вы знаете, что такой ситуации не бывает, поэтому предоставляете пустую строку в качестве значения по умолчанию в тех случаях, когда функция setStoredQuery(context:Context,query:String) еще не вызывалась. Поэтому здесь безопасно использовать оператор ненулевого утверждения (!!) без блока try/catch.

Функция setStoredQuery(Context) записывает запрос в хранилище общих настроек для заданного контекста. В приведенном выше коде QueryPreferences вызов SharedPreferences.edit() используется для получения экземпляра SharedPreferences.Editor. Этот класс используется для сохранения значений в SharedPreferences. Он позволяет объединять изменения в транзакции по аналогии с тем, как это делается во FragmentTransaction. Множественные изменения могут быть сгруппированы в одну операцию записи в хранилище.

После того как все изменения будут внесены, вызовите apply() для объекта Editor, чтобы эти изменения стали видимыми для всех пользователей файла SharedPreferences. Функция apply() вносит изменения в память немедленно, а

непосредственная запись в файл осуществляется в фоновом потоке.

`QueryPreferences` предоставляет всю функциональность долгосрочного хранения данных для PhotoGallery.

Теперь, когда у вас есть механизм простого сохранения и выборки последнего запроса пользователя, можно переходить к обновлению `PhotoGalleryFragment` для чтения и записи запроса из общих настроек. Мы считываем запрос при первом создании `ViewModel` и используем значение для инициализации `mutableSearchTerm`. Запишем запрос при каждом изменении `mutableSearchTerm`.

Листинг 26.12. Сохранение запроса в общих настройках (PhotoGalleryViewModel.kt)

```
    ~~class PhotoGalleryViewModel : ViewModel() {~~
class PhotoGalleryViewModel(private val app:
Application) : AndroidViewModel(app) {
    ...
    init {
                    mutableSearchTerm.value =
    ~~"planets"~~QueryPreferences.getStoredQuery(ap
p)
        ...
    }

    fun fetchPhotos(query: String = "") {
            QueryPreferences.setStoredQuery(app,
query)
        mutableSearchTerm.value = query
    }
}
```

Вашему ViewModel нужен контекст для использования функций QueryPreferences. Изменение родительского класса PhotoGalleryViewModel с ViewModel на AndroidViewModel предоставляет PhotoGalleryViewModel доступ к контексту приложения. Хранить ссылку на контекст приложения в PhotoGalleryViewModel безопасно, потому что контекст приложения переживает PhotoGalleryViewModel.

Далее очистите сохраненный запрос (установите его равным ""), когда пользователь выберет элемент «Clear Search» в меню.

Листинг 26.13. Очистка сохраненного запроса (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
        override fun onCreateOptionsMenu(menu:
Menu, inflater: MenuInflater) {
        ...
    }

      override fun onOptionsItemSelected(item:
MenuItem): Boolean {
        return when (item.itemId) {
            R.id.menu_item_clear -> {
                photoGalleryViewModel.fetchPhot
os("")
                true
            }
                                    else   ->
super.onOptionsItemSelected(item)
        }
    }
```

```
    ...
}
```

И последнее, но не по значению — научим `PhotoGalleryViewModel` подгружать интересные фотографии при очистке запроса вместо поиска по пустому поисковому запросу.

Листинг 26.14. Получение интересных фотографий, когда запрос пустой (PhotoGalleryViewModel.kt)

```
class PhotoGalleryViewModel(private val app: Application) : AndroidViewModel(app) {
    ...
    init {

                    mutableSearchTerm.value = QueryPreferences.getStoredQuery(app)

        galleryItemLiveData = Transformations.switchMap(mutableSearchTerm) { searchTerm ->
                **if (searchTerm.isBlank()) {**
                    **flickrFetchr.fetchPhotos()**
                **} else {**
                    flickrFetchr.searchPhotos(searchTerm)
                **}**
            }
    }
    ...
```

```
}
```

Поиск теперь должен работать прекрасно. Запустите приложение PhotoGallery и попробуйте найти что-нибудь забавное, например, «unicycle». Посмотрите на результаты. Затем полностью выйдите из приложения с помощью кнопки «Назад». Можно даже перезагрузить телефон. При перезапуске приложения вы должны увидеть результаты поиска по тому же поисковому запросу.

Последний штрих

Осталось внести последнее усовершенствование: заполнить текстовое поле поиска сохраненным запросом, когда пользователь нажимает кнопку поиска для открытия `SearchView`.

Сначала добавим вычисляемое свойство `PhotoGalleryViewModel` для открытия поисковой фразы, которая отображается в интерфейсе.

Листинг 26.15. Открытие поисковой фразы из PhotoGalleryViewModel (PhotoGalleryViewModel.kt)

```
class PhotoGalleryViewModel(private val app: Application) : AndroidViewModel(app) {
    ...
        private val mutableSearchTerm = MutableLiveData<String>()

    val searchTerm: String
        get() = mutableSearchTerm.value ?: ""
```

```
        init {
            ...
        }
        ...
    }
```

Функция View.OnClickListener.onClick() в SearchView вызывается при нажатии этой кнопки. Подключите функцию обратного вызова и задайте текст запроса SearchView при раскрытии представления.

Листинг 26.16. Предварительное заполнение SearchView (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
        override fun onCreateOptionsMenu(menu:
Menu, inflater: MenuInflater) {
        ...
        searchView.apply {

                setOnQueryTextListener(object :
SearchView.OnQueryTextListener {
                ...
            })

            setOnSearchClickListener {
                searchView.setQuery(photoGaller
yViewModel.searchTerm, false)
            }
        }
    }
```

```
        ...
    }
```

Запустите приложение и поэкспериментируйте с отправкой нескольких поисковых запросов. Проверьте, как работает последнее добавленное усовершенствование. Конечно, можно сделать еще один штрих...

Редактирование SharedPreferences с помощью Android KTX

В Jetpack входит базовая библиотека Android KTX. Android KTX — это набор расширений Kotlin, который позволяет использовать возможности языка Kotlin при наборе кода, основанного на Android API, написанного на Java. Использование Android KTX не изменяет функциональность существующих Java API, но помогает сохранить идиомы кода Kotlin.

На момент написания этой книги в Android KTX входят только расширения для подмножества Android Java API. Список доступных модулей расширений приведен в документации Android KTX (**developer.android.com/kotlin/ktx**). В основной модуль Android KTX входит расширение для редактирования общих настроек.

Настройте `QueryPreferences` для использования Android KTX. Во-первых, добавьте базовую зависимость Android KTX в файл `build.gradle` вашего модуля приложения.

Листинг 26.17. Добавление зависимости core-ktx (app/build.gradle)

```
dependencies {
```

```
        implementation fileTree(dir: 'libs',
include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-
stdlib-jdk7:$kotlin_version"
        implementation 'androidx.core:core-
ktx:1.0.0'
    ...
}
```

Далее обновите функцию `QueryPreferences.setStoredQuery(...)` для использования Android KTX.

Листинг 26.18. Использование Android KTX (QueryPreferences.kt)

```
object QueryPreferences {
    ...
    fun setStoredQuery(context: Context, query:
String) {
        PreferenceManager.getDefaultSharedPrefe
rences(context)
            .edit { ~~()~~ {
                .putString(PREF_SEARCH_QUERY,
query)
                ~~.apply()~~
        }
    }
}
```

Функция `SharedPreferences.edit(commit:Boolean=false,action:Editor.()->Unit)` является расширением в `core-ktx`. Если

ваш код содержит ошибку, проверьте, добавлен ли оператор импорта в `androidx.core.content.edit`.

В аргументе лямбда, который вы передаете в `SharedPreferences.Editor`, вы указываете желаемые правки. Здесь вы пишете одну строку, используя функцию `android.content.SharedPreferences.Editor.putString(...)`.

Явный вызов `SharedPreferences.Editor.apply()` удаляется, потому что по умолчанию функция расширения `edit` вызывает функцию `apply()`. Вы можете переопределить это поведение по умолчанию, передав `true` в качестве аргумента функции расширения редактирования. Это приводит к вызову функции редактирования `SharedPreferences.Editor.commit()` вместо `SharedPreferences.Editor.apply()`.

Запустите приложение и убедитесь, что функциональность PhotoGallery не пострадала. Ваше приложение должно вести себя так же, как и до того, как вы сделали изменения в Android KTX. Но теперь код ваших общих настроек сделан в духе Kotlin (это чтобы фанаты Kotlin порадовались).

Упражнение. Еще одно усовершенствование

Возможно, вы заметили, что при отправке запроса перед обновлением `RecyclerView` происходит небольшая задержка. В этом упражнении вам предлагается субъективно ускорить отклик приложения на отправленный запрос: сразу же после отправки запроса скройте виртуальную клавиатуру и сверните `SearchView`.

Очистите содержимое `RecyclerView` и отобразите индикатор загрузки (с неопределенным состоянием) сразу же

после отправки запроса. Закройте индикатор загрузки сразу же после завершения загрузки данных JSON. Иначе говоря, индикатор загрузки не должен отображаться, когда код перейдет к загрузке отдельных изображений.

27. Библиотека WorkManager

Теперь наше приложение PhotoGallery умеет загружать интересные изображения с Flickr, находить изображения по заданному пользователем поисковому запросу и запоминать запрос, когда пользователь покидает приложение. В этой главе мы добавим функционал, позволяющий определить, есть ли на ресурсе новые фотографии, которые пользователь еще не видел.

Эта работа будет выполняться в фоновом режиме, даже если пользователь не работает с приложением в данный конкретный момент. Если появятся новые фотографии, приложение покажет уведомление, предлагающее пользователю вернуться в приложение и просмотреть новый контент.

Инструменты из библиотеки компонентов архитектуры Jetpack WorkManager позволят периодически проверять Flickr на наличие новых фотографий. Мы создадим для этого класс `Worker`, а затем запланируем его работу. Когда новые фотографии будут найдены, пользователь получит уведомление с помощью `NotificationManager`,

Создание класса Worker

В класс `Worker` мы поместим выполняемую в фоновом режиме работу. Как только работник будет готов, мы создадим `WorkRequest`, который сообщит системе, когда работа должна быть выполнена по плану.

Прежде чем мы сможем добавить работника, вам сначала нужно добавить соответствующую зависимость в ваш файл `app/build.gradle`.

Листинг 27.1. Добавление зависимости WorkManager (app/build.gradle)

```
dependencies {
    ...
                                    implementation
'androidx.recyclerview:recyclerview:1.0.0'
      implementation "android.arch.work:work-
runtime:1.0.1"
    ...
}
```

Синхронизируйте проект для подгрузки зависимостей.

Когда новая библиотека загружена, перейдем к созданию работника. Создайте новый класс `PollWorker`, являющийся расширением базового класса `Worker`. Вашему `PollWorker` понадобятся два параметра: `Context` и `WorkerParameters`. Оба они будут переданы конструктору суперкласса. А пока мы переопределим функцию `doWork()` и выведем сообщение в консоли.

Листинг 27.2. Создание работника (PollWorker.kt)

```
private const val TAG = "PollWorker"

class PollWorker(val context: Context,
workerParams: WorkerParameters)
    : Worker(context, workerParams) {

    override fun doWork(): Result {
        Log.i(TAG, "Work request triggered")
        return Result.success()
    }
```

```
}
```

Функция `doWork()` вызывается из фонового потока, поэтому вы можете выполнять в ней любые долгосрочные задачи. Возвращаемые значения функции указывают на состояние работы. В данном случае возвращается информация об успехе, так как функция просто выводит сообщение в консоль.

Функция `doWork()` может вернуть ошибку, если работа не может быть завершена. В этом случае рабочий запрос выполняться не будет. Он также может вернуть результат повторной попытки, если ошибка была временной и вы хотите, чтобы в будущем работа снова запустилась.

`PollWorker` знает только то, как *выполнить* фоновую работу. Вам нужен еще один компонент для планирования работы.

Планирование работы

Чтобы запланировать выполнение `Worker`, нам нужен запрос `WorkRequest`. Сам класс `WorkRequest` является абстрактным, поэтому нам придется использовать один из его подклассов в зависимости от типа работы, которую вам нужно выполнить. Если у вас есть что-то, что нужно выполнить только один раз, используйте `OneTimeWorkRequest`. Если ваша работа должна выполняться периодически, используйте `PeriodicWorkRequest`.

Пока что мы будем использовать `OneTimeWorkRequest`. Это позволит вам убедиться, что ваш `PollWorker` работает правильно, а также узнать больше о создании и контроле запросов. Позже мы обновим приложение для использования `PeriodicWorkRequest`.

Откройте файл PhotoGalleryFragment.kt, чтобы создать рабочий запрос и запланировать его выполнение.

Листинг 27.3. Планирование WorkRequest (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        lifecycle.addObserver(thumbnailDownload
er.fragmentLifecycleObserver)

        val workRequest = OneTimeWorkRequest
            .Builder(PollWorker::class.java)
            .build()
        WorkManager.getInstance()
            .enqueue(workRequest)
    }
    ...
}
```

В OneTimeWorkRequest используется конструктор для создания экземпляра. Мы передаем класс Worker конструктору, который будет запущен в рабочем запросе. Как только ваш рабочий запрос будет готов, вам нужно запланировать его с помощью класса WorkManager. Мы вызываем функцию getInstance() для доступа к WorkManager, затем функцию enqueue(...) с рабочим запросом в качестве параметра. Это запланирует выполнение вашего рабочего запроса с учетом

типа запроса и любых ограничений, которые вы в него добавляете.

Запустите ваше приложение и выполните поиск по запросу `PollWorker` на панели **Logcat**. После запуска вы увидите сообщение в журнале (рис. 27.1).

Часто запланированная работа, которую вы хотите выполнить в фоновом режиме, связана с сетью. Например, вы запрашиваете новую информацию, которую пользователь еще не видел, или загружаете обновления с локальной базы данных, чтобы сохранить их на удаленном сервере. Даже если задача важная, вам следует убедиться, что вы не используете лишний трафик. Лучшее всего выполнять эти запросы тогда, когда устройство подключено к сети без измерения трафика.

Рис. 27.1. Содержимое журнала

Вы можете использовать класс `Constraints`, чтобы добавить эту информацию в ваши рабочие запросы. С помощью этого класса вы можете требовать выполнения определенных условий перед выполнением задачи. Одним из таких условий может быть требование определенного типа сети. Может также требоваться достаточное количество заряда аккумулятора или установка устройства на зарядку.

Отредактируйте код `OneTimeWorkRequest` в `PhotoGalleryFragment`, добавив в запрос ограничения.

Листинг 27.4. Добавление рабочих ограничений (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        lifecycle.addObserver(thumbnailDownload
er.fragmentLifecycleObserver)

        val constraints = Constraints.Builder()
            .setRequiredNetworkType(NetworkType
.UNMETERED)
            .build()
        val workRequest = OneTimeWorkRequest
            .Builder(PollWorker::class.java)
            .setConstraints(constraints)
            .build()
        WorkManager.getInstance()
            .enqueue(workRequest)
    }
    ...
}
```

Подобно рабочему запросу, в объекте `Constraints` для настройки нового экземпляра используется конструктор. В этом случае мы указываем, что для выполнения рабочего запроса устройство должно быть подключено к сети без ограничений трафика.

Для проверки этих функций вам нужно будет смоделировать различные типы сетей на вашем эмуляторе. По умолчанию

эмулятор подключается к моделируемой сети Wi-Fi. Поскольку Wi-Fi является неизмеряемой сетью, если вы сейчас запустите приложение с установленными ограничениями, вы должны увидеть в журнале сообщение от `PollWorker`.

Чтобы убедиться, что рабочий запрос не будет выполнен, когда устройство находится в измеряемой сети, вам необходимо изменить настройки сети вашего эмулятора. Выйдите из PhotoGallery и потяните уведомление, чтобы открыть меню быстрых настроек устройства. Проведите еще раз, чтобы открыть более быструю и понятную версию быстрых настроек (рис. 27.2). Обычно не имеет значения, какую версию вы используете, но в некоторых старых версиях Android для доступа к сетевым настройкам требуется двойное нажатие.

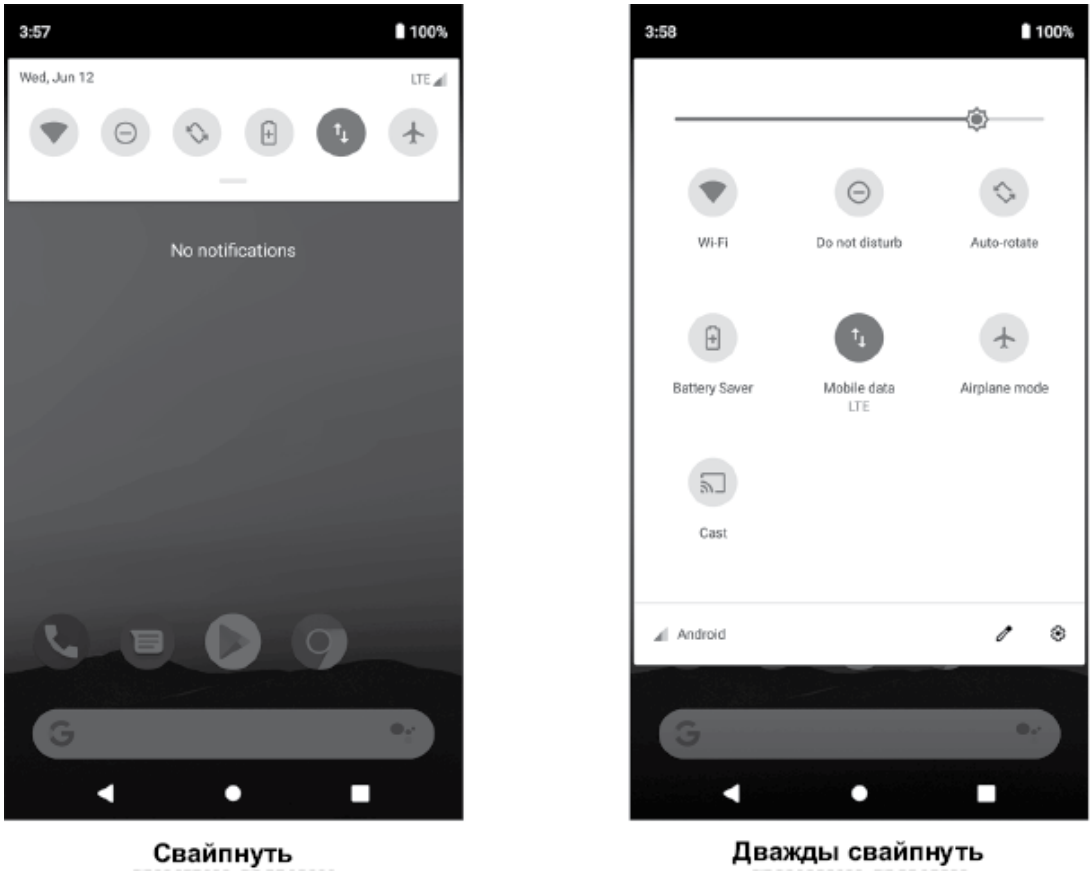


Рис. 27.2. Доступ к быстрым настройкам

Нажмите на значок Wi-Fi в меню быстрой настройки, чтобы отключить сеть Wi-Fi и заставить эмулятор использовать сотовую сеть.

Когда сеть Wi-Fi отключена, запустите PhotoGallery и убедитесь, что сообщение от `PollWorker` не выводится. Прежде чем двигаться дальше, снова включите сеть Wi-Fi в меню быстрых настроек.

Проверка на наличие новых фотографий

Когда работник начал выполняться, вы можете добавить логику проверки новых фотографий. Для этого нужно несколько вещей. Сначала вам понадобится способ сохранить идентификатор самой последней фотографии, которую видел пользователь, затем нужно будет обновить рабочий класс, чтобы получить новые фотографии и сравнить сохраненный идентификатор с самым новым, имеющимся на сервере. Ваш работник также берет на себя выбор типа запроса, если существует поисковый запрос.

Первое, что мы сделаем, это отредактируем файл `QueryPreferences`, чтобы сохранить и получить последний идентификатор фотографии из общих настроек.

Листинг 27.5. Сохранение идентификатора последней фотографии (QueryPreference.kt)

```
private const val PREF_SEARCH_QUERY =
"searchQuery"
private const val PREF_LAST_RESULT_ID =
"lastResultId"

object QueryPreferences {
```

```
    ...
    fun setStoredQuery(context: Context, query:
String) {
        ...
    }

        fun getLastResultId(context: Context):
String {
                                        return
PreferenceManager.getDefaultSharedPreferences(c
ontext)
                .getString(PREF_LAST_RESULT_ID,
"")!!
    }

        fun setLastResultId(context: Context,
lastResultId: String) {
        PreferenceManager.getDefaultSharedPrefe
rences(context).edit {
                putString(PREF_LAST_RESULT_ID,
lastResultId)
        }
    }
}
```

Как мы уже сделали в главе 26, используем оператор ненулевого утверждения (`!!`) при получении последнего `ID` от экземпляра `SharedPreferences` в функции `getLastResultId(...)`, поскольку из `getString(PREF_LAST_RESULT_ID,"")` возвращается строка, которая не может иметь значения null.

Имея возможность сохранять идентификаторы фотографий, вы можете перейти к загрузке новых фотографий. Вам нужно будет изменить код во `FlickrFetchr`, чтобы ваш `PollWorker` выполнял синхронные веб-запросы. Функции `fetchPhotos()` и `searchPhotos()`выполняют запросы асинхронно и доставляют результаты, используя `LiveData`. Так как `PollWorker` выполняется в фоновом потоке, вам не нужно заставлять `FlickrFetchr` делать запрос. Научим `FlickrFetchr` предоставлять объекты `Call` для вашего работника.

Листинг 27.6. Предоставление объектов Call (FlickrFetchr.kt)

```
class FlickrFetchr {
    ...
                    fun     fetchPhotosRequest():
Call<FlickrResponse> {
        return flickrApi.fetchPhotos()
    }

                          fun      fetchPhotos():
LiveData<List<GalleryItem>> {
                                                  return
fetchPhotoMetadata(flickrApi.fetchPhotos())
                                                  return
fetchPhotoMetadata(fetchPhotosRequest())
    }

       fun  searchPhotosRequest(query:  String):
Call<FlickrResponse> {
        return flickrApi.searchPhotos(query)
    }
```

```
            fun searchPhotos(query: String):
LiveData<List<GalleryItem>> {
                                                ~~return
~~fetchPhotoMetadata(flickrApi.searchPhotos(query
~~)~~
                                                return
fetchPhotoMetadata(searchPhotosRequest(query))
    }
    ...
}
```

Открытие объекта `Call` во `FlickrFetchr` позволяет добавлять запросы в `PollWorker`. Вам нужно будет определить, какой запрос `PollWorker` должен выполнить, в зависимости от того, есть ли сохраненный поисковый запрос. Получив самые свежие фотографии, вы проверите, совпадает ли самый последний идентификатор фотографии с тем, который вы сохранили. Если они не совпадают, то пользователю выводится уведомление.

Начнем с извлечения текущего поискового запроса и последнего идентификатора фотографии из `QueryPreferences`. Если поискового запроса нет, мы загружаем обычные фотографии. Если поисковый запрос есть, выполняем его. В целях безопасности мы будем использовать пустой список, если одному из запросов не удается вернуть какие-либо фотографии. Удалите оператор журнала, поскольку он больше не нужен.

Листинг 27.7. Получение последних фотографий (PollWorker.kt)

```
class PollWorker(val context: Context,
workerParameters: WorkerParameters)
```

```
    : Worker(context, workerParameters) {

    override fun doWork(): Result {
                    ~~Log.i(TAG, "Work request
triggered")~~
                                val query =
QueryPreferences.getStoredQuery(context)
                            val lastResultId =
QueryPreferences.getLastResultId(context)
            val items: List<GalleryItem> = if
(query.isEmpty()) {
            FlickrFetchr().fetchPhotosRequest()
                .execute()
                .body()
                ?.photos
                ?.galleryItems
        } else {
            FlickrFetchr().searchPhotosRequest(
query)
                .execute()
                .body()
                ?.photos
                ?.galleryItems
        } ?: emptyList()
        return Result.success()
    }
}
```

Добавляем возврат из функции `doWork()`, если список элементов пуст. В противном случае проверим наличие новых фотографий, сравнивая идентификатор первого элемента в списке со свойством `lastResultId`. Добавим вывод в консоль,

чтобы посмотреть результаты. Кроме того, обновим идентификатор последнего результата в QueryPreferences, если нашли новый результат.

Листинг 27.8. Проверка, нет ли новых фотографий (PollWorker.kt)

```
class PollWorker(val context: Context, workerParameters: WorkerParameters)
    : Worker(context, workerParameters) {

    override fun doWork(): Result {
                                val query = QueryPreferences.getStoredQuery(context)
                            val lastResultId = QueryPreferences.getLastResultId(context)
            val items: List<GalleryItem> = if (query.isEmpty()) {
                ...
            } else {
                ...
            }

            if (items.isEmpty()) {
                return Result.success()
            }

            val resultId = items.first().id
            if (resultId == lastResultId) {
                    Log.i(TAG, "Got an old result: $resultId")
            } else {
```

```
                Log.i(TAG, "Got a new result:
$resultId")
            QueryPreferences.setLastResultId(co
ntext, resultId)
        }

        return Result.success()
    }
}
```

Запустите ваше приложение на устройстве или эмуляторе. При первом его запуске в `QueryPreferences` не будет сохраняться последний идентификатор результата, поэтому вы должны увидеть сообщение журнала, указывающее на то, что `PollWorker` нашел новый результат. Если вы быстро запустите приложение снова, то увидите, что ваш работник обнаружил такой же идентификатор (рис. 27.3) (Убедитесь, что для вашего **Logcat** установлено значение `No Filters` (Без фильтров), а не `Show only selected application` (Показывать только выбранное приложение).

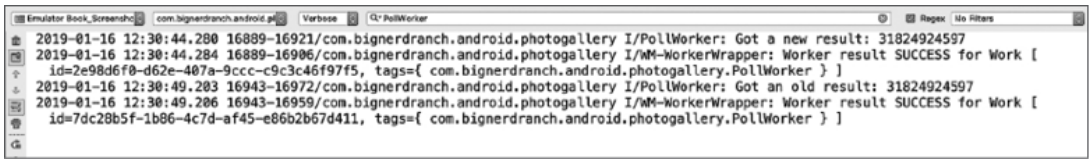


Рис. 27.3. Поиск новых и старых результатов

Вывод уведомления пользователю

Теперь работник проверяет наличие новых фотографий в фоновом режиме, но пользователь ничего об этом не знает. Когда PhotoGallery находит новые фотографии, которые

пользователь еще не видел, пользователю будет предложено открыть приложение и посмотреть новый контент.

Когда вашему приложению нужно что-то сообщить пользователю, всегда используются *уведомления*. Уведомления — это элементы, которые появляются на панели уведомлений, которую пользователь открывает, перетаскивая ее из верхней части экрана.

Прежде чем вы сможете создавать уведомления на устройствах Android под управлением Android Oreo (уровень API 26) и выше, нужно будет создать объект `Channel`. `Channel` классифицирует уведомления и дает пользователю возможность управлять настройками уведомлений. Вместо того чтобы иметь возможность отключить уведомления для всего приложения, пользователь может отключить определенные категории уведомлений в самом приложении. Пользователь также может настроить снижение громкости, вибрацию и другие параметры уведомлений по каналам.

Предположим, вы хотите, чтобы приложение PhotoGallery выделяло три категории уведомлений при получении новых изображений с милыми животными: новые изображения с котятами, новые изображения со щенятами и все миленькое сразу (для всех очаровательных изображений животных, независимо от вида). Нужно создать три канала, по одному для каждой категории уведомлений, и пользователь сможет настраивать их независимо друг от друга (рис. 27.4).

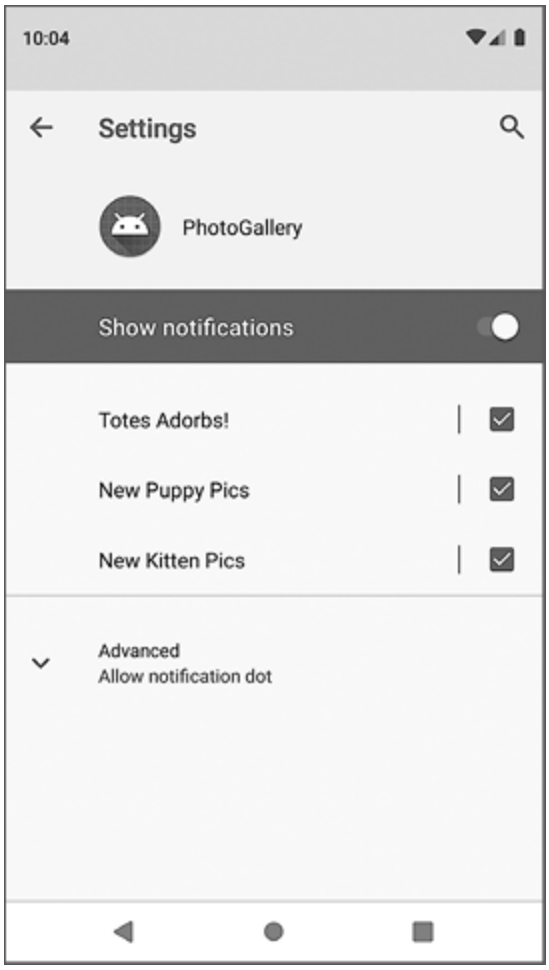


Рис. 27.4. Детальная настройка уведомлений для каналов

Ваше приложение должно создать хотя бы один канал для поддержки Android Oreo и версий выше. Официально предела количества каналов, которые может создать приложение, не существует. Но вы должны быть разумны и сохранять это число маленьким и осмысленным для пользователя. Помните, что наша цель — позволить пользователю настраивать уведомления в вашем приложении. Добавление слишком большого количества каналов может в конечном итоге запутать пользователя и ухудшить его восприятие.

Создайте класс `PhotoGalleryApplication`. Расширьте `Application` и переопределите функцию `Application.onCreate()`, добавив возможность создать и

добавить канал, если устройство работает под управлением Android Oreo или выше.

Листинг 27.9. Создание канала уведомлений (PhotoGalleryApplication.kt)

```
const val NOTIFICATION_CHANNEL_ID = "flickr_poll"

class PhotoGalleryApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            val name = getString(R.string.notification_channel_name)
            val importance = NotificationManager.IMPORTANCE_DEFAULT
            val channel = NotificationChannel(NOTIFICATION_CHANNEL_ID, name, importance)
            val notificationManager: NotificationManager =
                getSystemService(NotificationManager::class.java)
            notificationManager.createNotificationChannel(channel)
        }
    }
}
```

Название канала — это выводимая пользователю строка, которая отображается на экране настроек уведомлений приложения (как показано на рис. 27.4). Добавьте строковый ресурс в файл `res/values/strings.xml` для хранения имени канала. Добавьте другие строки, необходимые для вашего уведомления.

Листинг 27.10. Добавление строк (res/values/strings.xml)

```
<resources>
            <string   name="clear_search">Clear
Search</string>
                                          <string
name="notification_channel_name">FlickrFetchr</
string>
         <string  name="new_pictures_title">New
PhotoGallery Pictures</string>
     <string name="new_pictures_text">You have
new pictures in PhotoGallery.</string>
</resources>
```

Затем измените код манифеста, чтобы он указывал на новый класс приложения, который вы создали.

Листинг 27.11. Обновление тега приложения в манифесте (manifests/AndroidManifest.xml)

```
<manifest ... >
...
  <application
      android:name=".PhotoGalleryApplication"
      android:allowBackup="true"
      ... >
```

```
    </application>
</manifest>
```

Чтобы опубликовать уведомление, вам нужно создать объект `Notification`. Такие объекты создаются с помощью конструктора, так же как `AlertDialog`, который мы использовали в главе 13. Как минимум у объекта `Notification` должны быть:

- *значок* для отображения в строке состояния;
- *представление* для отображения самого уведомления на панели;
- `PendingIntent` для запуска, когда пользователь нажимает на уведомление на панели;
- `NotificationChannel`, позволяющий применить стилизацию и предоставить пользователю контроль над уведомлением.

Также нужно добавить в уведомление текст тикера. Этот текст не отображается, когда появляется уведомление, но вместо этого отправляется в службы доступа, чтобы программы вроде экранных считывателей могли уведомлять пользователей с нарушением зрения.

После того как вы создали объект `Notification`, вы можете разместить его, вызвав функцию `notify(Int,Notification)` в системной службе `NotificationManager`. `Int` — это идентификатор уведомления из вашего приложения.

Сначала вам необходимо добавить код, показанный в листинге 27.12. Откройте файл `PhotoGalleryActivity.kt` и

добавьте функцию newIntent(Context). Эта функция вернет экземпляр Intent, который может быть использован для запуска PhotoGalleryActivity. (В конце концов, PollWorker будет вызывать функцию PhotoGalleryActivity.newIntent(...), обертывать полученный интент в PendingIntent и устанавливать уведомление.)

Листинг 27.12. Добавление newIntent(...) в PhotoGalleryActivity (PhotoGalleryActivity.kt)

```
class PhotoGalleryActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
    }

    companion object {
        fun newIntent(context: Context): Intent {
            return Intent(context, PhotoGalleryActivity::class.java)
        }
    }
}
```

Теперь надо научить PollWorker уведомлять пользователя о том, что новый результат готов, создав объект Notification и вызвав функцию NotificationManager.notify(Int,Notification).

Листинг 27.13. Добавление уведомления (PollWorker.kt)

```
class PollWorker(val context: Context, workerParameters: WorkerParameters)
    : Worker(context, workerParameters) {

    override fun doWork(): Result {
        ...
        val resultId = items.first().id
        if (resultId == lastResultId) {
                Log.i(TAG, "Got an old result: $resultId")
        } else {
                 Log.i(TAG, "Got a new result: $resultId")
            QueryPreferences.setLastResultId(context, resultId)

                                    val intent = PhotoGalleryActivity.newIntent(context)
                            val pendingIntent = PendingIntent.getActivity(context, 0, intent, 0)

            val resources = context.resources
                                val notification = NotificationCompat
                                    .Builder(context, NOTIFICATION_CHANNEL_ID)
                .setTicker(resources.getString(R.string.new_pictures_title))
                .setSmallIcon(android.R.drawable.ic_menu_report_image)
```

```
                .setContentTitle(resources.getS
tring(R.string.new_pictures_title))
                .setContentText(resources.getSt
ring(R.string.new_pictures_text))
                .setContentIntent(pendingIntent
)
                .setAutoCancel(true)
                .build()

            val notificationManager =
NotificationManagerCompat.from(context)
                notificationManager.notify(0,
notification)
        }
        return Result.success()
    }
}
```

Давайте пройдемся по этому коду сверху вниз.

Мы используем класс `NotificationCompat` для работы с уведомлениями как на устройствах до Oreo, так и после Oreo. `NotificationCompat.Builder` принимает ID канала и использует его для установки параметра канала уведомления, если пользователь запустил приложение на Oreo или выше. Если у пользователя запущена более ранняя версия Android, `NotificationCompat.Builder` игнорирует канал. (Обратите внимание, что идентификатор канала, который вы передаете здесь, происходит от константы `NOTIFICATION_CHANNEL_ID`, которую вы добавили в `PhotoGalleryApplication`.)

В листинге 27.9 перед созданием канала вы проверили версию SDK, потому что AppCompat API для создания канала не существует. Здесь это не нужно делать, потому что

NotificationCompat в AppCompat выполняет всю работу по проверке версии сборки, сохраняя ваш код в чистоте и красоте. Это одна из причин, по которой нужно использовать версию AppCompat Android API, когда это доступно.

Текст уведомления и значок мы настраиваем с помощью функций setTicker (CharSequence) и setSmallIcon(Int). (Ресурс значка, который вы используете, входит во фреймворк Android, обозначается классификатором имен пакетов androidвandroid.R.drawable.ic_menu_report_image, так что вам не нужно переносить изображение значка в папку ресурса.)

После этого мы настроим внешний вид Notification на панели. Можно создать полностью пользовательский внешний вид, но проще всего использовать стандартный вид уведомления, в котором есть иконка, заголовок и текстовая область. Иконка берется из функции setSmallIcon(Int). Для установки заголовка и текста вызываются функции setContentTitle(CharSequence) и setContentText(CharSequence) соответственно.

Затем мы задаем, что произойдет, когда пользователь нажмет на уведомление. Это делается с помощью объекта PendingIntent. Объект PendingIntent, который вы передаете в setContentIntent(PendingIntent), будет уничтожен, когда пользователь нажмет на уведомление на панели. Вызов функции setAutoCancel(true) немного корректирует это поведение: уведомление также будет удалено из ящика уведомлений, когда пользователь нажмет на него.

Наконец, мы получаем экземпляр NotificationManager из текущего контекста (NotificationManagerCompat.from) и вызываем функцию NotificationManager.notify(...) для размещения вашего уведомления.

Целый параметр, который вы передаете в функцию `notify(...)`, является идентификатором вашего уведомления. Он должен быть уникальным во всем вашем приложении, но может быть использован повторно. Одно уведомление заменит другое тем же самым идентификатором, который все еще находится в ящике уведомлений. Если нет существующего уведомления с идентификатором, система покажет новое уведомление. Именно так вы реализуете индикатор выполнения или другие динамические визуализации.

И все. Запустите ваше приложение, и в конце концов в строке состояния появится значок уведомления (рис. 27.5) (лучше очистить поисковые запросы, чтобы ускорить процесс).

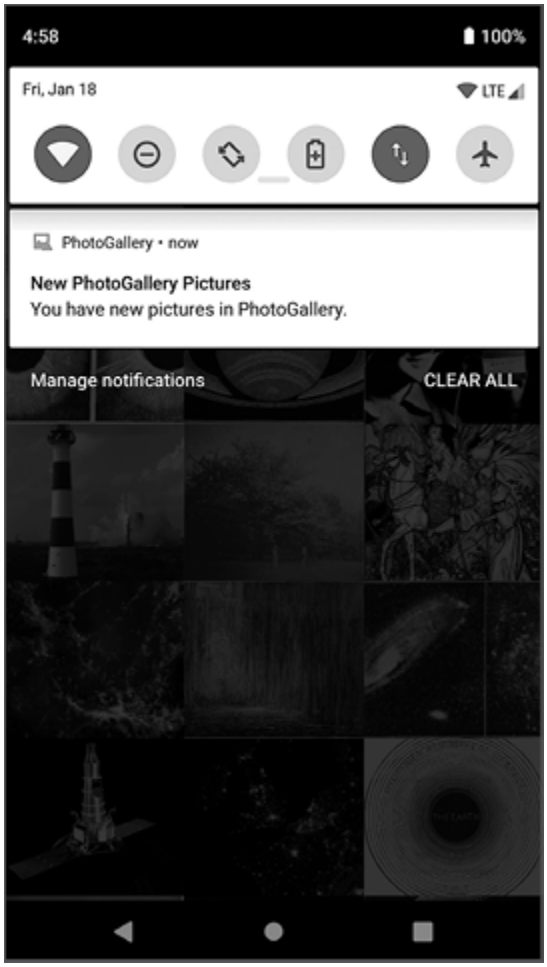


Рис. 27.5. Уведомление о новых фотографиях

Управление опросом от пользователя

Некоторые пользователи могут не захотеть, чтобы приложение работало в фоновом режиме. Важным элементом управления для обеспечения пользователей является возможность включения и отключения фонового опроса.

В приложении PhotoGallery мы добавим пункт меню на панели приложения, который будет переключать вашего работника, когда он будет выбран. Мы также обновим запрос на работу, чтобы работник запускался периодически, а не один раз.

Для переключения между функциями сначала необходимо определить, запущен ли работник в данный момент. Для этого добавьте `QueryPreferences`, чтобы сохранить флаг, указывающий, включен ли работник.

Листинг 27.14. Сохранение состояния Worker (QueryPreferences.kt)

```
private const val PREF_SEARCH_QUERY = "searchQuery"
private const val PREF_LAST_RESULT_ID = "lastResultId"
private const val PREF_IS_POLLING = "isPolling"

object QueryPreferences {
    ...
        fun setLastResultId(context: Context, lastResultId: String) {
        ...
```

```
    }

    fun isPolling(context: Context): Boolean {
                                            return
PreferenceManager.getDefaultSharedPreferences(c
ontext)
            .getBoolean(PREF_IS_POLLING, false)
    }

       fun setPolling(context: Context, isOn:
Boolean) {
        PreferenceManager.getDefaultSharedPrefe
rences(context).edit {
            putBoolean(PREF_IS_POLLING, isOn)
        }
    }
}
```

Далее добавим строковые ресурсы, которые нужны вашему меню опций. Вам понадобятся две строки: одна для того, чтобы предложить пользователю включить опрос, а другая — чтобы отключить.

Листинг 27.15. Добавление ресурсов переключения опроса (res/values/strings.xml)

```
<resources>
    ...
     <string name="new_pictures_text">You have
new pictures in PhotoGallery.</string>
           <string  name="start_polling">Start
polling</string>
```

```
    <string name="stop_polling">Stop
polling</string>
</resources>
```

После установки строк откройте файл res/menu/fragment_photo_gallery.xml меню и добавьте новый элемент для переключения между опросами.

Листинг 27.16. Добавление переключения между опросами (res/menu/fragment_photo_gallery.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<menu
xmlns:android="http://schemas.android.com/apk/r
es/android"
      xmlns:app="http://schemas.android.com/apk
/res-auto">
    ...
    <item android:id="@+id/menu_item_clear"
          android:title="@string/clear_search"
          app:showAsAction="never" />

                                          <item
android:id="@+id/menu_item_toggle_polling"
          android:title="@string/start_polling"
          app:showAsAction="ifRoom|withText"/>
</menu>
```

Текстом по умолчанию для этого элемента является строка start_polling. Вам нужно будет изменить этот текст, если работник уже запущен. Откройте файл PhotoGalleryFragment.kt и измените код функции OnCreateOptionsMenu(...), чтобы проверить, запущен ли

работник, и, если да, установить правильный текст заголовка. Также удалите логику OneTimeWorkRequest из функции onCreate(...), так как она больше не нужна.

Листинг 27.17. Установка правильного текста пункта меню (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : Fragment() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
        lifecycle.addObserver(thumbnailDownload
er)

                                ~~val constraints =~~
~~Constraints.Builder()~~
                ~~.setRequiredNetworkType(Network~~
~~Type.UNMETERED)~~
                ~~.build()~~
                                ~~val workRequest =~~
~~OneTimeWorkRequest~~
                ~~.Builder(PollWorker::class.java~~
~~)~~
                ~~.setConstraints(constraints)~~
                ~~.build()~~
            ~~WorkManager.getInstance()~~
                ~~.enqueue(workRequest)~~
    }
    ...
       override fun onCreateOptionsMenu(menu:
Menu, inflater: MenuInflater) {
```

```
            ...
            searchView.apply {
                ...
            }

                            val toggleItem =
menu.findItem(R.id.menu_item_toggle_polling)
                            val isPolling =
QueryPreferences.isPolling(requireContext())
            val toggleItemTitle = if (isPolling) {
                R.string.stop_polling
            } else {
                R.string.start_polling
            }
            toggleItem.setTitle(toggleItemTitle)
        }
        ...
    }
```

Наконец, измените функцию `onOptionsItemSelected(...)`, чтобы она отвечала на щелчки переключателя опросов. Если работник не запущен, создайте новый `PeriodicWorkRequest` и запланируйте его с помощью `WorkManager`. Если работник запущен, его нужно остановить.

Листинг 27.18. Обработка кликов по опросам (PhotoGalleryFragment.kt)

```
private const val TAG = "PhotoGalleryFragment"
private const val POLL_WORK = "POLL_WORK"
```

```
class PhotoGalleryFragment : Fragment() {
    ...
      override fun onOptionsItemSelected(item:
MenuItem): Boolean {
        return when (item.itemId) {
            R.id.menu_item_clear -> {
                photoGalleryViewModel.fetchPhot
os("")
                true
            }
            R.id.menu_item_toggle_polling -> {
                                val isPolling =
QueryPreferences.isPolling(requireContext())
                if (isPolling) {
                    WorkManager.getInstance().c
ancelUniqueWork(POLL_WORK)
                    QueryPreferences.setPolling
(requireContext(), false)
                } else {
                                    val constraints =
Constraints.Builder()
                            .setRequiredNetworkType
(NetworkType.UNMETERED)
                            .build()
                             val periodicRequest =
PeriodicWorkRequest
                            .Builder(PollWorker::cl
ass.java, 15, TimeUnit.MINUTES)
                            .setConstraints(constra
ints)
                            .build()
```

```
                    WorkManager.getInstance().e
nqueueUniquePeriodicWork(POLL_WORK,
                        ExistingPeriodicWorkPol
icy.KEEP,
                        periodicRequest)
                    QueryPreferences.setPolling
(requireContext(), true)
                }
                activity?.invalidateOptionsMenu
()
                return true
            }
                                        else   ->
super.onOptionsItemSelected(item)
        }
    }
    ...
}
```

Обратите внимание на блок `else`, который вы добавили здесь. Если работник в данный момент *не* запущен, то мы назначим новый запрос на работу с `WorkManager`. В этом случае вы используете `PeriodicWorkRequest`, чтобы заставить вашего работника самого запланировать себя через некоторый интервал. В запросе используется конструктор, такой как `OneTimeWorkRequest`, который вы использовали ранее. Конструктору нужен класс `Worker`, а также интервал выполнения.

Если вы считаете, что 15 минут — это слишком долго, то вы правы. Однако если вы установите меньшее значение интервала, то обнаружите, что работник все равно выполняется с 15-минутным интервалом. Это минимальный интервал,

допустимый для PeriodicWorkRequest, чтобы система не была постоянно привязана к выполнению одного и того же рабочего запроса. Это экономит системные ресурсы и ресурс батареи.

Конструктор PeriodicWorkRequest принимает ограничения, как и одноразовый запрос, поэтому вы можете добавить ограничение на работу с сетью с измерением трафика. Когда вы хотите спланировать запрос на работу, вы используете класс WorkManager, но на этот раз вы используете функцию enqueueUniquePeriodicWork(...). Эта функция принимает имя типа String, политику и ваш рабочий запрос. Имя позволяет однозначно идентифицировать запрос, что полезно, если вы захотите его отменить.

Политика работы подсказывает менеджеру, что делать, если вы уже запланировали запрос на работу с определенным именем. В этом случае вы используете опцию KEEP, которая отказывается от нового запроса в пользу уже существующего. Другая опция — REPLACE, которая, как следует из названия, заменяет существующий запрос на новый.

Если работник уже запущен, то вам необходимо сообщить WorkManager об отмене запроса на работу. В этом случае для удаления периодического запроса на работу вызывается функция cancelUniqueWork(...) с именем POLL_WORK.

Запустите приложение. Вы увидите новый пункт меню для переключения опроса. Если вы не хотите ждать 15 минут интервала, вы можете отключить опрос, подождать несколько секунд, а затем включить его снова.

Теперь PhotoGallery может автоматически держать пользователя в курсе последних изображений, даже если приложение не запущено. Но есть одна проблема: пользователь будет получать уведомления в любое время, когда приходят новые изображения, — даже если приложение уже запущено.

Это нежелательно, потому что отвлекает внимание пользователя от вашего приложения. Кроме того, если пользователь нажмет на уведомление, запустится новый экземпляр `PhotoGalleryActivity`, который будет добавлен в обратный стек вашего приложения.

Вы исправите это в следующей главе, предотвратив появление уведомлений во время работы PhotoGallery. При создании этих обновлений вы узнаете, как прослушивать сообщения от интентов трансляции и как работать с такими интентами с помощью приемника трансляции.

28. Широковещательные интенты

Выглядит неудобно и даже навязчиво, когда пользователь получает уведомления от приложения, которое и без того в данный момент активно использует. В этой главе мы доведем приложение PhotoGallery до ума, убрав появление уведомлений о новых фотографиях во время работы приложения.

При внесении этих обновлений вы узнаете, как прослушивать *широковещательныеинтенты* от системы и как обрабатывать их при помощи широковещательных приемников.

Также мы займемся динамической отправкой и получением широковещательных интентов во время выполнения. Наконец, мы используем упорядоченные широковещательные рассылки для определения того, выполняется в настоящий момент приложение на переднем плане или нет.

Обычные и широковещательные интенты

На устройствах Android постоянно что-нибудь происходит. Точки Wi-Fi входят и выходят из зоны приема, устанавливаются пакеты, поступают телефонные звонки и текстовые сообщения.

Если о возникновении некоторого события должны узнать многие компоненты системы, Android использует для распространения информации широковещательные интенты (broadcast intent).

Трансляции, которые отправляются из системы, называются *системными широковещательными трансляциями*, но вы также можете отправлять и получать свои собственные трансляции.

Механизм приема как системных, так и пользовательских трансляций идентичен, но в этой главе вы будете работать только с пользовательскими трансляциями.

Широковещательные интенты работают примерно так же, как уже знакомые вам обычные интенты, не считая того что их могут получать сразу несколько компонентов, называемых широковещательными приемниками (broadcast receivers) (рис. 28.1).

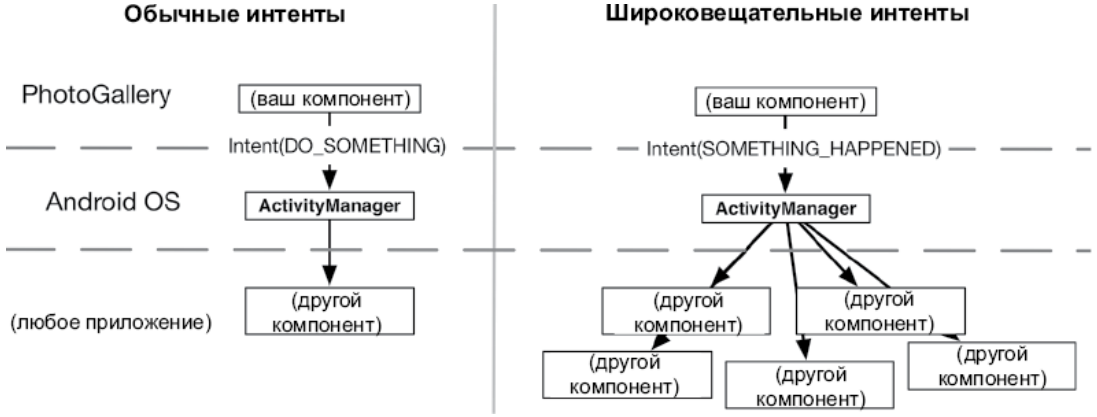


Рис. 28.1. Обычные и широковещательные интенты

Activity и службы должны реагировать на неявные интенты, когда они используются как часть открытого API. В других обстоятельствах явных интентов почти всегда хватает. С другой стороны, широковещательные интенты существуют именно для того, чтобы отправлять интенты более чем одному пользователю. Итак, хотя широковещательные приемники могут реагировать на явные интенты, они редко используются таким образом, потому что у явных интентов только один получатель.

Отключение уведомлений при открытом приложении

Уведомления работают отлично, но они выдаются даже в случае, если у пользователя уже и без того работает

приложение. Вы можете использовать широковещательные интенты для настройки поведения `PollWorker` в зависимости от того, находится ли ваше приложение на переднем плане.

Во-первых, мы будем отправлять широковещательный интент от вашего `PollWorker` при каждом получении новых фотографий. Затем зарегистрируем два широковещательных приемника. Первый приемник будет зарегистрирован в вашем манифесте для Android. Каждый раз, получая трансляцию от `PollWorker`, он будет уведомлять пользователя, как и раньше. Второй приемник регистрируется динамически и будет активен только тогда, когда ваше приложение видно пользователю. Его работа будет заключаться в перехвате трансляций до попадания в приемник, чтобы уведомление не выводилось.

Использование двух широковещательных приемников может показаться странным, но в Android нет механизма для определения того, какие activity или фрагменты выполняются в настоящее время. Поскольку `PollWorker` не может напрямую сказать, виден ли в данный момент пользовательский интерфейс, не получится отфильтровать показ уведомления с помощью простого оператора условия в `PollWorker`. Аналогично вы не можете выбрать условную отправку сообщения, основываясь на том, видно ли приложение PhotoGallery пользователю, поскольку нет способа определить, в каком состоянии находится ваше приложение. Однако вы можете использовать два приемника и настроить их так, чтобы только один из них реагировал на трансляцию. Так мы и поступим.

Отправка широковещательных интентов

Самая простая часть решения — отправка ваших собственных широковещательных интентов. Если говорить конкретнее, вы

разошлете широковещательный интент, который уведомляет заинтересованные компоненты о том, что оповещение о новых результатах поиска готово. Чтобы отправить широковещательный интент, просто создайте интент и передайте его `sendBroadcast(Intent)`. В нашем случае широковещательная рассылка будет применяться к определенному нами действию, поэтому также следует определить константу действия.

Измените код в файле `PollWorker.kt`, как показано ниже.

Листинг 28.1. Отправка широковещательного интента (PollWorker.kt)

```
class PollWorker(val context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams) {

    override fun doWork(): Result {
        ...
        val resultId = first().id
        if (resultId == lastResultId) {
                Log.i(TAG, "Got an old result: $resultId")
        } else {
            ...
                    val notificationManager = NotificationManagerCompat.from(context)
                notificationManager.notify(0, notification)

            context.sendBroadcast(Intent(ACTION_SHOW_NOTIFICATION))
```

```
        }

        return Result.success()
    }

    companion object {
        const val ACTION_SHOW_NOTIFICATION =
            "com.bignerdranch.android.photogall
ery.SHOW_NOTIFICATION"
    }
}
```

Создание и регистрация автономного широковещательного приемника

Трансляция уже отправлена, но пока никто ее не слушает. В качестве приемника у нас будет `BroadcastReceiver`. Существует два типа широковещательных приемников, и мы в данном случае будем использовать *автономный*.

Автономным приемником называется широковещательный приемник, объявленный в манифесте. Такой приемник может активизироваться даже в том случае, если процесс приложения мертв. (Далее вы узнаете о динамических приемниках, которые могут быть связаны с жизненным циклом визуального компонента приложения — фрагмента, activity и т.д.)

Подобно службам и activity, широковещательные приемники должны быть зарегистрированы в системе для выполнения любой полезной работы. Если приемник не зарегистрирован, то система не будет отправлять ему интенты, а функция `onReceive()` приемника не будет выполняться, как ожидалось.

Прежде чем регистрировать широковещательный приемник, его нужно написать. Создайте Kotlin-класс с именем

NotificationReceiver, являющийся подклассом android.content.BroadcastReceiver.

Листинг 28.2. Наш первый широковещательный приемник (NotificationReceiver.kt)

```
private const val TAG = "NotificationReceiver"

class NotificationReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        Log.i(TAG, "received broadcast: ${intent.action}")
    }
}
```

Широковещательный приемник — компонент, который получает интенты, как и служба или activity. При получении интента экземпляром NotificationReceiver будет вызвана его функция onReceive(...).

Откройте файл AndroidManifest.xml и включите объявление NotificationReceiver как автономный приемник.

Листинг 28.3. Включение приемника в манифест (manifests/AndroidManifest.xml)

```
<application ... >
                                    <activity
android:name=".PhotoGalleryActivity">
```

```
            ...
        </activity>
                                                <receiver
android:name=".NotificationReceiver">
        </receiver>
    </application>
```

Для приема трансляций, которые нужны приемнику, нужно настроить ему фильтр интентов. Этот фильтр будет вести себя точно так же, как и те, которые вы использовали с неявными интентами, за исключением того, что он будет фильтровать широковещательные интенты вместо обычных. Добавьте приемнику фильтр интентов, чтобы он принимал их с действием `SHOW_NOTIFICATION`.

Листинг 28.4. Добавление фильтра интентов приемнику (manifests/AndroidManifest.xml)

```
<receiver android:name=".NotificationReceiver">
  <intent-filter>
      <action
          android:name="com.bignerdranch.androi
d.photogallery.SHOW_NOTIFICATION" />
  </intent-filter>
</receiver>
```

Если после внесения этих изменений вы запустите PhotoGallery на устройстве под управлением Android Oreo или выше, вы не увидите свой журнал. Фактически функция `onReceive(...)` вашего приемника вообще не будет вызвана. Но на старых версиях Android журнал будет таким, как вы ожидаете. Это связано с ограничениями, которые новые версии

Android накладывают на трансляции. Но не бойтесь — ваша работа не будет напрасной.

Вы можете обойти эти ограничения, отправив вашу трансляцию с разрешения.

Ограничение широковещательной рассылки

Одна из проблем с использованием широковещательной рассылки заключается в том, что любой компонент в системе может прослушивать ее или инициировать ваши приемники. И то и другое обычно нежелательно. При указании разрешения на трансляцию вы также заставите ваш новый широковещательный приемник работать на более новых версиях Android, так что можете исправить две ошибки по цене одной.

Эти несанкционированные вмешательства в ваши личные дела можно предотвратить. Если приемник объявлен в манифесте и является внутренним по отношению к вашему приложению, добавьте в приемник разрешение `android:exported="false"`. С ним приемник становится невидимым для других приложений в системе. Применение разрешения означает, что только те компоненты, которые запросили (и получили) разрешение, могут передавать трансляцию приемнику.

Объявите и получите разрешение в `AndroidManifest.xml`.

Листинг 28.5. Добавление приватного разрешения (manifests/AndroidManifest.xml)

```
<manifest ... >
```

```
                                        <permission
android:name="    com.bignerdranch.android.photogallery.PRIVATE"
                    android:protectionLevel="signature" />

                                        <uses-permission
android:name="android.permission.INTERNET"/>
                                        <uses-permission
android:name="    com.bignerdranch.android.photogallery.PRIVATE" />
    ...
</manifest>
```

В этой разметке мы определяем собственное разрешение с *уровнем защитыsignature* (вскоре об уровнях защиты будет рассказано более подробно). Само разрешение представляет собой простую строку, как и действия интентов, категории и системные разрешения, использовавшиеся ранее. Разрешение всегда необходимо получить для его использования, даже если вы сами определяете его. Таковы правила.

Обратите внимание на константу, выделенную цветом фона. Эта строка является уникальным идентификатором пользовательского разрешения, который вы будете использовать для того, чтобы ссылаться на разрешение в другом месте в вашем манифесте, а также из вашего кода Kotlin, когда вы отправляете приемнику широковещательный интент. Идентификатор должен быть везде одинаковым. Лучше скопируйте его, вместо того чтобы вводить вручную.

Теперь применим разрешение к тегу приемника и присвоим атрибуту `exported` значение `false`.

Листинг 28.6. Установка разрешения и свойства exported равным false (manifests/AndroidManifest.xml)

```
<manifest ... >
    ...
    <application ... >
        ...
                                        <receiver
android:name=".NotificationReceiver"
                    android:permission="com.bigne
rdranch.android.photogallery.PRIVATE"
                    android:exported="false">
                    ...
        </receiver>
    </application>
</manifest>
```

Чтобы использовать разрешение, определите соответствующую константу в коде и передайте ее при вызове `sendBroadcast(...)`.

Листинг 28.7. Отправка с разрешением (PollWorker.kt)

```
class    PollWorker(val    context:    Context,
workerParams: WorkerParameters) :
    Worker(context, workerParams) {

    override fun doWork(): Result {
        ...
        val resultId = first().id
        if (resultId == lastResultId) {
                Log.i(TAG, "Got an old result:
$resultId")
```

```
        } else {
            ...
                    val notificationManager =
NotificationManagerCompat.from(context)
                notificationManager.notify(0,
notification)

            context.sendBroadcast(Intent(ACTION
_SHOW_NOTIFICATION), PERM_PRIVATE)
        }

        return Result.success()
    }

    companion object {
        const val ACTION_SHOW_NOTIFICATION =
            "com.bignerdranch.android.photogall
ery.SHOW_NOTIFICATION"
                const val PERM_PRIVATE =
   "com.bignerdranch.android.photogallery.PRIV
ATE"
    }
}
```

Теперь только ваше приложение может включать данный приемник. Снова запустите приложение PhotoGallery. Вы должны увидеть сообщения журнала от `NotificationReceiver` на панели **Logcat** (хотя ваши уведомления еще не умеют себя вести как следует и все равно будут отображаться).

Подробнее об уровнях защиты

Каждое пользовательское разрешение должно задавать уровень защиты — атрибут `android:protectionLevel`. Уровень защиты сообщает Android, как будет использоваться разрешение. В нашем примере используется уровень защиты `signature`.

Он означает, что если другое приложение захочет использовать ваше разрешение, то оно должно быть снабжено цифровой подписью с таким же ключом, как у вашего приложения. Обычно этот вариант оптимален для разрешений, используемых в вашем приложении. Так как ваш ключ недоступен для других разработчиков, они не могут получить доступ к функциональности, которую защищает ваше разрешение. Вдобавок, поскольку у вас есть собственный ключ, вы можете использовать разрешение в других приложениях, которые будут написаны позднее.

Четыре допустимых уровня защиты представлены в табл. 28.1.

Таблица 28.1. Значения атрибута protectionLevel

Значение	Описание
normal	Используется для защиты функциональности приложения, которая не выполняет потенциально опасных операций, например обращений к защищенным личным данным или отправки данных в интернете. Пользователь видит разрешение перед установкой приложения, но не получает запрос на его явное предоставление. android.permission.INTERNET использует этот уровень, как и разрешение на вибрацию телефона
dangerous	Используется для всего, для чего не используется normal: для обращений к личным данным, отправки и получения данных из сетевых интерфейсов, обращения к оборудованию, которое может использоваться для шпионских целей, и вообще ко всему, что может создать реальные проблемы для пользователя. В частности, разрешения на доступ к интернету, камере и контактам относятся к этой категории. Начиная с версии Marshmallow разрешения dangerous требуют вызова функции requestPermission(...) во время выполнения для получения от пользователя явного подтверждения на выполнение опасной операции

Значение	Описание
signature	Система предоставляет это разрешение, если приложение подписано тем же сертификатом, что и приложение, в котором объявлено разрешение, или отклоняет его в противном случае. Если разрешение предоставляется, пользователь об этом не оповещается. Значение обычно используется для функциональности, внутренней для вашего приложения: так как у вас имеется сертификат, а приложение может использоваться только приложениями, подписанными тем же сертификатом, вы контролируете состав пользователей разрешения. В нашем случае значение не позволит посторонним видеть вашу широковещательную рассылку, но при желании вы можете написать другое приложение, которое также сможет ее прослушивать
signature-OrSystem	Аналог signature, но разрешение также предоставляется всем пакетам в образе системы Android. Используется для взаимодействия с приложениями, встроенными в образ системы; если разрешение предоставляется, то пользователь не оповещается. Для большинства разработчиков интереса не представляет

Создание и регистрация динамического приемника

Теперь вам понадобится приемник для широковещательного интента `ACTION_SHOW_NOTIFICATION`. Его задача будет заключаться в предотвращении показа уведомления, если приложение находится на переднем плане.

Этот приемник будет зарегистрирован лишь тогда, когда activity находится на переднем плане. Если бы этот приемник был объявлен на более длительный срок службы (например, в течение жизненного цикла процесса вашего приложения), то вам пришлось бы как-то иначе узнавать, что `PhotoGalleryFragment` запущен (и тогда динамический приемник был бы не нужен).

Задача решается использованием динамического широковещательного приемника. Он регистрируется в коде, а не в манифесте. Для регистрации приемника используется вызов функции `Context.registerReceiver(Broadcast-Receiver,IntentFilter)`, а для ее отмены — вызов `Context.unregisterReceiver(BroadcastReceiver)`. Сам

приемник обычно определяется как внутренний экземпляр или лямбда, по аналогии со слушателем щелчка на кнопке. Но поскольку в функциях `registerReceiver(...)` и `unregisterReceiver(...)` должен использоваться один экземпляр, приемник необходимо присвоить переменной экземпляра.

Создайте новый абстрактный класс `VisibleFragment`, суперклассом которого является `Fragment`. Этот класс будет представлять обобщенный фрагмент, скрывающий оповещения переднего плана (другой такой фрагмент мы напишем в главе 29).

Листинг 28.8. Класс VisibleFragment (VisibleFragment.kt)

```
abstract class VisibleFragment : Fragment() {

    private val onShowNotification = object :
BroadcastReceiver() {
        override fun onReceive(context:
Context, intent: Intent) {
            Toast.makeText(requireContext(),
                "Got a broadcast:
${intent.action}",
                Toast.LENGTH_LONG)
                .show()
        }
    }

    override fun onStart() {
        super.onStart()
        val filter =
IntentFilter(PollWorker.ACTION_SHOW_NOTIFICATIO
```

```
N)
        requireActivity().registerReceiver(
            onShowNotification,
            filter,
            PollWorker.PERM_PRIVATE,
            null
        )
    }

    override fun onStop() {
        super.onStop()
        requireActivity().unregisterReceiver(on
ShowNotification)
    }
}
```

Обратите внимание: чтобы передать объект `IntentFilter`, необходимо создать его в коде. В данном случае объект `IntentFilter` идентичен фильтру, определяемому следующей разметкой XML:

```
<intent-filter>
    <action android:name=
            "com.bignerdranch.android.photogall
ery.SHOW_NOTIFICATION" />
</intent-filter>
```

Любой объект `IntentFilter`, который можно выразить в XML, также может быть представлен в коде подобным образом. Просто вызывайте функции `addCategory(String)`, `addAction(String)`, `addDataPath(String)` и так далее для настройки фильтра.

Динамически регистрируемые широковещательные приемники также должны принять меры для своей деинициализации. Как правило, если вы регистрируете приемник в функции жизненного цикла, вызываемом при запуске, в соответствующей функции завершения вызывается функция `Context.unregisterReceiver(BroadcastReceiver)`. В нашем примере регистрация выполняется в `onStart()` и отменяется в `onStop()`. Аналогичным образом, если бы регистрация выполнялась в `onCreate(...)`, то отменяться она должна была бы в `onDestroy()`.

(Кстати, будьте осторожны с `onCreate(...)` и `onDestroy()` при удержании фрагментов. Функция `getActivity()` будет возвращать разные значения в `onCreate(...)` и `onDestroy()`, если экран был повернут. Если вы хотите регистрировать/отменять регистрацию во `Fragment.onCreate(...)` и `Fragment.onDestroy()`, используйте `requireActivity().getApplicationContext()`.)

Сделайте `PhotoGalleryFragment` подклассом только что созданного класса `VisibleFragment`.

Листинг 28.9. Фрагмент делается видимым (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : ~~Fragment()~~
VisibleFragment() {
    ...
}
```

Запустите PhotoGallery и пару раз переключите режим фонового опроса. Вы увидите, как наряду с бегущей строкой на

экране появится окно сообщения (рис. 28.2).

Рис. 28.2. Доказательство, что ваш приемник широковещательной рассылки существует

Передача и получение данных с упорядоченной широковещательной рассылкой

Пора сделать последний шаг: позаботиться о том, чтобы динамически зарегистрированный приемник всегда получал широковещательный интент `PollService.ACTION_SHOW_NOTIFICATION` до того, как он будет принят другими приемниками, и отменял отправку оповещения.

На данный момент наша собственная широковещательная закрытая рассылка работает, но передача данных осуществляется только в одном направлении (рис. 28.3).

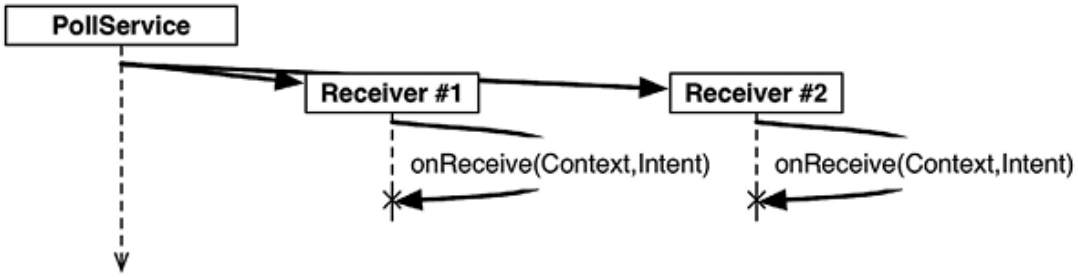


Рис. 28.3. Обычные широковещательные интенты

Это объясняется тем, что на концептуальном уровне обычный широковещательный интент принимается всеми одновременно. Сейчас `onReceive(...)` вызывается в главном потоке, поэтому на практике приемники не выполняются многопоточно. Мы не можем рассчитывать на то, что они будут выполняться в каком-то конкретном порядке, или на то, чтобы узнать, когда все они завершат выполнение. Этот факт значительно затрудняет взаимодействие между широковещательными приемниками или получение информации отправителем интента от приемников.

Двустороннее взаимодействие можно реализовать с использованием *упорядоченных* широковещательных интентов (рис. 28.4). Упорядоченные широковещательные интенты позволяют серии широковещательных приемников обработать широковещательный интент по порядку.

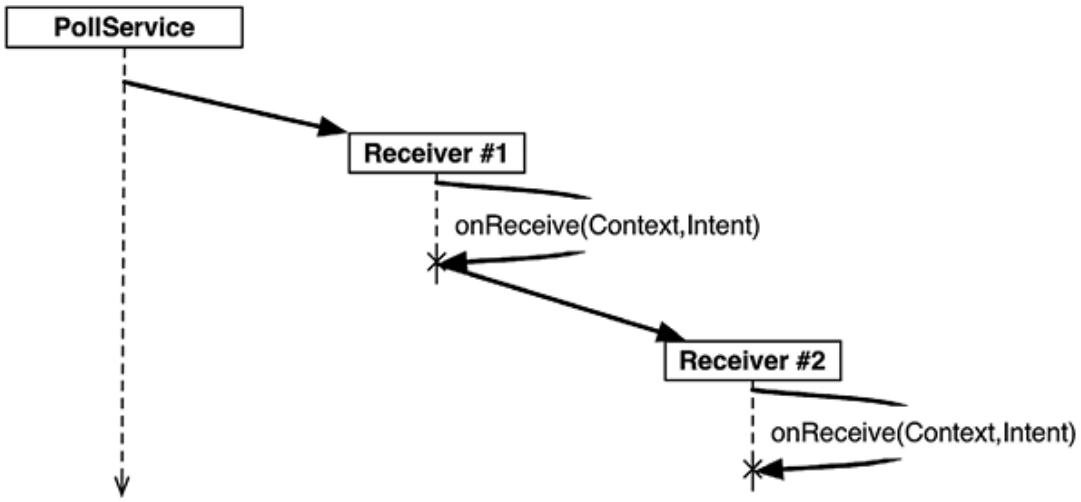


Рис. 28.4. Упорядоченные широковещательные интенты

На получающей стороне все выглядит практически так же, как при обычной широковещательной рассылке. Однако в вашем распоряжении появляется дополнительный инструмент: набор функций, используемых для изменения возвращаемого значения вашего приемника. В нашей ситуации нужно отменить оповещение; эта информация передается в виде простого целочисленного кода результата путем присвоения `resultCode` значения `Activity.RESULT_CANCELED`.

Внесите изменения в `VisibleFragment`, чтобы вернуть информацию отправителю `SHOW_NOTIFICATION`. Информация также будет отправляться другим широковещательным приемникам по цепочке.

Листинг 28.10. Возвращение простого результата (VisibleFragment.kt)

```
private const val TAG = "VisibleFragment"
abstract class VisibleFragment : Fragment() {

    private val onShowNotification = object :
BroadcastReceiver() {
```

```
                override fun onReceive(context:
Context, intent: Intent) {
                    Toast.makeText(requireActivity(
),
                                   "Got a broadcast:" +
intent.getAction(),
                                   Toast.LENGTH_LONG)
                                   .show()
                // Если мы получаем это, то видимы,
поэтому отмените оповещения
                                Log.i(TAG, "canceling
notification")
                                          resultCode =
Activity.RESULT_CANCELED
            }
        }
        ...
}
```

Так как в нашем примере необходимо лишь подать сигнал «да/нет», нам достаточно кода результата. Если потребуется вернуть более сложные данные, используйте значение `resultData` или вызывайте функцию `setResultExtras(Bundle?)`. А если захотите задать все три значения, вызовите функцию `setResult(Int,String?,Bundle?)`. После того как все возвращаемые значения будут заданы, каждый последующий приемник сможет увидеть или изменить их.

Чтобы эти функции делали что-то полезное, широковещательная передача должна быть упорядоченной. Напишите новый функционал для отправки упорядоченных широковещательных интентов из `PollService`. Эта функция

будет упаковывать обращение к Notification и отправлять его в широковещательном режиме. Измените код функции doWork(), чтобы она вызывала вашу новую функцию и отправляла упорядоченный широковещательный интент, вместо того чтобы отправлять оповещение непосредственно NotificationManager.

Листинг 28.11. Отправка упорядоченных широковещательных интентов (PollWorker.kt)

```
class PollWorker(val context: Context, workerParams: WorkerParameters) :
    Worker(context, workerParams) {

    override fun doWork(): Result {
        ...
        val resultId = items.first().id
        if (resultId == lastResultId) {
                Log.i(TAG, "Got an old result: $resultId")
        } else {
            ...
                            val notification = NotificationCompat
                                .Builder(context, NOTIFICATION_CHANNEL_ID)
                ...
                .build()

                        ~~val notificationManager = NotificationManagerCompat.from(context)~~
```

```
                    notificationManager.notify(0,
notification)

                    context.sendBroadcast(Intent(AC
TION_SHOW_NOTIFICATION), PERM_PRIVATE)

                    showBackgroundNotification(0,
notification)
        }

        return Result.success()
    }

    private fun showBackgroundNotification(
        requestCode: Int,
        notification: Notification
    ) {
                                val    intent    =
Intent(ACTION_SHOW_NOTIFICATION).apply {
            putExtra(REQUEST_CODE, requestCode)
                            putExtra(NOTIFICATION,
notification)
        }

            context.sendOrderedBroadcast(intent,
PERM_PRIVATE)
    }

    companion object {
        const val ACTION_SHOW_NOTIFICATION =
            "com.bignerdranch.android.photogall
ery.SHOW_NOTIFICATION"
```

```
                const val PERM_PRIVATE =
"com.bignerdranch.android.photogallery.PRIVATE"
        const val REQUEST_CODE = "REQUEST_CODE"
        const val NOTIFICATION = "NOTIFICATION"
    }
}
```

Функция `Context.sendOrderedBroadcast(Intent,String?)` ведет себя очень похоже на функцию `sendBroadcast(...)`, но при этом гарантирует, что трансляция будет доставлена приемнику вовремя. Код результата устанавливается равным `Activity.RESULT_OK`, когда эта трансляция будет отправлена.

Научим `NotificationReceiver` отправлять уведомление пользователю.

Листинг 28.12. Реализация приемника результатов (NotificationReceiver.kt)

```
private const val TAG = "NotificationReceiver"

class          NotificationReceiver          :
BroadcastReceiver() {

     override fun onReceive(context: Context,
intent: Intent) {
          Log.i(TAG, "received broadcast:
${i.action} result: $resultCode")
        if (resultCode != Activity.RESULT_OK) {
              // Активность переднего плана
отменила возврат трансляции.
        }
```

```
                val requestCode =
intent.getIntExtra(PollWorker.REQUEST_CODE, 0)
            val notification: Notification =
                intent.getParcelableExtra(PollWorke
r.NOTIFICATION)

            val notificationManager =
NotificationManagerCompat.from(context)
            notificationManager.notify(requestCode,
notification)
        }
    }
```

Чтобы `NotificationReceiver` принимал трансляцию после вашего динамически зарегистрированного приемника (чтобы можно было проверить, следует ли ему размещать уведомление в `NotificationManager`), вам необходимо установить низкий приоритет для `NotificationReceiver` в манифесте. Назначьте приоритет `-999`, чтобы он работал последним. Это самый низкий приоритет, который можно задать (значения `-1000` и ниже зарезервированы).

Листинг 28.13. Приоритизация приемника уведомлений (manifests/AndroidManifest.xml)

```
<receiver ... >
    <intent-filter android:priority="-999">
        <action
            android:name="com.bignerdranch.andr
oid.photogallery.SHOW_NOTIFICATION" />
    </intent-filter>
```

```
</receiver>
```

Запустите приложение PhotoGallery. Вы увидите, что уведомления больше не появляются, когда приложение находится на переднем плане. (Вы можете включать и выключать опрос, чтобы запустить ваш `PollWorker`, который будет запущен `WorkManager`.)

Приемники и продолжительные задачи

Что делать, если вы хотите, чтобы широковещательный интент запускал задачу более продолжительную, чем допускают ограничения главного цикла выполнения? Есть два варианта.

Первый — выделить эту работу в службу и запустить ее в широковещательном приемнике. Служба может обрабатывать запрос столько времени, столько нужно. Мы рекомендуем именно этот метод. У службы есть довольно длинное окно, которое можно использовать для выполнения работы, но его все равно можно остановить, если выполнение будет слишком долгим (длина окна варьируется в зависимости от версии ОС и устройства, но, как правило, составляет на более новых устройствах несколько минут). Вы также можете выбрать запуск службы на переднем плане, чтобы снять все ограничения на длительность работы, что идеально подходит для таких задач, как резервное копирование фотографий, воспроизведение музыки или предоставление пошаговой навигации.

Второй вариант основан на использовании функции `BroadcastReceiver.goAsync()`. Этот функция возвращает объект `BroadcastReceiver.PendingResult`, который может использоваться для передачи результата в будущем. Таким образом, вы передаете объект `PendingResult` экземпляру `AsyncTask` для выполнения продолжительной работы, а потом

отвечаете на широковещательную передачу, вызывая функции `PendingResult`.

У этого способа есть недостаток: он менее гибок. Вам все равно приходится обрабатывать широковещательную передачу за десять секунд или около того, и в вашем распоряжении меньше архитектурных вариантов, чем при использовании службы. Конечно, у функции `goAsync()` есть одно огромное преимущество: в нем можно задавать результаты упорядоченных широковещательных интентов. Если вам нужно именно это, другие решения не подойдут. Только позаботьтесь о том, чтобы выполнение не заняло слишком много времени.

Для любознательных: локальные события

Широковещательные интенты предоставляют возможность глобального распространения информации в системе. А если вы хотите распространить информацию о событии только в границах процесса приложения? Для этого существует хорошая альтернатива — шина событий (event bus).

Работа *шины событий* основана на принципе использования общей шины (или потока данных), на которую может подписаться ваше приложение. При отправке события по шине происходит активизация подписавшихся компонентов с выполнением их кода обратного вызова.

Для работы с шиной событий в своих Android-приложениях мы используем стороннюю библиотеку EventBus (разработчик greenrobot). Среди других альтернатив стоит рассмотреть Otto (разработчик Square), еще одну реализацию шины событий, или классы `Subject` и `Observable` из библиотеки RxJava.

Android предоставляет локальный механизм отправки широковещательных интентов, который называется `LocalBroadcastManager`. Однако наш опыт показывает, что

упоминавшиеся сторонние библиотеки предоставляют более гибкие и удобные API для широковещательных локальных событий.

Использование EventBus

Чтобы использовать EventBus в своих приложениях, необходимо включить в проект зависимость для библиотеки. После создания зависимости вы определяете класс, представляющий событие (чтобы передать дополнительные данные, можно добавить поля в событие):

```
class NewFriendAddedEvent(val friendName: String)
```

Событие можно отправить по шине практически из любой точки приложения:

```
val eventBus: EventBus = EventBus.getDefault()
eventBus.post(NewFriendAddedEvent("Susie Q"))
```

Чтобы подписаться на получение событий, приложение сначала регистрируется на прослушивание шины. Часто регистрация и отмена регистрации activity или фрагментов осуществляется в соответствующих функциях жизненного цикла, таких как `onStart(...)` и `onStop(...)`:

```
// В каком-то фрагменте или действии...
private lateinit var eventBus: EventBus

public override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    eventBus = EventBus.getDefault()
```

```
}

public override fun onStart() {
    super.onStart()
    eventBus.register(this)
}

public override fun onStop() {
    super.onStop()
    eventBus.unregister(this)
}
```

Чтобы указать, что должен сделать подписчик при отправке интересующего его события, реализуйте функцию с передачей типа события во входном параметре и снабдите ее аннотацией @Subscribe. Использование аннотации @Subscribe без параметров означает, что событие будет обрабатываться в том же потоке, из которого оно было отправлено. (Запись @Subscribe(threadMode=ThreadMode.MAIN) гарантирует, что событие, отправленное из фонового потока, будет обработано в главном потоке приложения.)

```
// В каком-то зарегистрированном компоненте — фрагменте или действии...
@Subscribe(threadMode = ThreadMode.MAIN)
fun onNewFriendAdded(event: NewFriendAddedEvent) {
     // Обновите UI или сделайте что-нибудь в ответ на событие...
}
```

Использование RxJava

Для реализации механизма широковещательной рассылки событий может использоваться RxJava — библиотека написания кода Java в «реактивном» стиле. Концепция «реактивного» кода широка и выходит за рамки излагаемого материала. В двух словах: она позволяет публиковать серии событий и подписываться на них, а также предоставляет широкий выбор общих инструментов для выполнения операций с сериями событий.

Разработчик создает Subject — объект, к которому можно обращаться с запросами на публикацию событий, а также с запросами на подписку:

```
val eventBus: Subject<Any, Any> =
                        PublishSubject.create<Any>
().toSerialized()
```

Этот объект используется для публикации событий:

```
val someNewFriend = "Susie Q"
val event = NewFriendAddedEvent(someNewFriend)
eventBus.onNext(event)
```

Пример подписки на события:

```
eventBus.subscribe { event: Any ->
    if (event is NewFriendAddedEvent) {
        val friendName = event.friendName
        // Обновить UI
    }
}
```

Преимущество решений на базе RxJava состоит в том, что объект `eventBus` также является реализацией `Observable`, представлением потока событий в RxJava. Это означает, что в

вашем распоряжении оказываются все средства для работы с событиями в RxJava. Если вас заинтересует эта тема, обращайтесь к вики на странице проекта RxJava: **github.com/ReactiveX/RxJava/wiki**.

Для любознательных: ограничения широковещательных приемников

Как вы уже убедились, у широковещательных приемников есть ограничения, объявленные в вашем Android-манифесте, которые могут привести к тому, что ваш приемник не будет вызван. Такое поведение не распространяется на приемники, которые вы регистрируете динамически, используя `registerReceiver(...)`, и применяется только тогда, когда ваше приложение работает на Android Oreo (API уровня 26) и более новых версиях Android.

Ограничения для автономных широковещательных приемников были введены с целью экономии ресурса аккумулятора и повышения производительности на устройствах пользователей. Если приемник зарегистрирован в манифесте, и приложение не работает, системе приходится запускать процесс всякий раз, когда приемнику нужно передать трансляцию. Этот процесс довольно незаметен, если запущенных приложений одно или два, но если их много, производительность сильно ухудшается.

Например, это может навредить работе пользователя, когда есть приложения, выполняющие автоматическое резервное копирование новых фотографий, сделанных камерой. Это может привести к запуску нескольких процессов в фоновом режиме, когда пользователь нажимает кнопку фотографирования в приложении для камеры, в результате чего камера вообще начнет игнорировать пользователя.

Чтобы снять проблему производительности, в новых версиях Android, начиная с Android Oreo, неявные трансляции больше не доставляются приемникам, объявленным в вашем манифесте, за некоторым исключением (на явные трансляции эти ограничения не влияют, но следует помнить, что такие трансляции передаются только на один приемник и используются редко).

Из этого правила исключается ряд системных трансляций. Приемники `BOOT_COMPLETE`, `TIMEZONE_CHANGED` и `NEW_OUTGOING_CALL`, зарегистрированные в вашем манифесте, все равно будут принимать эти трансляции. Дело в том, что они либо редко отправляются, либо альтернативного способа просто нет. Полный список исключений можно найти в документации для разработчиков Android на сайте **developer.android.com/guide/components/broadcast-exceptions**.

Как вы уже видели во время работы над этой главой, трансляции, отправленные с разрешением на уровне подписи, тоже лишены этого ограничения. Это позволяет вам продолжать использовать автономные широковещательные приемники, которые являются частными для вашего приложения и любых других приложений, которые вы делаете с использованием той же подписи. Поскольку только приложения от одного разработчика имеют разрешение на отправку этих трансляций, маловероятно, что в других частях системы возникнут проблемы с производительностью, вызванные такого рода трансляциями.

Для любознательных: проверка видимости фрагмента

Немного поразмыслив над реализацией PhotoGallery, мы видим, что глобальный механизм широковещательной рассылки использовался для рассылки интента

SHOW_NOTIFICATION. При этом получение рассылки при помощи разрешений ограничивается элементами, локальными для вашего приложения. Возникает естественный вопрос: почему я использую глобальный механизм, если просто передаю данные в своем приложении? Разве не логичнее использовать локальный механизм?

Дело в том, что мы намеренно пытались решить задачу проверки того, виден фрагмент PhotoGalleryFragment или нет. Для достижения цели мы воспользовались комбинацией упорядоченных рассылок, автономных приемников и динамически регистрируемых приемников. В Android существует и более прямолинейное решение этой задачи.

Если говорить конкретно, LocalBroadcastManager не подходит для широковещательных оповещений PhotoGallery и проверки видимости фрагментов по двум причинам.

Во-первых, LocalBroadcastManager не поддерживает упорядоченные широковещательные рассылки (хотя и предоставляет механизм широковещательной рассылки с блокировкой, а именно sendBroadcastSync(Intent)). Он не подойдет для PhotoGallery, потому что приемник NotificationReceiver должен гарантированно выполняться последним в цепочке.

Во-вторых, sendBroadcastSync(Intent) не поддерживает отправку и получение широковещательных интентов в разных потоках. В PhotoGallery интент должен отправляться из фонового потока (в PollWorker.doWork()), а приниматься в главном потоке (динамическим приемником, зарегистрированным PhotoGalleryFragment, в главном потоке в onStart(...)).

На момент написания книги семантика потоковой доставки LocalBroadcastManager была плохо документирована, и, по нашему опыту, ей не хватало логичности. Например, при

вызове `sendBroadcastSync(...)` из фонового потока все необработанные рассылки будут выданы в фоновый поток независимо от того, были ли они отправлены из главного потока.

Это не значит, что механизм `LocalBroadcastManager` бесполезен. Просто этот инструмент не подходит для задач, которыми мы занимались в этой главе.

29. Веб-серфинг и WebView

С каждой фотографией, загружаемой с Flickr, связана страница. В этой главе мы сделаем так, чтобы пользователь мог нажать на фотографию в PhotoGallery и просмотреть ее страницу. Вы освоите два разных способа интеграции веб-контента в ваши приложения, представленные на рис. 29.1. Первый способ работает с браузером, установленным на устройстве (слева), а второй использует класс `WebView` для отображения веб-контента (справа).

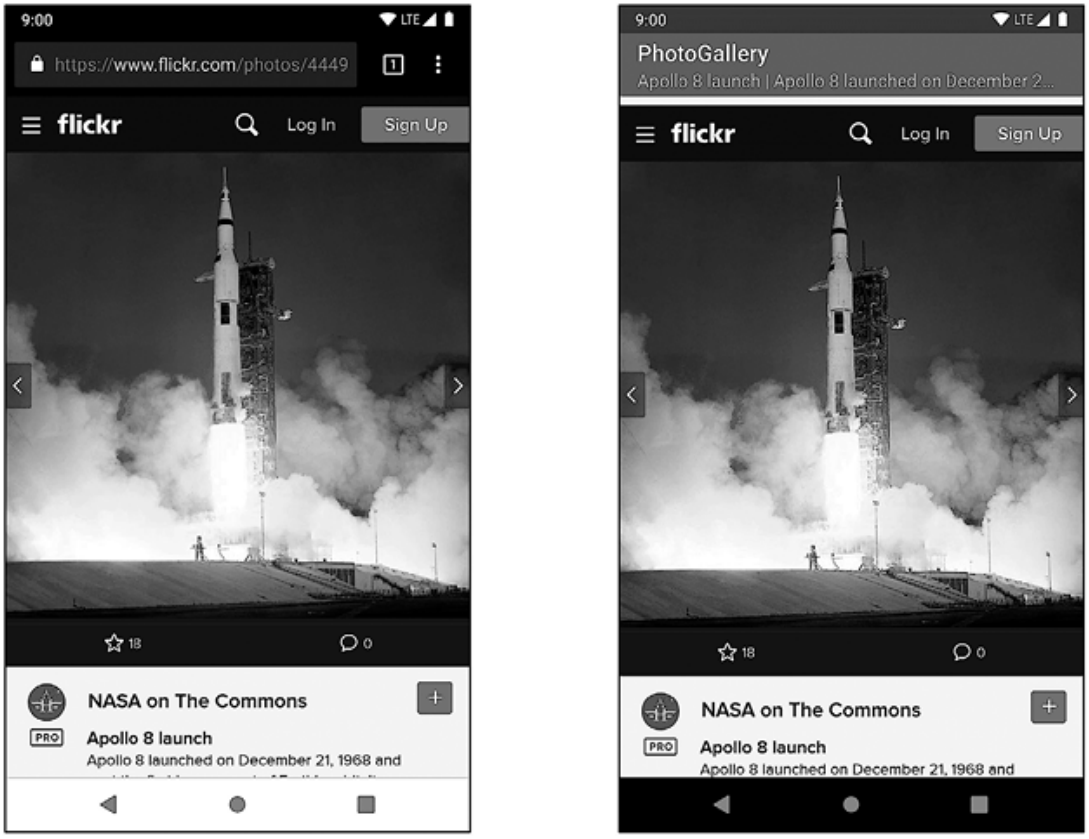


Рис. 29.1. Веб-контент: два разных подхода

И еще один фрагмент данных Flickr

Для обоих способов нам понадобится URL-адрес страницы фотографии на Flickr. Если присмотреться к разметке JSON, которую мы в настоящее время получаем для каждой фотографии, становится ясно, что страница в эти результаты не включена.

```
{
  "photos": {
    ...,
      "photo": [
        {
          "id": "9452133594",
          "owner": "44494372@N05",
          "secret": "d6d20af93e",
          "server": "7365",
          "farm": 8,
          "title": "Low and Wisoff at Work",
          "ispublic": 1,
          "isfriend": 0,
          "isfamily": 0,
          "url_s":"https://farm8.staticflickr.c
om/7365/9452133594_d6d20af93e_m.jpg"
        }, ...
      ]
    },
    "stat": "ok"
  }
```

(Обратите внимание, что `url_s` — это URL для мини-версии фотографии.)

Похоже, придется писать новые запросы JSON? К счастью, это не так. Обратившись к странице документации Flickr по

адресу **www.flickr.com/services/api/misc.urls.html**, мы видим, что URL-адреса страниц отдельных фотографий строятся по схеме:

```
https://www.flickr.com/photos/
идентификатор_пользователя/идентификатор_фото
```

Идентификатор фотографии совпадает со значением атрибута `photo_id` в разметке JSON. Мы уже сохранили его в поле `id` объекта `GalleryItem`. Как насчет идентификатора пользователя? Немного покопавшись в документации, мы находим, что атрибут `owner` в JSON содержит идентификатор пользователя. Таким образом, извлекая атрибут `owner`, мы можем построить URL-адрес по атрибутам из JSON фотографии:

```
https://www.flickr.com/photos/владелец/
идентификатор
```

Чтобы реализовать этот план, включите следующий код в `GalleryItem`.

Листинг 29.1. Добавление кода страницы фотографии (GalleryItem.kt)

```
data class GalleryItem(
    var title: String = "",
    var id: String = "",
     @SerializedName("url_s") var url: String =
"",
     @SerializedName("owner") var owner: String
= ""
) {
    val photoPageUri: Uri
        get() {
```

```
                                                    return
    Uri.parse("https://www.flickr.com/photos/")
                    .buildUpon()
                    .appendPath(owner)
                    .appendPath(id)
                    .build()
            }
    }
```

Для определения URL-адреса фотографии создается новое свойство `owner` и добавляется вычисляемое свойство `photoPageUri` для генерации URL-адресов страницы фото, как описано выше. Так как Gson превращает ответы JSON в `GalleryItems`, вы можете немедленно начать использовать свойство `photoPageUri`, не внося другие изменения в код.

Простой способ: неявные интенты

Сначала мы откроем страницу по URL-адресу при помощи старого знакомого — неявного интента. Этот интент запустит браузер с URL-адресом страницы фотографии.

Для начала нужно организовать прослушивание нажатий на элементах представления `RecyclerView`. Измените код реализации `PhotoHolder` из `PhotoGalleryFragment` и включите в нее слушателя кликов, который будет выдавать неявный интент.

Листинг 29.2. Выдача неявных интентов при нажатии (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : VisibleFragment()
{
```

```
    ...
    private inner class PhotoHolder(private val
itemImageView: ImageView)
                                                :
RecyclerView.ViewHolder(itemImageView),
    View.OnClickListener {

            private lateinit var galleryItem:
GalleryItem

        init {
            itemView.setOnClickListener(this)
        }

         val bindDrawable: (Drawable) -> Unit =
itemImageView::setImageDrawable

         fun bindGalleryItem(item: GalleryItem)
{
            galleryItem = item
        }

        override fun onClick(view: View) {
                                val   intent   =
Intent(Intent.ACTION_VIEW,
galleryItem.photoPageUri)
            startActivity(intent)
        }
    }
    ...
}
```

Добавление ключевого слова inner в PhotoHolder позволяет получить доступ к свойствам и функциям внешнего класса. В этом случае из папки PhotoHolder вызывается файл Fragment.startActivity(Intent).

Затем свяжите PhotoHolder с GalleryItem в PhotoAdapter.onBindViewHolder(...).

Листинг 29.3. Связывание GalleryItem (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : VisibleFragment()
{
    ...
      private inner class PhotoAdapter(private
val galleryItems: List>GalleryItem>) :
            RecyclerView.Adapter<PhotoHolder>()
{
        ...
          override fun onBindViewHolder(holder:
PhotoHolder, position: Int) {
                              val galleryItem =
galleryItems[position]
            holder.bindGalleryItem(galleryItem)
                  val placeholder: Drawable =
ContextCompat.getDrawable(
                requireContext(),
                R.drawable.bill_up_close
            ) ?: ColorDrawable()
            holder.bindDrawable(placeholder)
            thumbnailDownloader.queueThumbnail(
holder, galleryItem.url)
        }
```

```
        }
        ...
    }
```

Запустите приложение PhotoGallery и нажмите на фотографии. На экране открывается браузер, в котором загружается страница выбранной вами фотографии (как в левой части рис. 29.1).

Сложный способ: WebView

Решение с использованием неявного интента для отображения веб-страницы просто и эффективно. Но что, если вы не хотите, чтобы ваше приложение открывало браузер?

На практике веб-контент чаще требуется отобразить прямо в activity вашего приложения. Допустим, вы хотите отобразить самостоятельно сгенерированную разметку HTML или просто обойтись без браузера. Справочная документация в приложениях часто реализуется в виде веб-страницы, чтобы ее было удобнее обновлять. Запуск браузера для просмотра справочных страниц выглядит непрофессионально и препятствует изменению поведения, а также интеграции веб-страницы в ваш пользовательский интерфейс.

Для представления веб-контента в пользовательском интерфейсе приложения используется класс `WebView`. Мы назвали этот способ более сложным, но на самом деле он очень прост (хотя по сравнению с неявными интентами все можно назвать сложным).

Нашим первым шагом станет создание новой activity и фрагмента для отображения `WebView`. Начнем, как обычно, с определения файла макета; присвойте ему имя `res/layout/fragment_photo_page.xml`. Назначьте `ConstraintLayout` макетом верхнего уровня. В визуальном

конструкторе перетащите WebView на ConstraintLayout как дочернее представление. (Виджет WebView находится в разделе **Widgets**.)

После добавления WebView добавьте к родителю ограничения для всех сторон:

- от верхней стороны WebView к верхней стороне родителя;
- от нижней стороны WebView к нижней стороне родителя;
- от левой стороны WebView к левой стороне родителя;
- от правой стороны WebView к правой стороне родителя.

Наконец, задайте высоте и ширине значения MatchConstraint и уменьшите величину всех полей до 0. Также присвойте WebView идентификатор web_view.

Возможно, у вас возникла мысль: «От ConstraintLayout нет никакой пользы», — и верно. Однако позднее в этой главе мы добавим ему дополнительного блеска.

Теперь займитесь настройкой фрагмента. Создайте PhotoPageFragment как подкласс класса VisibleFragment, созданного в предыдущей главе. Необходимо заполнить файл макета, выделить из него WebView и передать URL-адрес как аргумент фрагмента.

Листинг 29.4. Настройка фрагмента браузера (PhotoPageFragment.kt)

```
private const val ARG_URI = "photo_page_url"

class PhotoPageFragment : VisibleFragment() {
```

```
    private lateinit var uri: Uri
    private lateinit var webView: WebView

      override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)

        uri = arguments?.getParcelable(ARG_URI)
?: Uri.EMPTY
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                                  val    view    =
inflater.inflate(R.layout.fragment_photo_page,
container, false)

                                       webView    =
view.findViewById(R.id.web_view)

        return view
    }

    companion object {
                    fun  newInstance(uri:  Uri):
PhotoPageFragment {
            return PhotoPageFragment().apply {
```

```
                arguments = Bundle().apply {
                    putParcelable(ARG_URI, uri)
                }
            }
        }
    }
}
```

Пока это всего лишь заготовка — вскоре мы заполним ее кодом. А пока создайте класс-контейнер `PhotoPageActivity` на основе класса `Fragment`.

Листинг 29.5. Создание веб-activity (PhotoPageActivity.kt)

```
class PhotoPageActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_photo_
page)

        val fm = supportFragmentManager
                    val currentFragment =
fm.findFragmentById(R.id.fragment_container)

        if (currentFragment == null) {
                        val fragment =
PhotoPageFragment.newInstance(intent.data)
            fm.beginTransaction()
                .add(R.id.fragment_container,
fragment)
```

```
                .commit()
        }
    }

    companion object {
        fun newIntent(context: Context,
photoPageUri: Uri): Intent {
            return Intent(context,
PhotoPageActivity::class.java).apply {
                data = photoPageUri
            }
        }
    }
}
```

Создайте недостающий файл макета `res/layout/activity_photo_page.xml` и добавьте `FrameLayout` с идентификатором `FrameLayout`.

Листинг 29.6. Добавление макета activity (res/layout/activity_photo_page.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
```

Измените код `PhotoGalleryFragment`, чтобы вместо неявного интента запускалась новая activity.

Листинг 29.7. Переключение на запуск activity (PhotoGalleryFragment.kt)

```
class PhotoGalleryFragment : VisibleFragment()
{
    ...
    private inner class PhotoHolder(private val
itemImageView: ImageView)
                                                :
RecyclerView.ViewHolder(itemImageView),
            View.OnClickListener {
        ...
        override fun onClick(view: View) {
                                ~~val intent =~~
~~Intent(Intent.ACTION_VIEW,~~
~~galleryItem.photoPageUri)~~
            val intent = PhotoPageActivity
                    .newIntent(requireContext(),
galleryItem.photoPageUri)
            startActivity(intent)
        }
    }
    ...
}
```

Наконец, добавьте новую activity в манифест.

Листинг 29.8. Добавление activity в манифест (manifests/AndroidManifest.xml)

```
<manifest ... >
    ...
    <application
```

```
        ... >
                                        <activity
android:name=".PhotoGalleryActivity">
            ...
        </activity>
                                        <activity
android:name=".PhotoPageActivity"/>
                                        <receiver
android:name=".NotificationReceiver"
                    ... >
            ...
        </receiver>
    </application>

</manifest>
```

Запустите приложение PhotoGallery и коснитесь фотографии. На экране появится новая пустая activity.

Возьмемся за дело и заставим наш фрагмент делать что-то полезное. Чтобы виджет `WebView` успешно отображал страницу фотографии на сайте Flickr, необходимо выполнить три условия. Первое условие очевидно — нужно сообщить ему, какой URL-адрес необходимо загрузить.

Второе условие — включить поддержку JavaScript. По умолчанию она отключена. Постоянно держать ее включенной необязательно, но для Flickr она нужна. (Android Lint выдает предупреждение из-за потенциальной опасности межсайтовых сценарных атак, так что предупреждения Lint тоже нужно отключить — для этого следует пометить `onCreateView(...)` аннотацией `@SuppressLint ("SetJavaScriptEnabled")`.)

Наконец, необходимо предоставить реализацию интерфейса `WebViewClient`, используемого при визуализации событий

виджетом WebView. Мы рассмотрим этот класс после того, как вы введете код.

Листинг 29.9. Загрузка URL в WebView (PhotoPageFragment.kt)

```
class PhotoPageFragment : VisibleFragment() {
    ...
    @SuppressLint("SetJavaScriptEnabled")
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                                    val    view    =
inflater.inflate(R.layout.fragment_photo_page,
container, false)

                                          webView    =
view.findViewById(R.id.web_view)
            webView.settings.javaScriptEnabled =
true
        webView.webViewClient = WebViewClient()
        webView.loadUrl(uri.toString())

        return view
    }
    ...
}
```

Загрузка данных с URL-адреса должна происходить после настройки WebView, поэтому она выполняется в последнюю очередь. До этого мы включаем JavaScript, обращаясь к свойству

settings для получения экземпляра WebSettings, с последующей установкой WebSettings.javaScriptEnabled=true. Объект WebSettings — первый из трех механизмов настройки WebView. Он содержит различные свойства, которые можно задать в коде, например строку пользовательского агента и размер текста.

После этого реализация WebViewClient добавляется к WebView. Чтобы понять, для чего это нужно, сначала посмотрим, что произойдет без WebViewClient.

Новый URL-адрес может загружаться двумя разными способами: страница может приказать вам перейти по другому URL-адресу (перенаправление), или же вы можете щелкнуть на ссылке. Без WebViewClient виджет WebView предложит менеджеру activity найти подходящую activity для загрузки нового URL.

Но это совсем не то, что нам нужно. Многие сайты (включая страницы с фотографиями Flickr) при загрузке в браузере на телефоне немедленно перенаправляют пользователя на мобильную версию того же сайта. Нет особого смысла создавать собственное представление страницы, если все равно все кончится инициированием неявного интента.

С другой стороны, если вы предоставите собственную реализацию WebViewClient для своего виджета WebView, процесс выглядит иначе. Вместо того чтобы обращаться за помощью к менеджеру activity, виджет обращается к вашей реализации WebViewClient. А реализация WebViewClient по умолчанию говорит: «Загрузи URL самостоятельно!» И страница появится в вашем виджете WebView.

Запустите приложение PhotoGallery, и вы увидите WebView на экране (как справа на рис. 29.1).

Класс WebChromeClient

Раз уж мы занялись созданием собственной реализации `WebView`, давайте немного украсим ее, добавив представление заголовка и индикатор прогресса. Такие украшения и пользовательский интерфейс вне `WebView` называются *«хромом»* (не путайте с браузером Google Chrome). Снова откройте файл `fragment_photo_page.xml`.

Перетащите виджет `ProgressBar` на `ConstraintLayout` (как второй дочерний виджет). Используйте горизонтальную версию индикатора прогресса **ProgressBar(Horizontal)**. Удалите верхнее ограничение `WebView` и назначьте виджету фиксированную высоту (`Fixed`), чтобы вам было удобнее работать с маркерами ограничений.

После того как это будет сделано, создайте следующие дополнительные ограничения:

- от `ProgressBar` к верхней, правой и левой стороне его родителя;
- от верхней стороны `WebView` к нижней стороне `ProgressBar`.

Затем верните высоте `WebView` значение `MatchConstraint`, задайте высоте `ProgressBar` значение `wrap_content` и измените ширину `ProgressBar` на `MatchConstraint`.

Наконец, выделите `ProgressBar` и обратитесь к окну свойств. Присвойте свойству `visibility` значение `gone`, а `visibility` — значение `visible`. Присвойте виджету идентификатор `progress_bar`.

Примерный результат показан на рис. 29.2.

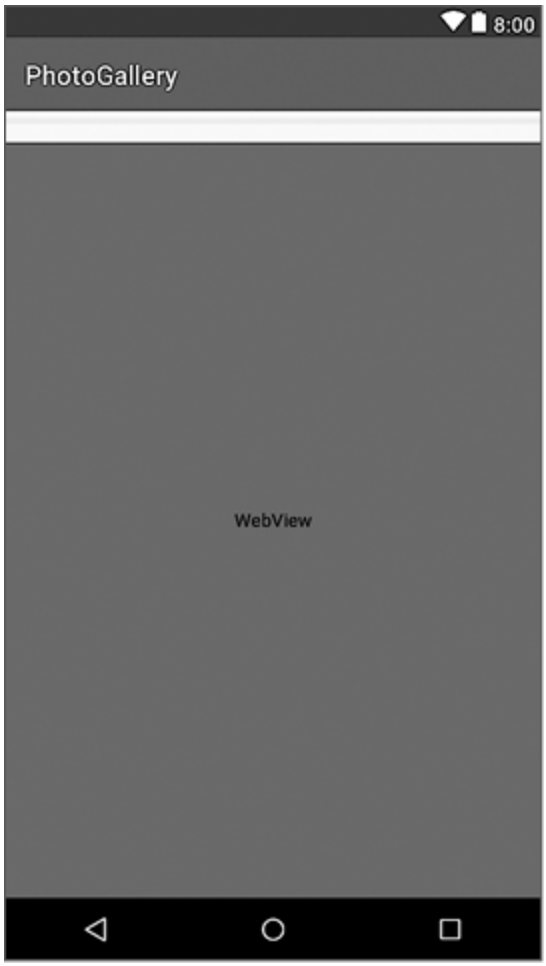


Рис. 29.2. Добавление индикатора прогресса

Чтобы добавить `ProgressBar`, нам понадобится вторая точка обратного вызова `WebView`: `WebChromeClient`. Если `WebViewClient` определяет интерфейс обработки событий визуализации, то `WebChromeClient` определяет событийный интерфейс обработки событий, которые должны изменять элементы «хрома» в браузере. К этой категории относятся предупреждения JavaScript, значки сайтов favicon, обновления прогресса загрузки и т.д.

Подключите его в функции `onCreateView(...)`.

Листинг 29.10. Использование WebChromeClient (PhotoPageFragment.kt)

```
class PhotoPageFragment : VisibleFragment() {

    private lateinit var uri: Uri
    private lateinit var webView: WebView
        private lateinit var progressBar: ProgressBar
    ...
    @SuppressLint("SetJavaScriptEnabled")
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
                                val view =
inflater.inflate(R.layout.fragment_photo_page,
container, false)

                                progressBar =
view.findViewById(R.id.progress_bar)
        progressBar.max = 100

                                webView =
view.findViewById(R.id.web_view)
          webView.settings.javaScriptEnabled =
true
           webView.webChromeClient = object :
WebChromeClient() {
                               override fun
onProgressChanged(webView:            WebView,
newProgress: Int) {
```

```
                if (newProgress == 100) {
                        progressBar.visibility =
View.GONE
                } else {
                        progressBar.visibility =
View.VISIBLE
                          progressBar.progress =
newProgress
                }
            }

             override fun onReceivedTitle(view:
WebView?, title: String?) {
                                    (activity as
AppCompatActivity).supportActionBar?.subtitle =
title
            }
        }
        webView.webViewClient = WebViewClient()
        webView.loadUrl(uri.toString())

        return view
    }
    ...
}
```

Обновления индикатора прогресса и заголовка имеют собственные функции обратного вызова, `onProgressChanged(WebView,Int)` и `onReceivedTitle(WebView,String)`. Информация о прогрессе, получаемая от функции `onProgressChanged (WebView,int)`, представляет собой целое число от 0 до 100.

Если результат равен 100, значит, загрузка страницы завершена, поэтому мы скрываем `ProgressBar`, задавая режим `View.GONE`.

Запустите приложение PhotoGallery и протестируйте внесенные изменения. Результат должен выглядеть так, как показано на рис. 29.3.

При нажатии на фотографии открывается `PhotoPageActivity`. Индикатор отображает информацию о ходе загрузки страниц, а на панели приложения появляется подзаголовок с текстом, полученным в `onReceivedTitle(...)`. После завершения загрузки индикатор прогресса исчезает.

Повороты в WebView

Поверните экран. Хотя приложение работает правильно, вы заметите, что `WebView` полностью перезагружает веб-страницу. Дело в том, что у `WebView` слишком много данных, чтобы сохранить их все в `onSaveInstanceState(...)`, и их приходится собирать с нуля каждый раз, когда они создаваются заново при повороте.

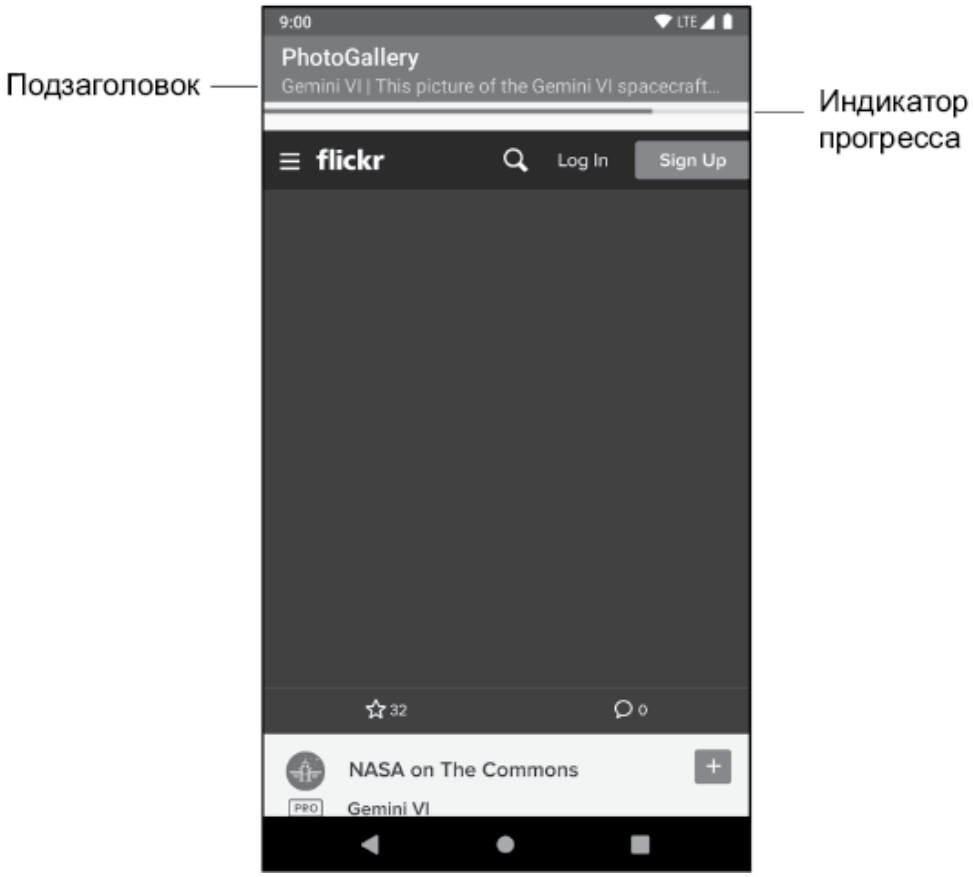


Рис. 29.3. Крутой WebView

Может показаться, что проблема проще всего решается удержанием `PhotoPageFragment`. Тем не менее это решение не работает, потому что `WebView` является частью иерархии представлений. Именно поэтому он уничтожается и создается заново при повороте.

Для таких классов (другой пример — `VideoView`) документация Android рекомендует позволить activity самой обрабатывать все изменения конфигурации. Это означает, что вместо уничтожения activity она просто перемещает свои представления для размещения по новым размерам экрана. В результате `WebView` не приходится заново загружать свои данные.

Чтобы заставить класс PhotoPageActivity обрабатывать свои изменения конфигурации, внесите следующие изменения в manifests/AndroidManifest.xml.

Листинг 29.11. Самостоятельный обработчик изменений конфигурации (manifests/AndroidManifest.xml)

```
<manifest ... >
    ...
    <activity android:name=".PhotoPageActivity"
        android:configChanges="keyboardHidden
|orientation|screenSize" />
    ...
</manifest>
```

Атрибут сообщает, что в случае изменения конфигурации из-за открытия или закрытия клавиатуры, изменения ориентации или размеров экрана (которое также происходит при переключении между книжной и альбомной ориентацией в Android после версии 3.2) activity должна обрабатывать изменения самостоятельно. На самом деле вам не нужно делать ничего особенного для обработки изменений конфигурации, поскольку представления автоматически меняют свой размер и положение под новый экран.

Вот и все. Попробуйте снова повернуть устройство; на этот раз все должно быть в ажуре.

Опасности при обработке изменений конфигурации

Решение получается настолько простым и эффективным, что у вас может возникнуть вопрос: почему бы не делать так всегда? Ведь жизнь разработчика стала бы намного проще. Тем не

менее самостоятельная обработка конфигурации — опасная привычка.

Во-первых, квалификаторы ресурсов перестают работать автоматически, и вам придется перезагружать представление вручную. Это сложнее, чем может показаться.

Во-вторых, при обработке изменений конфигурации в activity вы, скорее всего, не станете возиться с переопределением `Activity.onSavedInstanceState(...)` для сохранения временных состояний пользовательского интерфейса. Однако это все равно необходимо, даже если activity обрабатывает изменения конфигурации самостоятельно, потому что возможность уничтожения и повторного создания при нехватке памяти все равно остается. (Помните: activity может быть уничтожена системой в любой момент, когда она не находится в состоянии выполнения (см. рис. 4.9).)

Сравнение WebView и пользовательского интерфейса

Встроенный пользовательский интерфейс (без `WebView`) дает вам полный контроль над тем, как выглядит и ведет себя ваше приложение. Кроме того, этот интерфейс кажется пользователям более отзывчивым и последовательным. Но и в отображении веб-контента вместо развертывания собственного пользовательского интерфейса есть ряд преимуществ.

Отображение сайта Flickr в `WebView` позволяет намного быстрее интегрировать большие функции. Вам не придется думать о выгрузке описаний изображений, имен учетных записей пользователей или других метаданных фотографий для создания пользовательского интерфейса. Вы можете просто использовать то, что уже есть у Flickr.

Еще одним преимуществом отображения веб-содержимого является то, что оно может меняться, и обновлять приложение при этом не придется. Например, если вам нужно вывести политику конфиденциальности или условия предоставления услуг, вы можете просто отобразить веб-сайт вместо хранения документа в вашем приложении. Таким образом, любые изменения появятся сразу без обновления приложения.

Для любознательных: внедрение объектов JavaScript

Вы уже видели, как следует использовать `WebViewClient` и `WebChromeClient` для обработки некоторых событий, происходящих в `WebView`. Однако еще больше возможностей открывает внедрение произвольных объектов JavaScript в документ, содержащийся в виджете `WebView`. Откройте документацию по адресу **developer.android.com/reference/android/webkit/WebView.html** и прокрутите до описания функции `addJavascriptInterface(Object,String)`. В документации используются подписи методов Java, но помните, что `Object` — это то же самое, что `Any` в Kotlin. Эта функция позволяет внедрить в документ произвольный объект с заданным именем.

```
webView.addJavascriptInterface(object : Any() {
    @JavascriptInterface
    fun send(message: String) {
                Log.i(TAG, "Received message:
$message")
    }
}, "androidObject")
```

После этого объект используется следующим образом:

```
<input type="button" value="In WebView!"
     onClick="sendToAndroid('In Android land')"
/>

<script type="text/javascript">
    function sendToAndroid(message) {
        androidObject.send(message);
    }
</script>
```

В этом коде есть пара нетривиальных моментов. Во-первых, при вызове `send (String)` функция Kotlin не вызывается в основном потоке. Она вызывается в потоке, принадлежащем `WebView`. Итак, если вы хотите обновить пользовательский интерфейс Android, вам придется использовать `Handler` для возврата управления в основной поток.

Кроме того, набор поддерживаемых типов данных сильно ограничен. В вашем распоряжении `String`, основные примитивные типы... и все. Любой более сложный тип должен передаваться через `String`, обычно с преобразованием в JSON перед отправкой и последующим разбором при получении.

Начиная с API 17 (Jelly Bean 4.2) в JavaScript экспортируются только открытые функции с пометкой `@JavascriptInterface`. До этого были доступны все открытые функции в иерархии объектов.

В любом случае такое решение весьма рискованно — фактически вы разрешаете потенциально небезопасной веб-странице вмешиваться в работу вашей программы. Следовательно, его стоит применять только для принадлежащей вам разметки HTML или ограничиться предоставлением в высшей степени консервативного интерфейса.

Для любознательных: обновления WebView

Реализация **WebView** основана на проекте с открытым кодом Chromium. В ней используется то же ядро визуализации, что и в приложении Chrome для Android; это означает большее сходство внешнего вида и поведения страниц. (Впрочем, `WebView` не обладает всей функциональностью Chrome для Android. Хорошая сравнительная таблица доступна по адресу **developer.chrome.com/multidevice/webview/overview**.)

Переход на Chromium означал целый ряд интересных усовершенствований `WebView`, включая переход на новые веб-стандарты и обновленное ядро JavaScript. С точки зрения разработчика, одним из самых интересных новшеств является поддержка удаленной отладки `WebView` с использованием Chrome DevTools (которая включается вызовом функции `WebView.setWebContentsDebuggingEnabled()`).

Начиная с Lollipop (Android 5.0) Chromium-уровень `WebView` автоматически обновляется из магазина Google Play. Пользователям уже не приходится ждать новых выпусков Android для получения обновлений безопасности (и новой функциональности). В еще более поздних версиях, с версии Nougat (Android 7.0), Chromium-уровень `WebView` размещается непосредственно в APK-файле Chrome, что приводит к снижению затрат памяти и ресурсов. Приятно сознавать, что Google следит за своевременным обновлением компонентов `WebView`.

Упражнение. Пользовательские вкладки Chrome (еще один простой способ)

Мы уже рассмотрели два способа отображения веб-контента: вы можете либо запустить веб-браузер пользователя, либо

встроить контент в ваше приложение. Есть также третий вариант, который сочетает в себе то, что вы видели до сих пор.

Пользовательские вкладки Chrome (**developer.chrome.com/multidevice/android/customtabs**) позволяют вам запускать браузер Chrome прямо в вашем приложении. Вы можете настроить его внешний вид так, чтобы он выглядел как часть вашего приложения и создавалось ощущение, будто пользователь все еще в нем. На рис. 29.4 показан пример пользовательской вкладки. Видно, что результат выглядит как смесь Google Chrome и `PhotoPageActivity`.

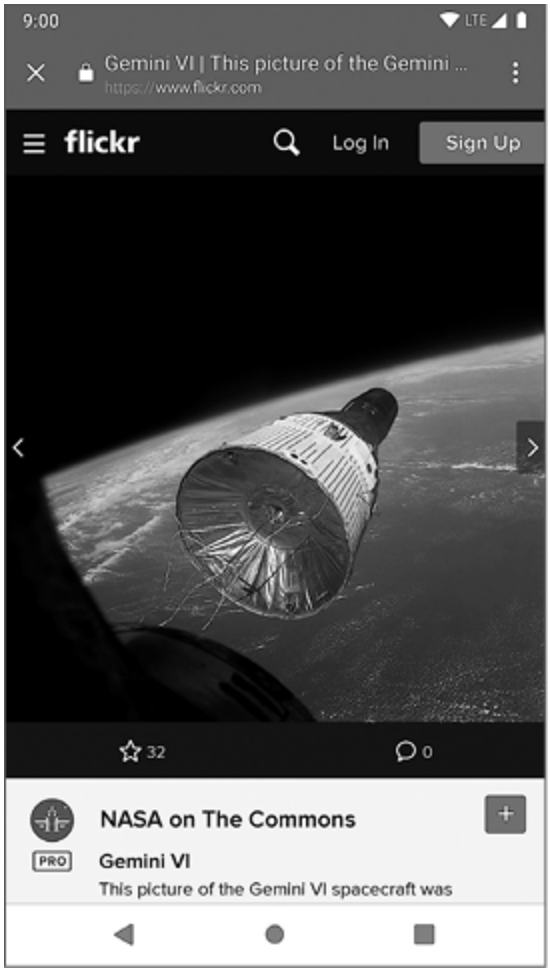


Рис. 29.4. Настраиваемая вкладка Chrome

Когда вы используете пользовательскую вкладку, ее поведение похоже на запуск Chrome. Экземпляр браузера даже имеет доступ к такой информации, как сохраненные пользователем пароли, к кэшу браузера и файлам cookie из самого браузера Chrome. Это означает, что если бы пользователь входил во Flickr в Chrome, то вход во Flickr выполнялся бы нормально в каждой пользовательской вкладке. При использовании `WebView` пользователю пришлось бы входить во Flickr как в Chrome, так и в PhotoGallery.

Недостатком использования пользовательских вкладок вместо `WebView` является то, что у вас будет мало способов контроля отображаемого содержимого. Например, вы не можете использовать пользовательские вкладки только в верхней половине экрана или добавлять кнопки навигации в нижнюю часть пользовательской вкладки.

Чтобы начать использовать пользовательские вкладки Chrome, необходимо добавить эту зависимость:

```
implementation 'androidx.browser:browser:1.0.0'
```

Затем вы можете запустить пользовательскую вкладку. Например, в PhotoGallery можно запустить пользовательскую вкладку вместо `PhotoPageActivity`:

```
class PhotoGalleryFragment : VisibleFragment()
{
    ...
    private inner class PhotoHolder(private val
itemImageView: ImageView)
                                                :
RecyclerView.ViewHolder(itemImageView),
            View.OnClickListener {
        ...
```

```
            override fun onClick(view: View) {
                                        val intent =
PhotoPageActivity
                            .newIntent(requireContext()
, galleryItem.photoPageUri)
                        startActivity(intent)

                        CustomTabsIntent.Builder()
                             .setToolbarColor(ContextCom
pat.getColor(
                                        requireContext(),
R.color.colorPrimary))
                            .setShowTitle(true)
                            .build()
                             .launchUrl(requireContext()
, galleryItem.photoPageUri)
                }
        }
        ...
}
```

В этом случае пользователь, нажимающий на фотографию, увидит пользовательскую закладку, как показано на рис. 29.4. (Если у пользователя не был установлен Chrome версии 45 или выше, то PhotoGallery будет использовать системный браузер. Результат будет такой же, как и при использовании неявного интента в начале этой главы.)

Упражнение. Использование кнопки «Назад» для работы с историей просмотра

Возможно, вы заметили, что после запуска `PhotoPageActivity` можно перемещаться по другим ссылкам в `WebView`. Однако сколько бы ссылок вы ни открывали, кнопка «Назад» всегда возвращает вас прямо к `PhotoGalleryActivity`. А если вы хотите, чтобы кнопка «Назад» работала с историей просмотра в `WebView`?

Реализуйте это поведение переопределением функции `Activity.onBackPressed()`. В этой функции для выполнения правильной операции используется комбинация функций `WebView` для работы с историей просмотра (`WebView.canGoBack()` и `WebView.goBack()`). Если история просмотра `WebView` не пуста, вернитесь к предыдущему элементу. В противном случае кнопка «Назад» должна работать как обычно — вызовите `super.onBackPressed()`.

30. Пользовательские представления и события касания

В этой главе мы займемся обработкой событий касания. Для этого мы создадим подкласс `View` с именем `BoxDrawingView`, который займет центральное место в новом проекте DragAndDraw. На этом представлении пользователь рисует прямоугольники, прикасаясь к экрану и перемещая палец. Результат выглядит примерно так, как показано на рис. 30.1.

Создание проекта DragAndDraw

Создайте новый проект с именем DragAndDraw. Выберите в поле минимального SDK уровень API 21 и создайте пустую activity. Присвойте ей имя `DragAndDrawActivity`.

Класс `DragAndDrawActivity` представляет собой подкласс `AppCompatActivity`, который заполняет обычный макет с одним фрагментом. Все отображение и обработка касаний реализуется в `BoxDrawingView`.

Создание пользовательского представления

Android предоставляет много превосходных стандартных представлений и виджетов, но иногда требуется создать нестандартное представление с визуальным оформлением, полностью уникальным для вашего приложения.

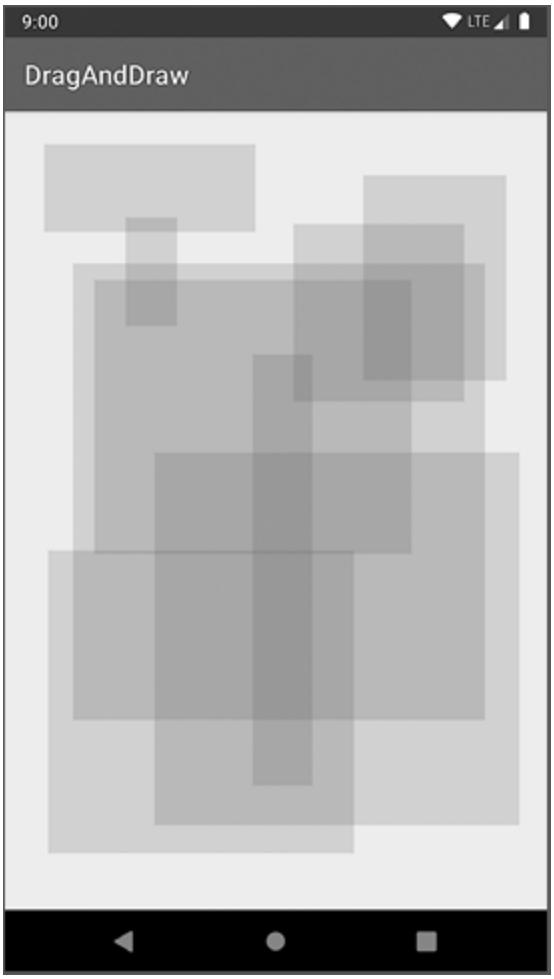


Рис. 30.1. Прямоугольники разных форм и размеров

Все многообразие нестандартных представлений можно условно разделить на две общие категории:

- *простые* представления — простое представление может быть устроено достаточно сложно; «простым» оно называется только потому, что не имеет дочерних представлений. Простое представление почти всегда также выполняет нестандартную прорисовку;

- *составные* представления — состоят из других объектов представлений. Составные представления обычно управляют дочерними представлениями, но не занимаются

своей прорисовкой. Вместо этого каждому дочернему представлению делегируется своя часть работы по прорисовке.

Создание нестандартного представления состоит из трех шагов:

1. Выберите суперкласс. Для простого нестандартного представления `View` предоставляет пустой «холст» для рисования, поэтому этот выбор наиболее распространен. Для составных нестандартных представлений выберите подходящий класс макета (например, `FrameLayout`).

2. Подклассируйте выбранный класс и переопределите как минимум один конструктор из суперкласса, или создайте собственный конструктор, вызывающий один из конструкторов суперкласса.

3. Переопределите другие ключевые функции для настройки поведения.

Создание класса BoxDrawingView

Класс `BoxDrawingView` относится к категории простых представлений и является прямым подклассом `View`.

Создайте класс `BoxDrawingView` и назначьте `View` его суперклассом. Добавьте в файл `BoxDrawingView.kt` конструктор, принимающий объект `Context` и допускающий `null`, и `AttributeSet` со значением по умолчанию `null`.

Листинг 30.1. Исходная реализация BoxDrawingView (BoxDrawingView.kt)

```
class BoxDrawingView(context: Context, attrs:
AttributeSet? = null) :
        View(context, attrs) {
}
```

Установка значения null по умолчанию для набора атрибутов фактически создает два конструктора для представления. Нужно именно два конструктора, так как ваше представление может создаваться в коде или из файла макета. Представление, экземпляр которого создается из файла макета, получает экземпляр AttributeSet, содержащий атрибуты XML, которые были указаны в файле XML. Даже если вы не планируете использовать оба конструктора, стоит предусмотреть их оба.

Затем измените код в файле макета res/layout/activity_drag_and_draw.xml, чтобы в нем использовалось новое представление.

Листинг 30.2. Включение BoxDrawingView в макет (res/layout/activity_drag_and_draw.xml)

```
    <androidx.constraintlayout.widget.Constrain
tLayout
        xmlns:android="http://schemas.android.c
om/apk/res/android"
        xmlns:app="http://schemas.android.com/a
pk/res-auto"
        xmlns:tools="http://schemas.android.com
/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context="com.bignerdranch.android
.draganddraw.DragAndDrawActivity">
```

```
    </androidx.constraintlayout.widget.ConstraintLayout>
    <com.bignerdranch.android.draganddraw.BoxDrawingView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
```

Чтобы заполнитель макетов нашел класс `BoxDrawingView`, вы должны использовать полностью уточненное имя класса. Заполнитель просматривает файл макета, создавая экземпляры `View`. Если имя элемента будет неполным, то заполнитель ищет класс с указанным именем в пакетах `android.view` и `android.widget`. Если класс находится в другом месте, заполнитель его не найдет, и в приложении произойдет сбой.

По этой причине для классов, не входящих в `android.view` и `android.widget`, необходимо всегда задавать полностью уточненное имя.

Запустите приложение DragAndDraw и убедитесь в том, что настройка была выполнена правильно. Правда, пока на экране нет ничего, кроме пустого представления (рис. 30.2).

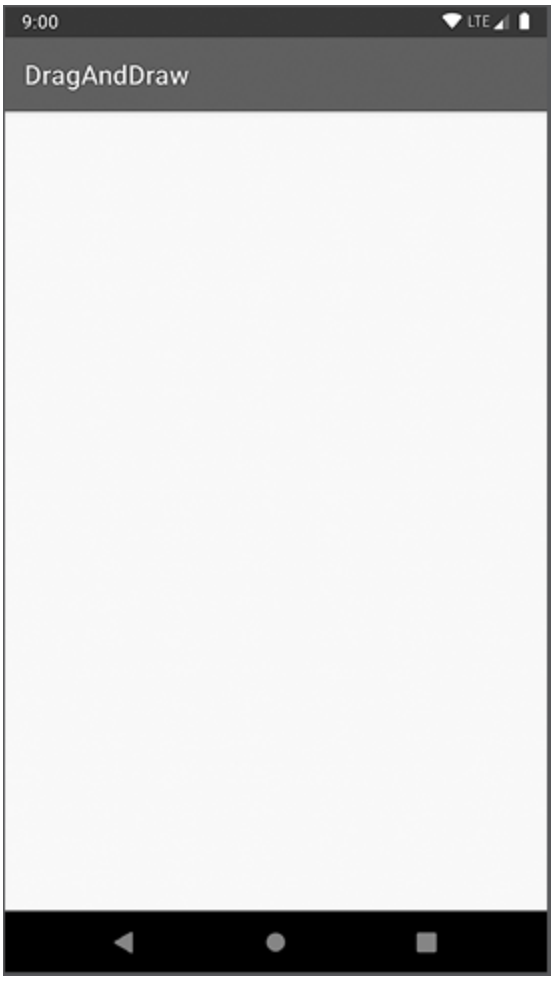


Рис. 30.2. BoxDrawingView без прямоугольников

На следующем этапе мы научим `BoxDrawingView` прослушивать события касания и использовать содержащуюся в них информацию для рисования прямоугольников на экране.

Обработка событий касания

Для прослушивания событий касания можно назначить слушателя события при помощи следующей функции класса `View`:

```
fun setOnTouchListener(l: View.OnTouchListener)
```

Этот функция работает почти так же, как `setOnClickListener(View.OnClickListener)`. Вы предоставляете реализацию `View.OnTouchListener`, а слушатель вызывается каждый раз, когда происходит событие касания.

Но поскольку мы подклассируем `View`, можно пойти по сокращенному пути и переопределить следующую функцию класса `View`:

```
override fun onTouchEvent(event: MotionEvent): Boolean
```

Этот функция получает экземпляр `MotionEvent` — класса, описывающего событие касания, включая его позицию и действие. Действие описывает стадию события.

Константы действий	Описание
ACTION_DOWN	Пользователь прикоснулся к экрану
ACTION_MOVE	Пользователь перемещает палец по экрану
ACTION_UP	Пользователь отводит палец от экрана
ACTION_CANCEL	Родительское представление перехватило событие касания

В своей реализации `onTouchEvent(MotionEvent)` для проверки действия можете воспользоваться следующей функцией класса `MotionEvent`:

```
final fun getAction(): Int
```

Добавьте в файл `BoxDrawingView.kt` тег для журнала и реализацию `onTouchEvent(MotionEvent)`, которая регистрирует в журнале информацию о каждом из четырех разных действий.

Листинг 30.3. Реализация BoxDrawingView (BoxDrawingView.kt)

```
private const val TAG = "BoxDrawingView"

class BoxDrawingView(context: Context, attrs:
AttributeSet? = null) :
        View(context, attrs) {

            override fun onTouchEvent(event:
MotionEvent): Boolean {
        val current = PointF(event.x, event.y)
        var action = ""
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                action = "ACTION_DOWN"
            }
            MotionEvent.ACTION_MOVE -> {
                action = "ACTION_MOVE"
            }
            MotionEvent.ACTION_UP -> {
                action = "ACTION_UP"
            }
            MotionEvent.ACTION_CANCEL -> {
                action = "ACTION_CANCEL"
            }
        }

         Log.i(TAG, "$action at x=${current.x},
y=${current.y}")

        return true
    }
}
```

Обратите внимание: координаты X и Y упаковываются в объекте `PointF`. В оставшейся части главы эти два значения обычно будут передаваться вместе. `PointF` — предоставленный Android класс-контейнер, который решает эту задачу за вас.

Запустите приложение DragAndDraw и откройте панель **LogCat**. Прикоснитесь к экрану и проведите пальцем (в эмуляторе перетащите). Вы увидите в журнале сообщения с координатами X и Y каждого действия касания, полученного `BoxDrawingView`.

Отслеживание событий перемещения

Класс `BoxDrawingView` должен рисовать прямоугольники, а не регистрировать координаты. Для этого необходимо решить ряд задач.

Прежде всего для определения прямоугольника нам понадобятся две точки: базовая (в которой было сделано исходное касание) и текущая (в которой находится палец).

Следовательно, для определения прямоугольника необходимо отслеживать данные от нескольких событий `MotionEvent`. Данные будут храниться в объекте `Box`.

Создайте класс `Box` для хранения данных, определяющих прямоугольник.

Листинг 30.4. Класс Box (Box.kt)

```
class Box(val start: PointF) {

    var end: PointF = start

    val left: Float
```

```
            get() = Math.min(start.x, end.x)

        val right: Float
            get() = Math.max(start.x, end.x)

        val top: Float
            get() = Math.min(start.y, end.y)

        val bottom: Float
            get() = Math.max(start.y, end.y)

    }
```

Когда пользователь прикасается к `BoxDrawingView`, новый объект `Box` создается и включается в массив существующих прямоугольников (рис. 30.3).

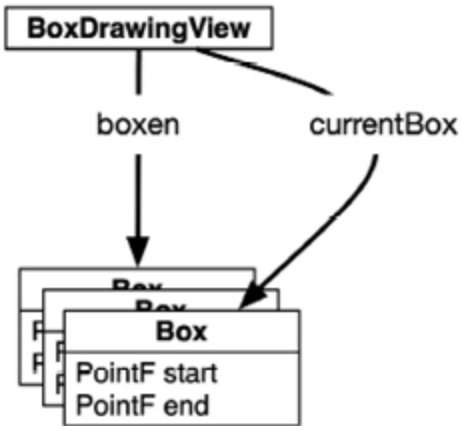


Рис. 30.3. Объекты в DragAndDraw

Добавьте в `BoxDrawingView` код, использующий новый объект `Box` для отслеживания текущего состояния рисования.

Листинг 30.5. Добавление функций жизненного цикла событий касания (BoxDrawingView.kt)

```
class BoxDrawingView(context: Context, attrs:
AttributeSet? = null) :
        View(context, attrs) {

    private var currentBox: Box? = null
    private val boxen = mutableListOf<Box>()

           override fun onTouchEvent(event:
MotionEvent): Boolean {
        val current = PointF(event.x, event.y)
        var action = ""
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                action = "ACTION_DOWN"
                // Сбросить состояние объекта
                 currentBox = Box(current).also
{
                    boxen.add(it)
                }
            }
            MotionEvent.ACTION_MOVE -> {
                action = "ACTION_MOVE"
                updateCurrentBox(current)
            }
            MotionEvent.ACTION_UP -> {
                action = "ACTION_UP"
                updateCurrentBox(current)
                currentBox = null
            }
```

```
                MotionEvent.ACTION_CANCEL -> {
                    action = "ACTION_CANCEL"
                    currentBox = null
                }
            }

            Log.i(TAG, "$action at x=${current.x},
y=${current.y}")

            return true
        }

        private fun updateCurrentBox(current:
PointF) {
            currentBox?.let {
                it.end = current
                invalidate()
            }
        }
    }
```

При каждом получении события ACTION_DOWN в поле currentBox сохраняется новый объект Box с базовой точкой, соответствующей позиции события. Этот объект Box добавляется в массив прямоугольников (в следующем разделе, когда мы займемся прорисовкой, BoxDrawingView будет выводить каждый объект Box из массива).

В процессе перемещения пальца по экрану приложение обновляет currentBox.end. Затем, когда касание отменяется или палец не касается экрана, поле currentBox обнуляется для

завершения операции. Объект `Box` завершен; он сохранен в массиве и уже не будет обновляться событиями перемещения.

Обратите внимание на вызов `invalidate()` в функции `updateCurrentBox()`. Он заставляет `BoxDrawingView` перерисовать себя, чтобы пользователь видел прямоугольник в процессе перетаскивания. Мы подошли к следующему шагу: рисованию прямоугольников на экране.

Рендеринг внутри onDraw(Canvas)

При запуске приложения все его представления *недействительны* (invalid). Это означает, что они ничего не вывели на экран. Для исправления ситуации Android вызывает функцию `draw()` объекта `View` верхнего уровня. В результате представление перерисовывает себя, что заставляет его потомков делать то же самое. Затем потомки этих потомков перерисовывают себя и так далее вниз по иерархии. Когда все представления в иерархии перерисуют себя, объект `View` верхнего уровня перестает быть недействительным.

Вы также можете вручную указать, что представление недействительно, даже если в данный момент оно находится на экране. Это заставит систему вывести представление заново с учетом изменений. `BoxDrawingView` отмечается как недействительный каждый раз, когда пользователь создает новое окно или изменяет его размер, перемещая палец. В этом случае пользователь сможет видеть прямоугольники в момент их создания.

Чтобы вмешаться в процесс прорисовки, следует переопределить следующую функцию `View`:

```
protected fun onDraw(canvas: Canvas)
```

Вызов invalidate(), выполняемый в ответ на действие ACTION_MOVE в onTouchEvent (MotionEvent), снова делает объект BoxDrawingView недействительным. Это заставляет его перерисовать себя и приводит к повторному вызову функции onDraw(Canvas).

Обратите внимание на параметр Canvas. Canvas и Paint — два главных класса, используемых при рисовании в Android.

- Класс Canvas содержит все выполняемые операции графического вывода. Функции, вызываемые для объекта Canvas, определяют, где и что выводится — линия, круг, слово или прямоугольник.

- Класс Paint определяет, как будут выполняться эти операции. Функции, вызываемые для объекта Paint, определяют характеристики вывода: должны ли фигуры заполняться, каким шрифтом должен выводиться текст, каким цветом должны выводиться линии и т.д.

В файле BoxDrawingView.kt создайте два объекта Paint при инициализации BoxDrawingView.

Листинг 30.6. Создание объектов Paint (BoxDrawingView.kt)

```
class BoxDrawingView(context: Context, attrs:
AttributeSet? = null) :
        View(context, attrs) {

    private var currentBox: Box? = null
    private val boxen = mutableListOf<Box>()
    private val boxPaint = Paint().apply {
        color = 0x22ff0000.toInt()
```

```
    }
    private val backgroundPaint = Paint().apply {
        color = 0xfff8efe0.toInt()
    }
    ...
}
```

После создания объектов `Paint` можно переходить к рисованию прямоугольников на экране.

Листинг 30.7. Переопределение onDraw(Canvas) (BoxDrawingView.kt)

```
class BoxDrawingView(context: Context, attrs:
AttributeSet? = null) :
        View(context, attrs)
    ...
    override fun onDraw(canvas: Canvas) {
        // Заполнение фона
        canvas.drawPaint(backgroundPaint)

        boxen.forEach { box ->
            canvas.drawRect(box.left, box.top,
box.right, box.bottom, boxPaint)
        }
    }
}
```

Первая часть кода тривиальна: используя серовато-белый цвет, мы заполняем «холст» задним фоном для вывода прямоугольников.

Затем для каждого прямоугольника в списке мы определяем значения `left`, `right`, `top` и `bottom` по двум точкам. Значения `left` и `top` будут минимальными, а `bottom` и `right` — максимальными.

После вычисления параметров вызов функции `Canvas.drawRect(...)` рисует красный прямоугольник на экране.

Запустите приложение DragAndDraw и нарисуйте несколько прямоугольников (рис. 30.4).

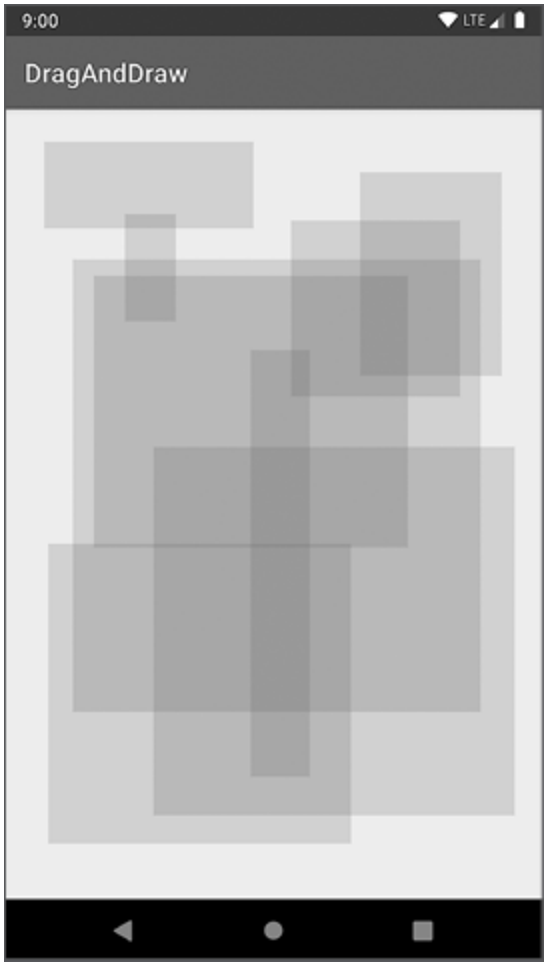


Рис. 30.4. Приложение с нарисованными прямоугольниками

Мы создали представление, которое обрабатывает свои события касания и выполняет прорисовку.

Для любознательных: GestureDetector

Еще один вариант обработки событий касания — использование объекта GestureDetector. Вместо того чтобы добавлять логику обнаружения таких событий, как смещение или смахивание, в GestureDetector есть слушатели, которые берут тяжелую работу на себя и уведомляют вас о наступлении определенного события. В реализации GestureDetector.OnGestureListener есть функции для таких событий, как длительные нажатия, смахивание и прокрутка. Существует также слушатель GestureDetector.OnDoubleTapListener, который будет срабатывать при обнаружении двойного нажатия. Во многих случаях переопределение функции onTouch не требуется, поэтому использование GestureDetector будет отличным вариантом.

Упражнение. Сохранение состояния

Подумайте, как обеспечить сохранение состояния прямоугольников при изменении ориентации из View. В этом вам могут помочь следующие функции View:

```
protected fun onSaveInstanceState(): Parcelable
protected fun onRestoreInstanceState(state: Parcelable)
```

Эти функции работают не так, как функция onSaveInstanceState(Bundle) классов Activity и Fragment. Во-первых, они вызываются только в том случае, если View присвоен идентификатор. Во-вторых, вместо объекта Bundle они возвращают и обрабатывают объект, реализующий интерфейс Parcelable.

Мы рекомендуем использовать `Bundle` в качестве `Parcelable`, вместо того чтобы писать реализацию `Parcelable` самостоятельно. (Реализация интерфейса `Parcelable` весьма сложна. Лучше избегать ее там, где это возможно.)

Наконец, вы также должны поддерживать сохраненное состояние родителя `BoxDrawingView` класса `View`. Сохраните результат `super.onSaveInstanceState()` в новом объекте `Bundle` и передайте его суперклассу при вызове `super.onRestoreInstanceState(Parcelable)`.

Упражнение. Повороты прямоугольников

Еще одно, более сложное упражнение: реализуйте возможность вращения прямоугольников вторым пальцем. Для этого вам потребуется отслеживать операции с несколькими указателями в коде обработки `MotionEvent`. Также придется отрабатывать повороты «холстов».

При работе с множественными касаниями вам понадобятся:

- *индекс указателя* — сообщает, к какому указателю в текущем наборе относится событие;
- *идентификатор указателя* — обеспечивает однозначную идентификацию конкретного пальца в жесте.

Индекс указателя может изменяться, но идентификатор остается неизменным.

За дополнительной информацией обращайтесь к документации по следующим функциям `MotionEvent`:

```
final fun getActionMasked(): Int
```

```
final fun getActionIndex(): Int
final fun getPointerId(pointerIndex: Int): Int
final fun getX(pointerIndex: Int): Float
final fun getY(pointerIndex: Int): Float
```

Также посмотрите документацию по константам `ACTION_POINTER_UP` и `ACTION_POINTER_DOWN`.

Упражнение. Поддержка специальных возможностей

Встроенные виджеты поддерживают такие варианты доступа, как TalkBack и Switch Access. Создание собственных виджетов возлагает на вас ответственность за доступность вашего приложения. В качестве последнего упражнения в этой главе добавьте реализацию описания `BoxDrawingView` с помощью TalkBack для пользователей со слабым зрением.

Сделать это можно несколькими способами. Вы можете предоставить общую сводку представления и сказать пользователю, какая часть представления закрыта прямоугольниками. Кроме того, вы можете сделать каждый прямоугольник доступным элементом и заставить его описать пользователю свое местоположение на экране. Обратитесь к главе 18 для получения дополнительной информации о том, как сделать ваши приложения доступными.

31. Анимация свойств

Чтобы приложение было работоспособным, достаточно правильно написать код. Но чтобы работа с приложением была настоящим удовольствием, этого недостаточно. Приложение должно восприниматься как реальное, физическое явление, происходящее на экране телефона или планшета.

Как известно, реальные явления двигаются. Чтобы элементы пользовательского интерфейса двигались, к ним применяется *анимация*.

В этой главе мы напишем приложение, которое изображает сцену с солнцем в небе. При нажатии солнце опускается за горизонт, а небо окрашивается в закатный цвет.

Построение сцены

Все начинается с построения сцены, к которой будет применяться анимация. Создайте новый проект с именем Sunset. Убедитесь в том, что `minSdkVersion` присвоено значение 21. Задайте шаблон activity и включите артефакты AndroidX.

Закат у моря полон красок, поэтому для удобства мы присвоим названия некоторым цветам. Добавьте в папку `res/values` файл `colors.xml` и включите в него следующие значения.

Листинг 31.1. Добавление закатных цветов (res/values/colors.xml)

```
<resources>
    <color name="colorPrimary">#008577</color>
```

```
                                                <color
name="colorPrimaryDark">#00574B</color>
    <color name="colorAccent">#D81B60</color>

    <color name="bright_sun">#fcfcb7</color>
    <color name="blue_sky">#1e7ac7</color>
    <color name="sunset_sky">#ec8100</color>
    <color name="night_sky">#05192e</color>
    <color name="sea">#224869</color>
</resources>
```

Прямоугольные представления неплохо подойдут для моря и неба. Однако вряд ли кто-нибудь оценит квадратное солнце, какие бы доводы вы ни выдвигали в пользу технической простоты такого решения. Добавьте в папку `res/drawable/` круглый графический объект `sun.xml`.

Листинг 31.2. Добавление графического XML-объекта для солнца (res/values/colors.xml)

```
<shape
xmlns:android="http://schemas.android.com/apk/res/android"
        android:shape="oval">
    <solid android:color="@color/bright_sun" />
</shape>
```

Если вывести эллипс в квадратном представлении, получится круг. Зрители одобрительно кивнут при виде столь убедительной имитации.

Вся сцена будет построена в файле макета. Откройте файл `res/layout/activity_main.xml`, удалите его содержимое и

добавьте показанное ниже.

Листинг 31.3. Создание макета (res/layout/activity_main.xml)

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/r
es/android"
              android:id="@+id/scene"
              android:orientation="vertical"
              android:layout_width="match_parent
"
              android:layout_height="match_paren
t">
    <FrameLayout
            android:id="@+id/sky"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="0.61"
            android:background="@color/blue_sky
">
        <ImageView
                android:id="@+id/sun"
                android:layout_width="100dp"
                android:layout_height="100dp"
                android:layout_gravity="center"
                android:src="@drawable/sun" />
    </FrameLayout>
    <View
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="0.39"
            android:background="@color/sea" />
```

```
</LinearLayout>
```

Проверьте результат. У вас должна получиться дневная сцена с солнцем в синем небе над темно-синим морем. Прежде чем двигаться дальше, запустите Sunset и убедитесь в том, что все связи созданы правильно. Приложение должно выглядеть так, как показано на рис. 31.1.

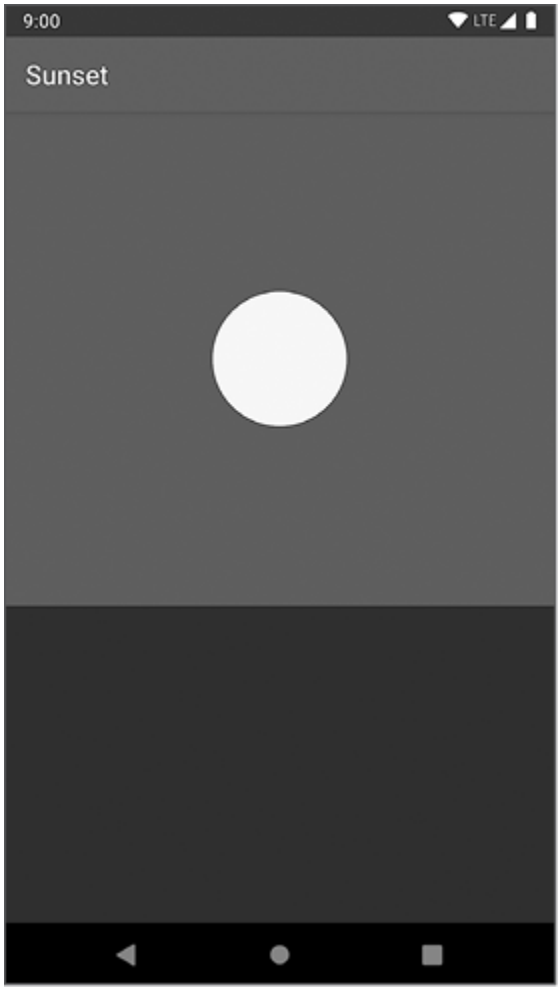


Рис. 31.1. Перед закатом

Простая анимация свойств

Итак, сцена подготовлена; теперь нужно привести ее составляющие в движение. Анимация будет использована для перемещения солнца под линию горизонта.

Но прежде чем браться за анимацию, стоит подготовить кое-какие данные во фрагменте. В функции `onCreate(...)` занесите пару представлений в поля `MainActivity`.

Листинг 31.4. Получение ссылок на представления (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    private lateinit var sceneView: View
    private lateinit var sunView: View
    private lateinit var skyView: View

      override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        sceneView = findViewById(R.id.scene)
        sunView = findViewById(R.id.sun)
        skyView = findViewById(R.id.sky)
    }
}
```

После завершения подготовки можно переходить к написанию кода анимации. План выглядит следующим образом: `sunView` плавно перемещается так, чтобы верхний край представления совпал с верхним краем моря. Для этого к

позиции верхнего края sunView будет применен сдвиг до нижнего края родителя.

Причина, по которой представление солнца движется сзади моря, не сразу может быть очевидна. Это связано с порядком отображения представлений. Представления отображаются в том порядке, в котором они объявлены в макете. Представления, объявленные позже, рисуются поверх тех, что находятся дальше вверх. В этом случае, так как солнце объявлено раньше моря, оно находится за ним. Когда солнце будет проходить мимо моря, оно будет как бы «заходить» за него.

Прежде всего следует определить начальное и конечное состояния анимации. Этот первый шаг будет реализован в новой функции startAnimation().

Листинг 31.5. Получение верхних координат представлений (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    ...
     override fun onCreate(savedInstanceState:
Bundle?) {
        ...
    }

    private fun startAnimation() {
        val sunYStart = sunView.top.toFloat()
        val sunYEnd = skyView.height.toFloat()
    }
}
```

Свойство `top` является одним из четырех свойств представления, которые возвращают *локальный макет* представления: это свойства `top`, `bottom`, `right` и `left`. Под макетом понимается прямоугольное окошко представления, которое задается этими четырьмя свойствами. Локальный макет представления определяет положение и размер представления по отношению к его родителю, определенные в момент, когда оно было установлено.

Можно изменить положение представления на экране, изменив эти значения, но делать так не рекомендуется. Эти значения сбрасываются всякий раз при прохождении макета, поэтому они, как правило, не сохраняются.

В любом случае анимация будет начинаться от верха текущего положения представления и заканчиваться в состоянии, при котором верх находится у нижнего края родителя `sunView`, то есть `skyView`. Для перехода в новое состояние потребуется смещение на величину, равную высоте `skyView`, которая может быть определена вызовом `height.toFloat()`. Значение свойства `height` представлено результатом вычитания значения `top` из `bottom`.

Теперь, когда вы знаете, где должна начинаться и завершаться анимация, создайте и запустите экземпляр `ObjectAnimator` для ее выполнения.

Листинг 31.6. Создание анимации солнца (MainActivity.kt)

```
private fun startAnimation() {
    val sunYStart = sunView.top.toFloat()
    val sunYEnd = skyView.height.toFloat()

    val heightAnimator = ObjectAnimator
```

```
                .ofFloat(sunView, "y", sunYStart,
sunYEnd)
        .setDuration(3000)

    heightAnimator.start()
}
```

О работе `ObjectAnimator` мы поговорим буквально через мгновение. Пока что подключите функцию `startAnimation()`, чтобы она выполнялась каждый раз, когда пользователь касается любой точки сцены.

Листинг 31.7. Запуск анимации по нажатию (MainActivity.kt)

```
override fun onCreate(savedInstanceState:
Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    sceneView = findViewById(R.id.scene)
    sunView = findViewById(R.id.sun)
    skyView = findViewById(R.id.sky)

    sceneView.setOnClickListener {
        startAnimation()
    }
}
```

Запустите приложение Sunset и коснитесь любой точки сцены, чтобы запустить анимацию (рис. 31.2).

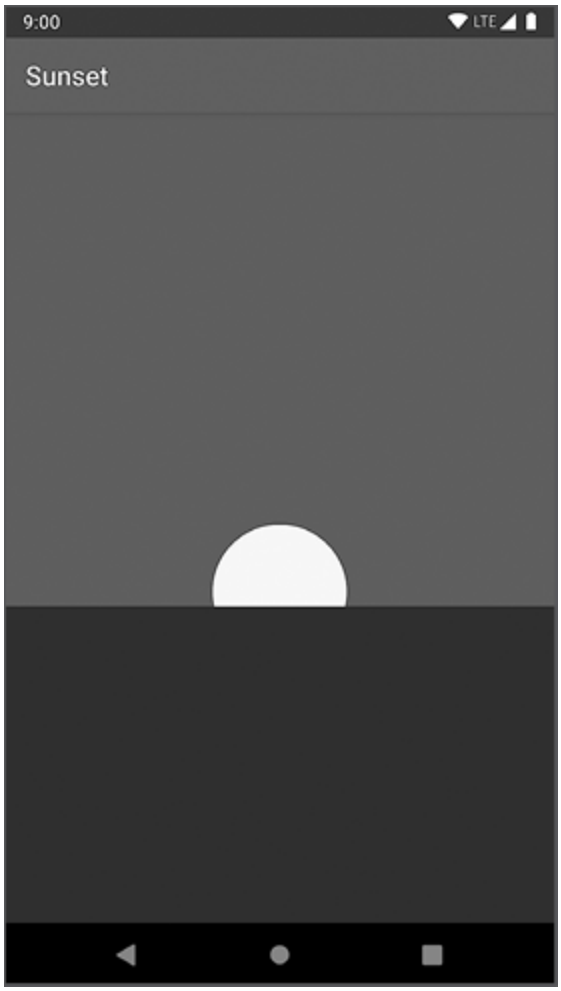


Рис. 31.2. Заход солнца

Вы увидите, как солнце опускается за горизонт.

Как же работает это решение? `ObjectAnimator` называется *аниматором свойства*. Ничего не зная о том, как перемещать представление по экрану, аниматор свойства многократно вызывает сеттеры свойства с разными значениями.

Объект `ObjectAnimator` создается вызовом функции `ObjectAnimator.ofFloat (sunView,"y",0,1)`. При запуске `ObjectAnimator` функция `sunView.setY(float)` многократно вызывается с постепенно увеличивающимися значениями, начиная с 0:

```
sunView.setY(0)
```

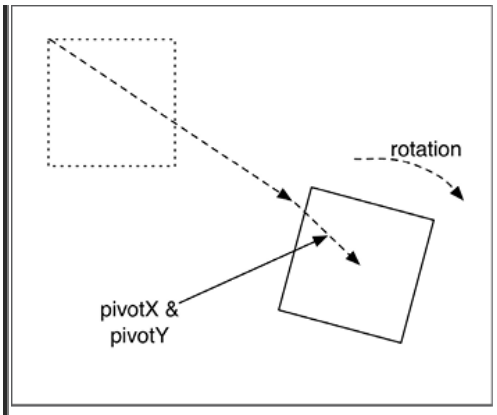
```
sunView.setY(0.02)
sunView.setY(0.04)
sunView.setY(0.06)
sunView.setY(0.08)
...
```

...и так далее, пока не будет вызвана функция `sunView.setY(1)`. Процесс вычисления значений между начальной и конечной точками называется *интерполяцией*. Между каждой интерполированной парой проходит небольшой промежуток времени; так создается иллюзия перемещения представления.

Свойства преобразований

Аниматоры свойств весьма удобны, но, пользуясь только ими, было бы невозможно организовать анимацию представлений настолько легко, как мы это сделали. Современная анимация свойств в Android работает в сочетании со *свойствами преобразований*.

С нашим представлением связывается прямоугольник локального макета, позиция и размер которого назначаются в процессе макета. Вы можете перемещать представление, задавая дополнительные свойства представления, называемые свойствами преобразований (transformation properties). В вашем распоряжении три свойства для выполнения поворотов (`rotation`, `pivotX` и `pivotY`, рис. 31.3), два свойства для масштабирования представления по вертикали и по горизонтали (`scaleX` и `scaleY`, рис. 31.4), два свойства для сдвига представлений (`translationX` и `translationY`, рис. 31.5).



поворот

Рис. 31.3. Поворот представления

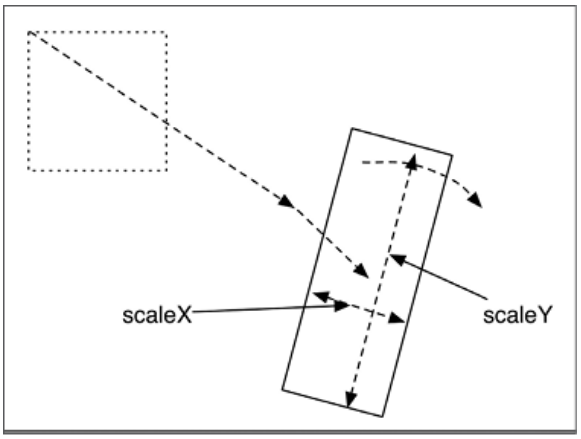


Рис. 31.4. Масштабирование представления

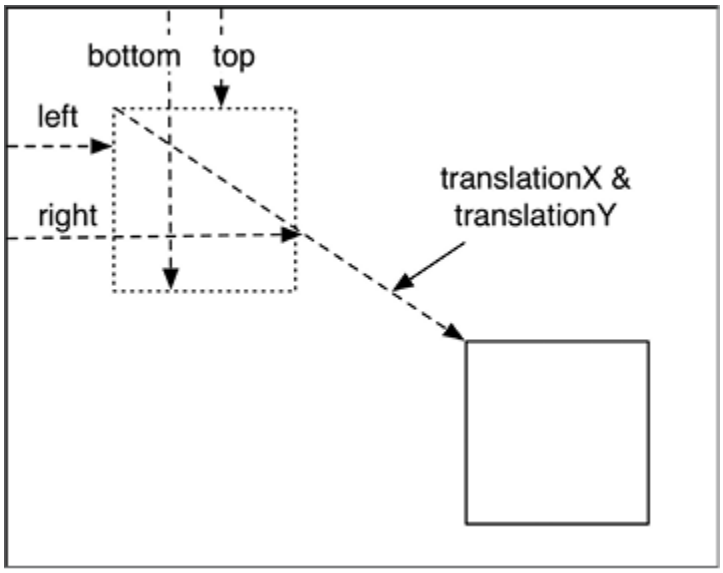


Рис. 31.5. Сдвиг представления

У всех таких свойств существуют геттеры и сеттеры. Например, для получения текущего значения `translationX` можно использовать `view.translationX`, а если вы хотите

задать свойству новое значение, вызовите `view.translationX=Float`.

Как работает свойство `y`? Вспомогательные свойства `x` и `y` созданы на базе локальных координат макета и свойств преобразований. С ними вы можете писать код, который означает: «Разместить это представление в точке с такими координатами X и Y». Во внутренней реализации эти свойства изменяют `translationX` или `translationY`, чтобы разместить представление в нужной точке. Это означает, что вызов функции `sunView.setY(50)` в действительности эквивалентен:

```
sunView.translationY = 50 - sunView.top
```

Выбор интерполятора

Наша анимация хорошо смотрится, но выполняется слишком резко. Если солнце неподвижно, ему понадобится некоторое время для того, чтобы набрать скорость. Чтобы смоделировать это ускорение, достаточно воспользоваться объектом `TimeInterpolator`. Класс `TimeInterpolator` решает всего одну задачу: он изменяет способ перехода анимации от точки A к точке B.

Добавьте в `startAnimation()` строку кода, которая обеспечит небольшое ускорение солнца в начале анимации с использованием объекта `AccelerateInterpolator`.

Листинг 31.8. Добавление ускорения (MainActivity.kt)

```
private fun startAnimation() {
    val sunYStart = sunView.top.toFloat()
    val sunYEnd = skyView.height.toFloat()
```

```
        val heightAnimator = ObjectAnimator
                .ofFloat(sunView, "y", sunYStart,
    sunYEnd)
            .setDuration(3000)
                    heightAnimator.interpolator =
    AccelerateInterpolator()

        heightAnimator.start()
    }
```

Снова запустите приложение Sunset и коснитесь экрана, чтобы просмотреть анимацию. Солнце начинает медленно двигаться и ускоряется при подходе к горизонту.

Существует много разных видов движения, которые могут использоваться в приложениях, поэтому существует много разновидностей `TimeInterpolator`. Полный список всех интерполяторов, входящих в поставку Android, приведен в разделе Known Indirect Subclasses справочной документации `TimeInterpolator`.

Изменение цвета

Теперь давайте окрасим небо в закатный цвет. В функции `onCreateView(...)` загрузите все цвета, определенные в файле `colors.xml`, в переменные экземпляров.

Листинг 31.9. Загрузка закатных цветов (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {

    private lateinit var sceneView: View
    private lateinit var sunView: View
    private lateinit var skyView: View
```

```
    private val blueSkyColor: Int by lazy {
                    ContextCompat.getColor(this,
R.color.blue_sky)
    }
    private val sunsetSkyColor: Int by lazy {
                    ContextCompat.getColor(this,
R.color.sunset_sky)
    }
    private val nightSkyColor: Int by lazy {
                    ContextCompat.getColor(this,
R.color.night_sky)
    }
    ...
}
```

Включите в `startAnimation()` дополнительную анимацию цвета неба от `blueSkyColor` до `sunsetSkyColor`.

Листинг 31.10. Анимация цвета неба (MainActivity.kt)

```
private fun startAnimation() {
    val sunYStart = sunView.top.toFloat()
    val sunYEnd = skyView.height.toFloat()

    val heightAnimator = ObjectAnimator
            .ofFloat(sunView, "y", sunYStart,
sunYEnd)
        .setDuration(3000)
               heightAnimator.interpolator    =
AccelerateInterpolator()
```

```
        val sunsetSkyAnimator = ObjectAnimator
                .ofInt(skyView, "backgroundColor",
    blueSkyColor, sunsetSkyColor)
                .setDuration(3000)

        heightAnimator.start()
        sunsetSkyAnimator.start()
    }
```

Вроде бы все делается правильно, но, запустив приложение, вы увидите, что что-то не так. Вместо плавного перехода от синего цвета к оранжевому цвета хаотично меняются, словно в калейдоскопе.

Это происходит из-за того, что целочисленное представление цвета — это не просто число, а четыре меньших числа, объединенных в одно значение `Int`. Таким образом, чтобы объект `ObjectAnimator` мог правильно вычислить промежуточный цвет на пути от синего к оранжевому, он должен знать, как это делать.

Когда обычного умения `ObjectAnimator` по вычислению промежуточных значений между начальной и конечной точками оказывается недостаточно, вы можете определить подкласс `TypeEvaluator` для решения проблемы. `TypeEvaluator` — объект, который сообщает `ObjectAnimator`, какое значение находится, допустим, на четверти пути от начальной до конечной точки. Android предоставляет подкласс `TypeEvaluator` с именем `ArgbEvaluator`, который позволит добиться желаемого результата.

Листинг 31.11. Назначение ArgbEvaluator (MainActivity.kt)

```
private fun startAnimation() {
    val sunYStart = sunView.top.toFloat()
    val sunYEnd = skyView.height.toFloat()

    val heightAnimator = ObjectAnimator
            .ofFloat(sunView, "y", sunYStart,
sunYEnd)
        .setDuration(3000)
              heightAnimator.interpolator    =
AccelerateInterpolator()

    val sunsetSkyAnimator = ObjectAnimator
            .ofInt(skyView, "backgroundColor",
blueSkyColor, sunsetSkyColor)
        .setDuration(3000)
    sunsetSkyAnimator.setEvaluator(ArgbEvaluato
r())

    heightAnimator.start()
    sunsetSkyAnimator.start()
}
```

(У `ArgbEvaluator` есть несколько версий, нам надо импортировать `android.animation`.)

Запустите анимацию еще раз, и вы увидите, как небо окрашивается в красивый оранжевый цвет (рис. 31.6).

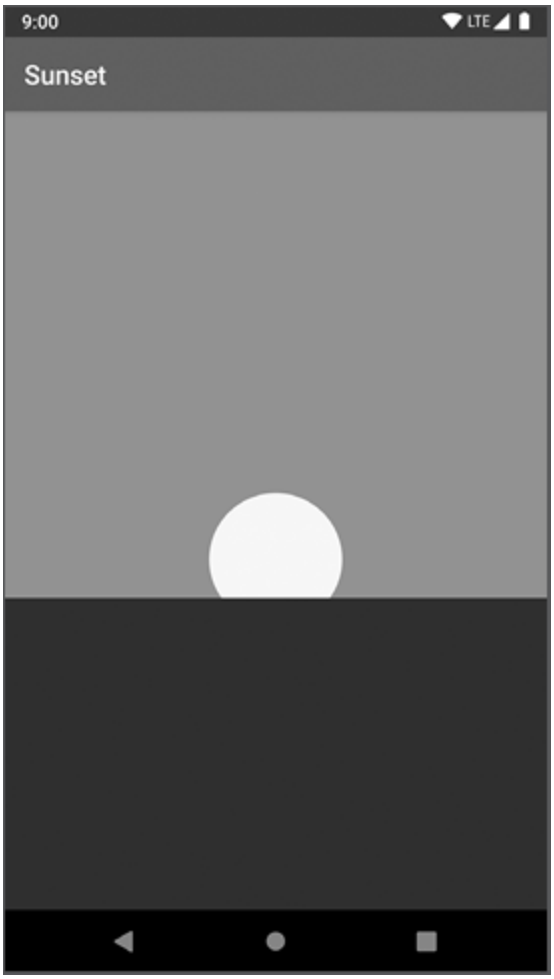


Рис. 31.6. Изменение цвета закатного неба

Одновременное воспроизведение анимаций

Если вам требуется всего лишь воспроизвести несколько анимаций одновременно, то ваша задача проста: одновременно вызовите для них функцию `start()`. Все анимации будут выполняться синхронно.

В ситуациях с более сложными анимациями этого недостаточно. Например, чтобы завершить иллюзию заката, было бы неплохо окрасить оранжевое небо в полуночный синий цвет после захода солнца.

Задача решается при помощи объекта AnimatorListener. Он сообщает о завершении анимации; таким образом, вы можете написать слушателя, который ожидает завершения первой анимации, а потом запустить вторую анимацию ночного неба. Тем не менее такое решение чрезвычайно хлопотно и требует слишком большого количества слушателей. Намного проще воспользоваться AnimatorSet.

Сначала создайте анимацию ночного неба и удалите старый код начала анимации.

Листинг 31.12. Создайте ночной анимации (MainActivity.kt)

```
private fun startAnimation() {
    val sunYStart = sunView.top.toFloat()
    val sunYEnd = skyView.height.toFloat()

    val heightAnimator = ObjectAnimator
            .ofFloat(sunView, "y", sunYStart,
sunYEnd)
        .setDuration(3000)
              heightAnimator.interpolator    =
AccelerateInterpolator()

    val sunsetSkyAnimator = ObjectAnimator
            .ofInt(skyView, "backgroundColor",
blueSkyColor, sunsetSkyColor)
        .setDuration(3000)
    sunsetSkyAnimator.setEvaluator(ArgbEvaluato
r())

    val nightSkyAnimator = ObjectAnimator
```

```
            .ofInt(skyView, "backgroundColor",
sunsetSkyColor, nightSkyColor)
        .setDuration(1500)
    nightSkyAnimator.setEvaluator(ArgbEvaluator
())

        heightAnimator.start()
        sunsetSkyAnimator.start()
}
```

Затем создайте и запустите AnimatorSet.

Листинг 31.13. Создайте AnimatorSet (MainActivity.kt)

```
private fun startAnimation() {
    ...
    val nightSkyAnimator = ObjectAnimator
            .ofInt(skyView, "backgroundColor",
sunsetSkyColor, nightSkyColor)
        .setDuration(1500)
    nightSkyAnimator.setEvaluator(ArgbEvaluator
())

    val animatorSet = AnimatorSet()
    animatorSet.play(heightAnimator)
        .with(sunsetSkyAnimator)
        .before(nightSkyAnimator)
    animatorSet.start()
}
```

Объект AnimatorSet представляет набор анимаций, которые могут воспроизводиться совместно. Существует

несколько способов построения таких объектов; проще всего воспользоваться функцией `play(Animator)`, которая использовалась выше.

При вызове функции `play(Animator)` вы получаете объект `AnimatorSet.Builder`, который позволяет построить цепочку инструкций. Объект `Animator`, передаваемый функции `play(Animator)`, является «субъектом» цепочки. Таким образом, написанная нами цепочка вызовов может быть описана в виде «Воспроизвести `heightAnimator` с `sunsetSkyAnimator`; также воспроизвести `heightAnimator` до `nightSkyAnimator`». Возможно, в сложных разновидностях `AnimatorSet` потребуется вызвать функцию `play(Animator)` несколько раз; это вполне нормально.

Запустите приложение еще раз и оцените созданный вами умиротворяющий закат. Волшебно.

Для любознательных: другие API анимации

Анимация свойств — наиболее универсальный механизм в инструментарии анимации, но не единственный. Полезно знать и о других возможностях, независимо от того, используете вы их или нет.

Устаревшие средства анимации

Еще одну группу составляют классы из пакета `android.view.animation` (не путайте с более новым пакетом `android.animation`, появившимся в Honeycomb).

Это устаревшая инфраструктура анимации, о которой следует знать в основном для того, чтобы избегать ее. Если в имени класса присутствует слово «animaTION» вместо

«animaTOR», это верный признак того, что перед вами старый инструмент и пользоваться им не следует.

Переходы

В Android 4.4 появилась новая инфраструктура, которая позволяет создавать эффектные переходы (transitions) между иерархиями представлений. Например, можно определить переход, при котором маленькое представление в одной activity «разворачивается» в увеличенную версию этого представления в другой activity.

Основной принцип механизма переходов — определение сцен, представляющих состояние иерархии представлений в некоторой точке, и переходов между этими сценами. Сцены могут описываться в XML-файлах макетов, а переходы — в XML-файлах анимации.

Когда activity уже работает, как в этой главе, механизм переходов особой пользы не принесет. В таких ситуациях эффективно работает механизм анимации свойств. С другой стороны, механизм анимации свойств не столь удобен для анимации макета в процессе его появления на экране.

Для примера возьмем фотографии места преступления из приложения CriminalIntent. Если вы попытаетесь реализовать анимацию «увеличения» в диалоговом окне изображения, вам придется самостоятельно рассчитывать, где находится исходное изображение и где будет находиться новое изображение в диалоговом окне. С `ObjectAnimator` реализация таких эффектов требует значительного объема работы. В таких случаях лучше воспользоваться инфраструктурой переходов.

Упражнения

Для начала добавьте возможность «обратить» закат солнца после его завершения: при первом нажатии солнце заходит, а при втором происходит восход. Для этого вам придется построить новый объект `AnimatorSet` — эти объекты не могут выполняться в обратном направлении.

Затем добавьте к солнцу вторичную анимацию: заставьте его дрожать от жара или создайте вращающийся ореол (для повторения анимации можно воспользоваться функцией `setRepeatCount(int)` класса `ObjectAnimator`).

Еще одно интересное упражнение — изобразить отражение солнца в воде.

На последнем шаге добавьте возможность обращения сцены заката по нажатию во время ее выполнения. Иначе говоря, если пользователь нажимает на сцене, пока солнце находится на середине пути, оно снова поднимается на небо. Аналогичным образом при нажатии во время наступления темноты небо снова должно окрашиваться в цвет зари.

Послесловие

Поздравляем! Вы добрались до последней страницы книги. Не каждому хватило бы терпения сделать то, что вы сделали, узнать то, что вы узнали. Ваш самоотверженный труд не пропал даром — теперь вы стали разработчиком Android.

Последнее упражнение

У нас осталось еще одно, последнее упражнение для вас: станьте *хорошим* разработчиком Android. Каждый хороший разработчик хорош по-своему, поэтому с этого момента вы должны найти собственный путь.

С чего начать, спросите вы? Мы можем дать несколько советов.

Пишите код. Начинайте прямо сейчас. Вы быстро забудете то, что узнали в книге, если не будете применять полученные знания. Примите участие в проекте или напишите собственное простое приложение. Что бы вы ни выбрали, не тратьте времени: пишите код.

Учитесь. В этой книге вы узнали много полезной информации. Что-то из этого пробудило ваше воображение? Напишите код, чтобы поэкспериментировать со своими любимыми возможностями. Найдите и почитайте дополнительную документацию по ним — или целую книгу, если она есть. Также загляните на канал Android Developers на YouTube и послушайте подкаст Android Developers Backstage.

Встречайтесь с людьми. Локальные встречи также помогут вам найти единомышленников. Многие первоклассные разработчики Android активно общаются в Twitter и в Google Plus. Посещайте конференции Android, на них вы

познакомитесь с другими разработчиками Android (а может, и с нами!).

Присоединяйтесь к сообществу разработки с открытым кодом. Разработка Android процветает на сайте **www.github.com**. Обнаружив полезную библиотеку, посмотрите, в каких еще проектах участвуют ее авторы. Делитесь своим кодом — никогда не знаешь, кому он пригодится. Список рассылки Android Weekly поможет вам быть в курсе происходящего в сообществе Android (**androidweekly.net**).

Спасибо вам!

Без таких читателей, как вы, наша работа была бы невозможной. Спасибо вам за то, что купили и прочитали нашу книгу.

Чистый Agile. Основы гибкости

Роберт Мартин

ЧИСТЫЙ AGILE

ОСНОВЫ ГИБКОСТИ

РОБЕРТ МАРТИН

Прошло почти двадцать лет с тех пор, как появился Манифест Agile. Легендарный Роберт Мартин (Дядя Боб) понял, что пора стряхнуть пыль с принципов Agile и заново рассказать о гибком подходе не только новому поколению программистов, но и специалистам из других отраслей. Автор полюбившихся айтишникам книг «Чистый код», «Идеальный программист», «Чистая архитектура» стоял у истоков Agile. «Чистый Agile» устраняет недопонимание и путаницу, которые за годы существования Agile усложнили его применение по сравнению с изначальным замыслом. По сути Agile — это всего лишь небольшая подборка методов и инструментов, помогающая небольшим командам программистов управлять небольшими проектами,... но приводящая к большим результатам, потому что каждый крупный проект состоит из огромного количества кирпичиков. Пять десятков лет работы с проектами всех мыслимых видов и размеров позволяют Дяде Бобу показать, как на самом деле должен работать Agile. Если вы хотите понять преимущества Agile, не ищите лёгких путей — нужно правильно применять Agile. «Чистый Agile» расскажет, как это делать разработчикам, тестировщикам, руководителям, менеджерам проектов и их клиентам.

Мифический человеко-месяц, или Как создаются программные системы

Фредерик Брукс

ФРЕДЕРИК БРУКС МЛАДШИЙ

ОЧЕРКИ О ПРОГРАММНОЙ ИНЖЕНЕРИИ

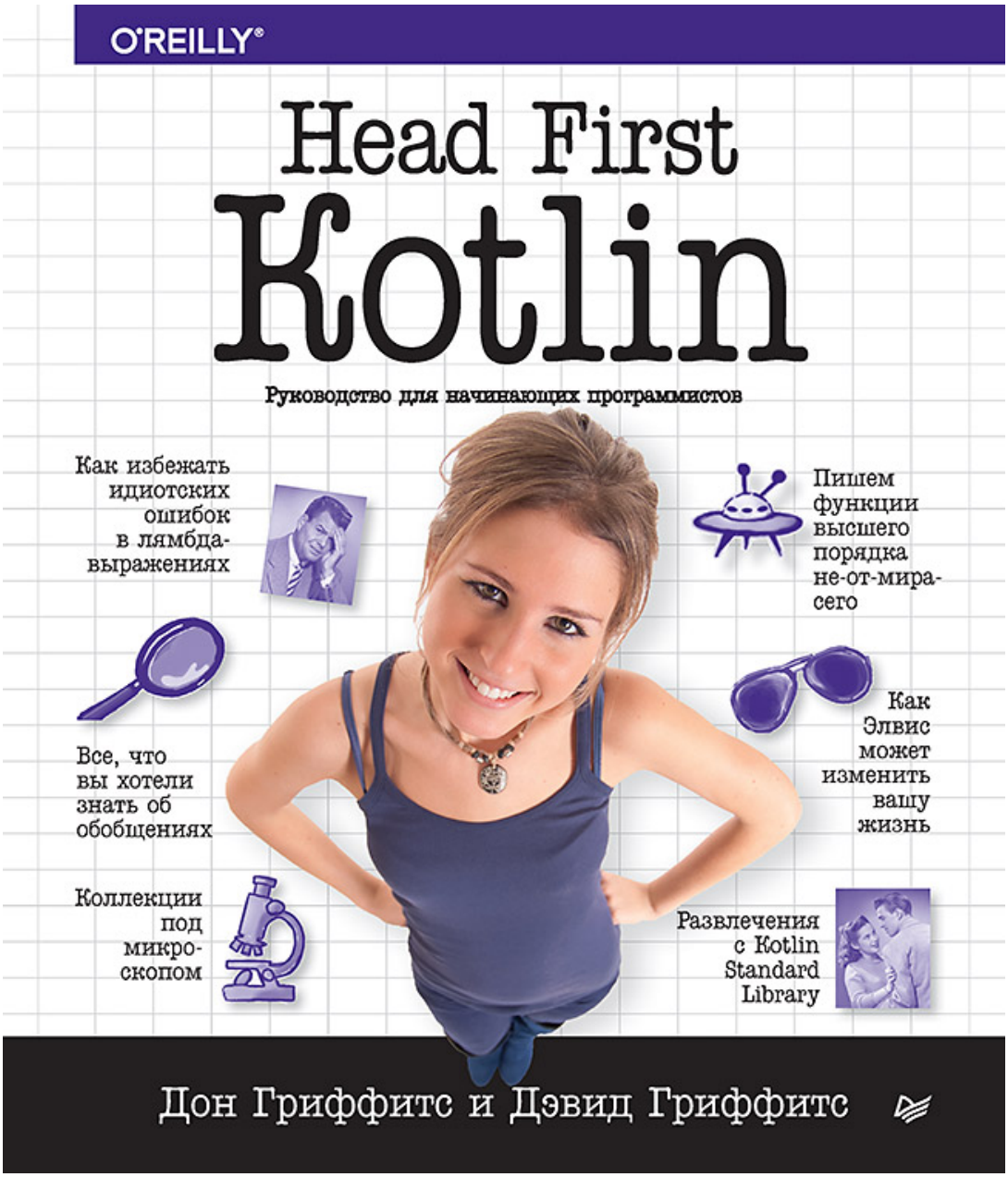
МИФИЧЕСКИЙ ЧЕЛОВЕКО-МЕСЯЦ

ИЛИ КАК СОЗДАЮТСЯ ПРОГРАММНЫЕ СИСТЕМЫ

Немногие книги по управлению проектами можно назвать столь же значимыми, как «Мифический человеко-месяц». Смешение примеров из реальной разработки ПО, мнений и размышлений создает яркую картину управления сложными проектами. Эти эссе основаны на пятидесятилетнем опыте работы Брукса менеджером проектов в IBM System/360, а затем в OS/360. Первое издание книги вышло 45 лет назад, второе - 25 лет назад. Возникают новые методологии, появляются новые языки программирования, растет количество процессоров, но эта книга продолжает оставаться актуальной. Почему? Спустя полвека мы продолжаем повторять ошибки, которые описал Брукс. Некоторые темы, поднимаемые в книге, кажутся устаревшими, но это лишь видимость. Фундаментальные проблемы, стоящие за ними, все так же актуальны в наше время. Важно знать свое прошлое, чтобы понимать, куда развивается индустрия разработки программного обеспечения. Поэтому спустя 45 лет мы и читаем Брукса. Многое изменилось в мире, но девять женщин все так же не могут выносить ребенка за один месяц. ;)

Head First. Kotlin

Дон Гриффитс, Дэвид Гриффитс



Вот и настало время изучить Kotlin. В этом вам поможет уникальная методика Head First, выходящая за рамки синтаксиса и инструкций по решению конкретных задач. Хотите мыслить, как выдающиеся разработчики Kotlin? Эта книга даст вам все необходимое — от азов языка до продвинутых методов.

А еще вы сможете попрактиковаться в объектно-ориентированном и функциональном программировании. Если вы действительно хотите понять, как устроен Kotlin, то эта книга для вас! Почему эта книга не похожа на другие? Подход Head First основан на новейших исследованиях в области когнитивистики и теории обучения. Визуальный формат позволяет вовлечь в обучение мозг читателя лучше, чем длинный текст, который вгоняет в сон. Зачем тратить время на борьбу с новыми концепциями? Head First задействует разные каналы получения информации и разрабатывался с учетом особенностей работы вашего мозга. «Четко, доступно, просто для понимания. Если вы только осваиваете Kotlin, эта книга станет отличным вводным пособием». — Кен Коусен (Ken Kousen) Сертифицированный преподаватель Kotlin «Head First Kotlin поможет вам быстро проникнуть в суть дела, заложить надежный фундамент и (снова) получать удовольствие от написания кода». — Инго Кроцки (Ingo Krotzky), изучающий Kotlin «Наконец-то! Kotlin для тех, кто не знает Java. Просто, лаконично и занимательно. Я давно ждал появления такой книги». — Доктор Мэтт Венэм (Dr. Matt Wenham), специалист по data science и программист Python

Kotlin. Программирование для профессионалов

Джош Скин, Дэвид Гринхол


о. Котлин
Кронштадт
Kotlin
Программирование
для профессионалов
Джош Скин и Дэвид Гринхол
Big Nerd
Ranch

Kotlin — язык программирования со статической типизацией, который взяла на вооружение Google в ОС Android. Книга Джоша Скина и Дэвида Гринхола основана на популярном курсе Kotlin Essentials от Big Nerd Ranch. Яркие и полезные примеры, четкие объяснения ключевых концепций и основополагающих API не только знакомят с языком Kotlin, но и учат эффективно использовать его возможности, а также позволяют освоить среду разработки IntelliJ IDEA от JetBrains. Неважно, опытный вы разработчик, который хочет выйти за рамки Java, или изучаете первый язык программирования. Джош и Дэвид проведут вас от основных принципов к расширенному использованию Kotlin, чтобы вы могли создавать надежные и эффективные приложения.