УДК 004.43
ББК 32.972
Л91

Льюис Ш., Данн М.
Л91 Нативная разработка мобильных приложений / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2020. – 376 с.: ил.

ISBN 978-5-97060-845-6

В этой книге вы познакомитесь с простыми подходами к разработке мобильных приложений для iOS и Android. Если вашей команде приходится разрабатывать проекты сразу для двух этих систем или вы планируете перейти с одной системы на другую, это практическое руководство покажет вам, как решаются наиболее распространенные задачи на каждой из этих платформ.

В первой части представлены решения распространенных задач, которые приходится решать на любой платформе, таких как запись файла в локальное хранилище или создание HTTP-запроса. Вторая часть описывает процесс создания приложения на каждой платформе с использованием приемов из первой части. Примеры кода для Android представлены на двух языках – Java и Kotlin, поэтому книга может служить перекрестным справочником не только между iOS и AOSP, но и между Java и Kotlin для разработчиков на Android.

Издание предназначено для программистов, специализирующихся на разработке приложений для iOS и/или Android.

УДК 004.43
ББК 32.972

ISBN 978-1-492-05287-6 (англ.)

ISBN 978-5-97060-845-6 (рус.)

Содержание

Об авторах

Шон Льюис (Shaun Lewis) – бывший ведущий разработчик программного обеспечения для iOS, а ныне руководитель подразделения Mobile Engineering в издательстве O'Reilly Media. Первая книга «How to Build a Website in a Weekend», которую он прочитал, коренным образом изменила устремления 15-летнего юноши. Теперь он имеет за плечами 12-летний опыт профессиональной разработки приложений для iPhone, начав заниматься этим, еще когда iOS еще называлась iPhone OS. Сотрудничал с рядом компаний из списка Fortune 500 и иногда выступает на встречах разработчиков продуктов Apple. Шон живет в Огайо со своей женой, двумя детьми и полным ящиком старых смартфонов.

Майк Данн (Mike Dunn) – ведущий инженер по мобильным технологиям и технологиям Android в издательстве O'Reilly Media. Майк является признанным членом сообщества AOSP и активно участвует в развитии экосистемы открытого исходного кода Android, занимаясь в том числе развитием и обслуживанием популярной библиотеки TileView. Внес свой вклад в разработку библиотеки Google Closure и в поддержку ExoPlayer – мультимедийного проигрывателя Google следующего поколения. Майк профессионально занимается программированием на протяжении многих лет и продолжает обучаться информатике в рамках магистратуры в технологическом институте штата Джорджия.

О колофоне

На обложке книги «Нативная разработка мобильных приложений» изображена норикийская лошадь (*Equus ferus caballus*). Название «норикийская» происходит от названия австрийской провинции *Норикум* в Римской империи. Порода выведена в Австрии, где она также известна как пинцгауская лошадь.

Норикийских лошадей разводили в австрийских Альпах для перевозки товаров по всей Европе. Это сильные, терпеливые и послушные животные. Мускулистые и крепкие, они весят почти тонну (в среднем 1550 фунтов, или 700 кг). Их короткие ноги надежно удерживают такой вес на пересеченной местности. Норикийские лошади могут иметь черный, лавровый или каштановый окрас.

В настоящее время, когда благодаря индустриализации спрос на ломовых лошадей снизился, норикийские лошади используются в основном для верховой езды. Они прекрасно себя чувствуют в горных условиях на высотах до двух тысяч метров.

Многие животные, изображаемые на обложках книг издательства O'Reilly, находятся под угрозой вымирания; все они очень важны для нашего мира. В наши дни норикийская лошадь считается «породой с ограниченным распространением». Чтобы узнать, чем вы можете помочь, посетите сайт animals.oreilly.com.

Иллюстрацию для обложки нарисовала Карен Монтгомери (Karen Montgomery) на основе старой черно-белой гравюры (источник неизвестен).

Вступление

Почему мы написали эту книгу

Эта книга является практическим перекрестным справочником разработчика приложений для iOS и Android. Под «нативной разработкой» мы подразумеваем использование оригинальных инструментов, предлагаемых каждой платформой – Swift и Cocoa для iOS и Java или Kotlin с набором инструментов для разработки программного обеспечения (Software Development Kit, SDK) от Android Open Source Project (AOSP) для Android.

Написать эту книгу нас побудила банальная потребность в таком справочнике. Оба автора имеют опыт работы с обеими платформами, но специализируются на одной. Нередко члены нашей команды (включая нас самих), занимаясь какой-то проблемой на одной платформе, находили ее решение на этой платформе, а затем делились этим решением с членами команды, работающими на другой платформе.

Часто это были задачи, общие для обеих платформ, такие как чтение или запись в базу данных либо в файлы, создание сетевого подключения или отображение обратной связи в привычном для пользователей виде. Начав писать код и документировать эти задачи на обеих платформах, мы быстро заметили, что подавляющее большинство решений относится к этой категории и перекрестный справочник по этим решениям мог бы очень пригодиться командам со смешанным составом, начинающим переход на другую платформу и даже разработчикам, желающим начать изучение обеих платформ сразу.

Мы надеемся, что эта книга будет полезна читателям в таком качестве и поможет им в решении типичных задач, возникающих при разработке мобильных приложений.

Имея целую команду разработчиков мобильных приложений с богатым опытом разработки для двух платформ, мы относительно легко преодолевали затруднения. Тем не менее мы долго вынашивали идею перекрестного справочника, потому что до настоящего времени на рынке не было книг, охватывающих эту тему с необходимой широтой. Мы легко можем представить разочарование разработчика, работающего в одиночку и столкнувшегося с такой же ситуацией; эта книга поможет освоить базовые подходы к решению большинства типичных задач разработки приложений для обеих платформ. В каждой главе, посвященной определенной категории задач, описывается весь процесс их решения в Android и iOS с использованием простых и понятных примеров кода. По нашей оценке, эти примеры охватывают 80 % базовых знаний, необходимых, чтобы приступить к разработке приложений; конечно же, мы полагаем, что читатель не остановится на достигнутом и продолжит расширять свои знания и читать документацию, чтобы самостоятельно найти решение для задач, которые мы намеренно обошли стороной. Мы также включили в справочник пошаговый пример разработки приложения, демонстрирующий реализацию практически всего, что может потребоваться современному при-

ложению, с использованием информации из предшествующего перекрестного справочника.

Поскольку все примеры кода для Android представлены на двух языках – Java и Kotlin, – эта книга имеет приятный побочный эффект: она может служить перекрестным справочником не только между iOS и AOSP, но и между Java и Kotlin для разработчиков на Android.

Примеры кода не являются псевдокодом; они написаны на конкретных языках программирования и должны компилироваться и действовать, как описано.

Кому адресована эта книга

Эта книга предназначена для всех программистов, работающих только с какой-то одной платформой или с обеими, либо знакомых с одной, но желающих освоить другую. Мы предполагаем у читателя наличие хотя бы поверхностного знания некоторого языка программирования. Вам не обязательно быть экспертом в Java или Swift, но некоторый опыт программирования пользовательского интерфейса не будет лишним.

Возможно, вам придется обратиться к официальной документации с описанием Objective-C, Swift, Java или Kotlin, чтобы понять некоторые основные особенности языков, которые упоминаются в этой книге.

Программистам, имеющим опыт работы с одной из платформ (iOS или Android), будет легко усвоить предоставленную информацию, потому что почти каждому примеру кода для одной платформы соответствует функционально эквивалентный пример для другой платформы.

Организация книги

Эта книга разделена на две части. В первой части представлены решения распространенных задач, которые приходится решать на любой платформе, таких как запись файла в локальное хранилище или создание HTTP-запроса. Вторая часть описывает процесс создания приложения на каждой платформе с использованием приемов из первой части.

Соглашения

В этой книге используются следующие соглашения по оформлению:

Курсив
: Используется для обозначения новых терминов, адресов URL и электронной почты, имен файлов и расширений имен файлов.

`Моноширинный шрифт`
: Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

Моноширинный жирный
Обозначает команды или другой текст, который должен вводиться пользователем.

Моноширинный курсив
Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.


Так выделяются советы и предложения.


Так обозначаются примечания общего характера.


Так обозначаются предупреждения и предостережения.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте http://dmkpress.com/authors/publish_book/ или напишите в издательство: dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Скачивание исходного кода

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Благодарности

Мы хотели бы поблагодарить своих рецензентов: Пэриса Баттфилд-Аддисона (Paris Buttfield-Addison), Яна Дарвина (Ian Darwyn), Дона Гриффитса (Dawn Griffiths), Бена Кригера (Ben Kreeger), Пьера-Оливье Лоуренса (Pierre-Olivier Laurence), Шейна Степлса (Shane Staples) и Субатра Танабалан (Subathra Thanabalan).

Часть I

ЗАДАЧИ И ОПЕРАЦИИ

В этой части мы предоставим основополагающий код для выполнения различных задач, типичных для мобильных приложений. К их числу мы относим отображение пользовательского интерфейса, передачу данных, отправку и получение событий, выполнение сетевых запросов и обработку ответов, доступ к файловой системе и управление ею, а также чтение или запись в постоянные хранилища, такие как базы данных или файлы с настройками.

Примечание о текущем положении дел в сфере разработки мобильных приложений

На момент написания этой книги сфера разработки мобильных приложений для Android и iOS находилась в крайне неустойчивом состоянии, была сильно фрагментирована и переполнена противоречиями. Добавлялись новые библиотеки, предназначенные для замены существующих API, но не все они являлись результатом обсуждения и согласования членами сообщества разработчиков. Кроме того, почти каждый новый API был значительно сложнее своего предшественника. Выбирая информацию для включения в будущую книгу, включая основополагающие библиотеки и API, мы решили, что должны постараться оказать максимальную помощь как можно большему числу людей и проектов. В подавляющем большинстве случаев разработчики реализуют одни и те же особенности и адаптируют имеющиеся, а не занимаются созданием чего-то принципиально нового (с чистого листа). Учитывая это, почти во всех случаях мы по умолчанию используем существующие и проверенные временем библиотеки и приемы и стараемся избегать применения новых, еще не устоявшихся библиотек, появившихся относительно недавно (мы произвольно выбрали порог около года).

Кроме того, мы обнаружили, что многие новые API, пришедшие на смену прежним, уводят инфраструктуру от шаблонов, применяемых в существующих технологиях. Например, новый набор компонентов `Navigation` в Android можно использовать для управления пользовательским интерфейсом с применением тех же базовых подходов, которые представлены в этой книге (экземпляры `Fragment`), но в совершенно иной манере. Учитывая объем необходимой инфраструктуры, мы решили, что включение описания этих дополнительных шаблонов и поясняющего кода, помимо проверенных решений, которые, как мы

считаем, представляют реальность разработки для Android на данный момент, принесет больше вреда, чем пользы. Другим примером является технология баз данных: совсем недавно компания Google рекомендовала разработчикам для Android использовать свою библиотеку Room, тогда как в стандартные библиотеки AOSP включены средства для работы с SQLite. Никто не утверждает, что SQLite не имеет ограничений, и мы допускаем, что Room предлагает более современный подход, но одна из наших основных целей состояла в том, чтобы предоставить сведения, опираясь на которые, любой, знакомый с программированием в целом, смог бы быстро приступить к продуктивной работе, к тому же SQL является, пожалуй, одной из самых распространенных и зрелых технологий. Room пока не может похвастаться зрелостью; поэтому в своих примерах хранения данных мы используем SQLite. Если вы решите использовать Room, в этом нет ничего плохого! Мы советуем не упускать из виду новые технологии и рекомендации, и мы сами постараемся напоминать о них, когда это будет уместно, но не удивляйтесь, что мы используем `FragmentManager` со списком компонентов, необходимых для применения `Navigation` API, – это обдуманное решение, которое мы считаем правильным *на данный момент*.

Точно так же, и снова в интересах использования любых существующих знаний о предметной области, которыми вы, уважаемый читатель, обладаете, мы можем использовать менее эффективные, но более понятные или распространенные шаблоны либо методы. Например, для чтения из потока данных в Android мы часто используем `InputStream.read` без буферизации. Даже притом что во многих случаях буферизация является уместной, отказ от ее применения позволил нам не только сократить и упростить примеры кода, но и избавил от необходимости объяснять, как работают буферы (с обеих сторон, на входе и на выходе), какой размер буфера является наиболее подходящим в тех или иных обстоятельствах и почему эффективнее предварительно выделить один буфер, чем создавать новый для каждой операции чтения или записи. Потоковый буфер – сам по себе довольно простое понятие, но *правильно и полностью* объяснить его работу – нетривиальная задача. По аналогичной причине современные версии Java предлагают конструкцию try-with-resources для операций с экземплярами `Closable`. В этом конкретном случае кто-то, знакомый с конструкцией try-catch в другом языке (например, JavaScript), сразу распознает стандартный синтаксис и сможет сосредоточить свое внимание на описываемой нами задаче, и ему не придется приостанавливаться, чтобы познакомиться с альтернативным синтаксисом try-with-resources и с трудом продираться через еще один или два абзаца, которые никак не способствуют его главной цели. Конечно, в некоторых случаях try-with-resources – это более чем уместный прием, и мы всем советуем стремиться узнать как можно больше о каждом языке и фреймворке, но эта книга призвана помочь программистам начать *программирование*, а не освоить каждую технологию, которую мы будем затрагивать здесь.

Спасибо всем, кто терпеливо выслушивал нас, и за все высказанные мнения, которые привели к появлению этого примечания, мы искренне ценим вашу вдумчивость.

Глава 1

Контроллеры пользовательского интерфейса

Контроллеры пользовательского интерфейса (User Interface, UI) играют роль связующего звена между пользовательским интерфейсом и бизнес-логикой приложения, которая управляет этим пользовательским интерфейсом или получает информацию от него.

Если провести аналогию между приложением и пьесой Шекспира, поставленной в каком-нибудь театре Старого Света, тогда контроллер пользовательского интерфейса можно сравнить с помощником режиссера. Он выводит актеров на сцену, получает команды от режиссера и помогает в смене декораций.

Каждый раз, когда в приложении потребуется отобразить изображение, список или фрагмент текста, вам потребуется пользовательский интерфейс. Представление пользовательского интерфейса – отображение на экране – обычно определяется макетом (часто оформленным в виде разметки XML или HTML); контроллер пользовательского интерфейса действует как мост между командами ввода, запросами к базе данных, запросами межпроцессных взаимодействий (IPC), сообщениями и многим другим. В каком-то смысле это сердце любого приложения.

Все это жонглирование требует невероятно сложной серии событий с применением множества технологий, основанных друг на друге и действующих согласованно. К счастью, обе платформы, Android и iOS, предлагают ряд общих инструментов и абстракций для управления этим тяжелым процессом. Давайте познакомимся с некоторыми основными задачами в данной области, которые являются центральными для обеих платформ.

Задачи

В этой главе вы узнаете:

1) как создать начальный контроллер пользовательского интерфейса приложения;
2) как изменить активный контроллер пользовательского интерфейса;

3) основные этапы жизненного цикла контроллера пользовательского интерфейса.

Android

Менее чем за год до того, как мы взялись за эту книгу, компания Google рекомендовала для навигации в приложении использовать один экземпляр `Activity` и экземпляры класса `Fragment` в нем для реализации операций и управления представлениями. Для управления взаимодействиями между фрагментами и историей отображения предлагалось использовать новый компонент `Navigation`, выпущенный в пакете Jetpack.

Обратите внимание, что эта рекомендация идет вразрез с практиками, предлагавшимися с момента появления Android более десяти лет назад, когда `Activity` рекомендовалось использовать для любой «деятельности» (примерным аналогом «деятельности», или «активности», является «экран» либо отдельная веб-страница), а использование вложенных экземпляров `Fragment` то приветствовалось, то не приветствовалось. Фактически даже в настоящее время разработчики для Android начинают главу об `Activity` со следующих слов:

> Деятельность – это узкоспециализированная экранная форма, позволяющая пользователю выполнить определенную операцию.

В пользу обеих сторон можно привести веские аргументы, но, поскольку разработчиком Android является Google, мы считаем, что в будущем должны принять ее рекомендацию. Тем не менее существует *множество* давнишних приложений, разработчики которых не использовали этот шаблон и не планируют менять код, написанный за несколько лет работы, чтобы внедрить его. Мы не будем принимать чью-либо сторону и покажем основы обоих подходов. В случае сомнений мы будем использовать преобладающий существующий подход – запускать новые экземпляры `Activity`, передавать данные в виде экземпляров `Bundle` и управлять модульным контентом с помощью экземпляров `Fragment` и методов контроллера `Activity`, стараясь избегать использования более нового компонента `Navigation` архитектуры навигации и его родственников.

Как создать начальный контроллер пользовательского интерфейса приложения

Давайте сразу приступим к делу. Когда приложение запустится, оно выполнит некоторую логику инициализации и создаст «фон окна» (обычно сплошной цвет, в зависимости от вашего экрана, который можно заменить любым допустимым экземпляром `Drawable`). Эта работа выполняется в главном потоке и не может быть прервана или приостановлена – она просто будет выполнена. Обратите внимание, что если для приложения реализован свой класс `Application`, в этот момент будет вызван его метод `onCreate`. И снова, важно помнить об этом, вызов произойдет в главном потоке выполнения (его еще называют потоком пользовательского интерфейса), поэтому все остальные операции бу-

дут отложены до окончания его выполнения. Однако в настоящее время есть возможность организовать асинхронное выполнение операций в фоновых потоках.

По завершении инициализации приложение запустит один экземпляр класса `Activity`, который вы определили в манифесте приложения со значением `android.intent.category.LAUNCHER` в его элементе `category`. Описание этого экземпляра `Activity` должно также включать элемент `action` со значением `android.intent.action.MAIN`, определяющим любую из точек входа в ваше приложение (например, через значок запуска, глубокую ссылку, общесистемные широковещательные сообщения и т. д.).

Вы должны указать только каноническое имя класса, а создание экземпляров и их настройка выполняются автоматически (то есть этот процесс совершенно непрозрачен для нас как разработчиков или пользователей).

```
<intent-filter>
  <action android:name="android.intent.action.MAIN" />
  <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

В полном манифесте этот фрагмент выглядит так:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="org.oreilly.nmd"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <application
      android:allowBackup="false"
      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=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity android:name=".BrowseContentActivity" />
    <activity android:name=".BookDetailActivity" />
    <activity android:name=".SearchResultsActivity" />
  </application>

</manifest>
```

Обратите внимание, что любой экземпляр `Activity`, который предполагается использовать в приложении, должен быть зарегистрирован в *ApplicationManifest.xml* как дочерний узел узла `application` (`manifest` ⇒ `application` ⇒ все элементы `activity`). Убедитесь в этом, заглянув в блок кода выше, как прочтете это примечание.

```
<activity android:name=".MyActivity" />
```

Считается, что в процессе взаимодействия с приложением Android пользователь всегда находится в Activity (исключение составляют удаленные операции, такие как взаимодействие строки состояния с Service, но это очень незначительное исключение для данной главы). У вас никогда не будет пригодного для использования элемента пользовательского интерфейса, не входящего в состав какого-то экземпляра Activity (единственным исключением является класс RemoteViews – небольшое подмножество простых классов View, – доступный в окнах уведомлений).

Обратите внимание, что экземпляры Activity нельзя вкладывать друг в друга. Вообще говоря, один экземпляр Activity занимает сразу весь экран (или, по крайней мере, часть, делегированную приложению).

Как уже упоминалось, вы не должны создавать новые экземпляры Activity; от вас требуется только указать класс Activity, который следует запустить. За кулисами платформа Android создаст нужный экземпляр и выполнит все подготовительные операции, прежде чем отобразить его. Кроме того, эта операция выполняется *асинхронно*, и система сама решает, когда запустить новый экземпляр Activity.

Это особенно важно, потому что в файле манифеста разным экземплярам Activity могут быть назначены разные режимы запуска. Конкретный режим запуска может позволить одновременно существовать любому количеству экземпляров класса Activity. Например, вы можете разрешить пользователю иметь любое количество экземпляров ComposeEmailActivity в одном стеке задач, и при этом ограничить количество экземпляров других видов Activity, например разрешить только один экземпляр LoginActivity, что может привести к перемещению последнего использовавшегося LoginActivity на вершину стека задач или к уничтожению всего, что находится между текущим экземпляром Activity и последним использовавшимся LoginActivity, в зависимости от режима запуска. Мы не будем подробно останавливаться на режимах запуска, но вы обязательно загляните в документацию для разработчиков и познакомьтесь с этим вопросом поближе, если, конечно, вам это интересно.

Итак, мы благополучно запустили Activity, но почему на экране ничего не появляется? Потому что Activity – это класс уровня контроллера, а не видимое представление. Чтобы отобразить элементы на экране, нужен как минимум один экземпляр View или несколько (в виде дочерних элементов в экземпляре View, используемом в качестве корня в Activity). Обычно это делается с помощью метода setContentView и передачи ресурса макета в формате XML. Подробнее об этом мы расскажем в главе 2.

Как изменить активный контроллер пользовательского интерфейса

После того как начальный («запускающий») экземпляр Activity отобразится, появляется возможность запустить любой другой экземпляр Activity, вызвав метод startActivity(Intent intent) любого экземпляра Context (класс Activity наследует Context, поэтому он связан с ним отношением «является» – экземпляр Activity *является* экземпляром Context). Экземпляр Intent, в свою очередь, тре-

бует передать экземпляр Context в первом параметре и ссылку на запускаемый класс Activity:

Java

```
// предполагается, что запускающий Activity находится в области видимости
Intent intent = new Intent(this, AnotherActivity.class);
startActivity(intent);

// если он находится вне области видимости, но имеется доступ к объекту
// Context, можно использовать такой код...
// предполагается, что переменная "context" содержит ссылку на объект Context.
Intent intent = new Intent(context, AnotherActivity.class);
context.startActivity(intent);
```

Kotlin

```
// предполагается, что запускающий Activity находится в области видимости
val intent = Intent(this, AnotherActivity::class.java)
startActivity(intent)

// если он находится вне области видимости, но имеется доступ к объекту
// Context, можно использовать такой код...
// предполагается, что переменная "context" содержит ссылку на объект Context.
val intent = Intent(context, AnotherActivity::class.java)
context.startActivity(intent)
```

Важно понимать, что созданием, инициализацией и настройкой экземпляров Activity, которые вы покажете своему пользователю, будет заниматься система – их нельзя создать с помощью ключевого слова new, настроить или иным образом изменить при запуске. Мы передаем системе экземпляр Intent, который определяет, какой класс Activity мы хотим использовать, а система делает все остальное. По этой причине нет никакой возможности изменять свойства и вызывать методы экземпляра Activity непосредственно (с использованием стандартной библиотеки) во время запуска.

Но коль скоро нельзя изменять свойства и вызывать методы экземпляра Activity непосредственно в процессе запуска, как тогда передать ему информацию? Во многих фреймворках пользовательского интерфейса есть возможность создать новый экземпляр класса контроллера представления, записать в него некоторые данные и дать ему возможность отобразить их.

В Android ваши возможности весьма ограничены. Классический подход заключается в привязке простых значений к объекту Intent, например так:

Java

```
// предполагается, что запускающий Activity находится в области видимости
Intent intent = new Intent(this, AnotherActivity.class);
intent.putExtra("id", 10);
startActivity(intent);
```

Kotlin

```
// предполагается, что запускающий Activity находится в области видимости
Intent intent = Intent(this, AnotherActivity::class.java);
intent.putExtra("id", 10)
startActivity(intent)
```

Экземпляр Intent, запустивший Activity, доступен через метод getIntent:

Java

```
public class MyActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Intent intent = getIntent();
    int id = intent.getIntExtra("id", 0);
    Log.d("MyTag", "id: " + id);
  }
}
```

Kotlin

```
class MyActivity : Activity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val id = intent.getIntExtra("id", 0)
    Log.d("MyTag", "id: $id")
  }
}
```

Этот способ идеально подходит для передачи простых данных, таких как числовой идентификатор или URL, но не подходит для больших данных (например, сериализованных классов Java или даже просто больших строк со сложными экземплярами классов в формате JSON). Эти данные содержатся в определенном хранилище системного уровня, имеющем ограниченный объем 1 Мбайт, и могут использоваться всеми процессами на устройстве. Вот выдержка из документации с описанием Bundle API:

> Буфер транзакций Binder имеет ограниченный размер, в настоящее время 1 Мбайт, и используется всеми транзакциями, выполняемыми процессом. Поскольку это ограничение определяется на уровне процесса, а не на уровне активности, этот буфер используют все транзакции привязки в приложении, такие как onSaveInstanceState, startActivity и любые взаимодействия с системой.

Чтобы передать сложную информацию во вновь созданный экземпляр Activity, нужно сохранить ее на диск перед запуском нового Activity, чтобы ее можно было прочитать обратно после создания этого Activity, или передать ссылку на «глобальную» структуру данных. Часто роль такой структуры играет простая переменная уровня класса (т. е. статическая), но в этом случае вам придется учитывать все недостатки, свойственные статическим переменным. Инженеры Android в свое время рекомендовали использовать статический член Map<WeakReferences> в служебном классе или в экземпляре Application (всегда доступном из любого экземпляра Context через Context.getApplicationContext). Важно отметить, что пока приложение работает, экземпляр Application будет доступен, и, как утверждают некоторые, при его использовании вы никогда не столкнетесь с утечками памяти. В Kotlin глобальный контекст обрабатывается немного иначе, но в целом все предупреждения, касающиеся передачи информации, остаются в силе.

Фрагменты

Фрагментом (класс `Fragment`) на языке Android называется этакий облегченный вариант `Activity`. Его можно рассматривать как *контроллер* представления, а не как само представление, но он должен иметь доступ к корневому представлению (в Android роль реализации «представления» в шаблонах Model-View-Presenter [MVP], Model-View-Controller [MVC], Model-View-ViewModel [MVVM] и др. выполняется классом `View`, который обычно является атомарным визуальным элементом, таким как фрагмент текста, изображение или контейнер для другого экземпляра `View`; более подробно об этом рассказывается в главе 2).

Замечательной чертой `Fragment`, отличающей этот класс от класса `Activity`, является возможность создавать его экземпляры напрямую, используя конструкторы, конфигурации, члены и методы и т. д. Экземпляр класса `Fragment` создается точно так же, как экземпляр любого другого класса в Java. Кроме того, экземпляры `Fragment`, в отличие от `Activity`, *могут* быть вложенными, однако исторически сложилось так, что вложенность сопряжена с некоторой ненадежностью, в частности в отношении вызовов методов жизненного цикла, но обсуждение этого вопроса выходит за рамки данной главы. В Google давно ведутся «жаркие споры о целесообразности использования фрагментов в Android», и в интернете можно найти массу статей по этой теме. Однако мы в своей книге предпочитаем оставаться на нейтральных позициях в этой бессмысленной войне взглядов.

Итак, вот как можно создать экземпляр `Fragment`:

Java

```
public class MyFragment extends Fragment {
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
    return inflater.inflate(R.layout.my_layout, container, false);
  }
}
```

Kotlin

```
class MyFragment : Fragment() {
  override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle): View? {
    return inflater.inflate(R.layout.my_layout, container, false)
  }
}
```

В идеале фрагменты лучше добавлять в макеты XML, описывающие представления `View`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name=".ListFragment"
            android:layout_width="200dp"
            android:layout_height="match_parent" />
```

```
    <fragment android:name=".DetailFragment"
            android:layout_weight="1"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
</LinearLayout>
```

Однако мы снова сталкиваемся с непрозрачной процедурой создания экземпляров на системном уровне. Чтобы настроить экземпляр класса `Fragment` программным способом, нужно создать его экземпляр с помощью ключевого слова `new` и использовать `FragmentManager` и `FragmentTransaction` для его добавления в существующую иерархию представления.

Кроме того, вы можете объявить свой класс, наследующий `Fragment`, со своими параметрами, однако при повторном воссоздании экземпляра `Fragment` аргументы его конструктора будут утеряны, поэтому разработчикам для Android предлагается использовать конструкторы без аргументов и исходить из предположения, что экземпляры `Fragment` могут создаваться с помощью метода `Class.newInstance`.

Java

```
Fragment fragment = new MyFragment();
```

Kotlin

```
val fragment = MyFragment()
```

Далее, поскольку `Fragment` является не представлением, а скорее контроллером представления или пользовательского интерфейса, его следует настроить для отображения определенного представления `View` или дерева представлений. Для хранения экземпляров `View`, которые формируют изображение для `Fragment`, обычно используется один пустой контейнер `ViewGroup`, такой как `FrameLayout`.

Java

```
FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
transaction.add(R.id.my_view_group, fragment);
transaction.commit();
```

Kotlin

```
val transaction = supportFragmentManager.beginTransaction()
transaction.add(R.id.my_view_group, fragment)
transaction.commit()
```

`FragmentTransaction` может выполнять различные обновления для любых экземпляров `Fragment`, на которые имеются ссылки. Вот типичная последовательность действий: открыть транзакцию, внести все необходимые изменения и затем подтвердить транзакцию:

Java

```
FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
//transaction.add(R.id.my_layout, fragment);
//transaction.replace(R.id.my_layout, anotherFragment);
```

```
//transaction.remove(fragment);
//transaction.detach(fragment);
//transaction.attach(fragment);
//transaction.hide(fragment);
//transaction.show(fragment);
transaction.commit();
```

Kotlin

```
val transaction = supportFragmentManager.beginTransaction()
//transaction.add(R.id.my_layout, fragment)
//transaction.replace(R.id.my_layout, anotherFragment)
//transaction.remove(fragment)
//transaction.detach(fragment)
//transaction.attach(fragment)
//transaction.hide(fragment)
//transaction.show(fragment)
transaction.commit()
```

В отличие от `Activity`, класс `Fragment` не наследует `Context` и поэтому не имеет прямого доступа ко многим API; однако экземпляры `Fragment` имеют методы `getContext` и `getActivity`, чего в большинстве случаев вполне достаточно.

На момент написания этой книги компонент `Navigation` считался стабильным, однако некоторые связанные с ним возможности (например, пользовательский интерфейс редактора навигации – Navigation Editor UI) – нет. Существуют также некоторые противоречия, связанные с включением в Android перспективных инструментов генерации кода пользовательского интерфейса. Тем не менее компонент `Navigation` способен обрабатывать действия фрагмента `Fragment`, подобные предыдущим, без использования традиционных `FragmentTransaction` или `FragmentManager`.

Основные этапы жизненного цикла контроллера пользовательского интерфейса

По мере прохождения контроллера пользовательского интерфейса через различные состояния, от создания до завершения, вызывается множество его методов жизненного цикла, которые могут послужить отличным средством для получения событий приложения. Оба класса, `Activity` и `Fragment`, имеют свои события жизненного цикла (а также как экземпляры `View`, но они имеют довольно ограниченный круг событий, и мы не будем обсуждать их в этой главе).

В документации приводится подробная диаграмма (https://oreil.ly/LW_u1), описывающая жизненный цикл `Activity`, но здесь мы затронем только самые основные этапы.

На рис. 1.1 показана диаграмма из документации, которая послужит нам основой для дальнейшего обсуждения.

Когда экземпляр `Activity` создается впервые, вызывается его метод `onCreate`.

Важно понимать, что `onCreate` *вызывается также при повторном создании* `Activity`. Время от времени система будет отбирать и утилизировать ресурсы приложения, чтобы использовать их для других целей; в таких случаях приложение полностью уничтожается за кулисами, правда, некоторая информация о его текущем состоянии сохраняется на локальном диске.

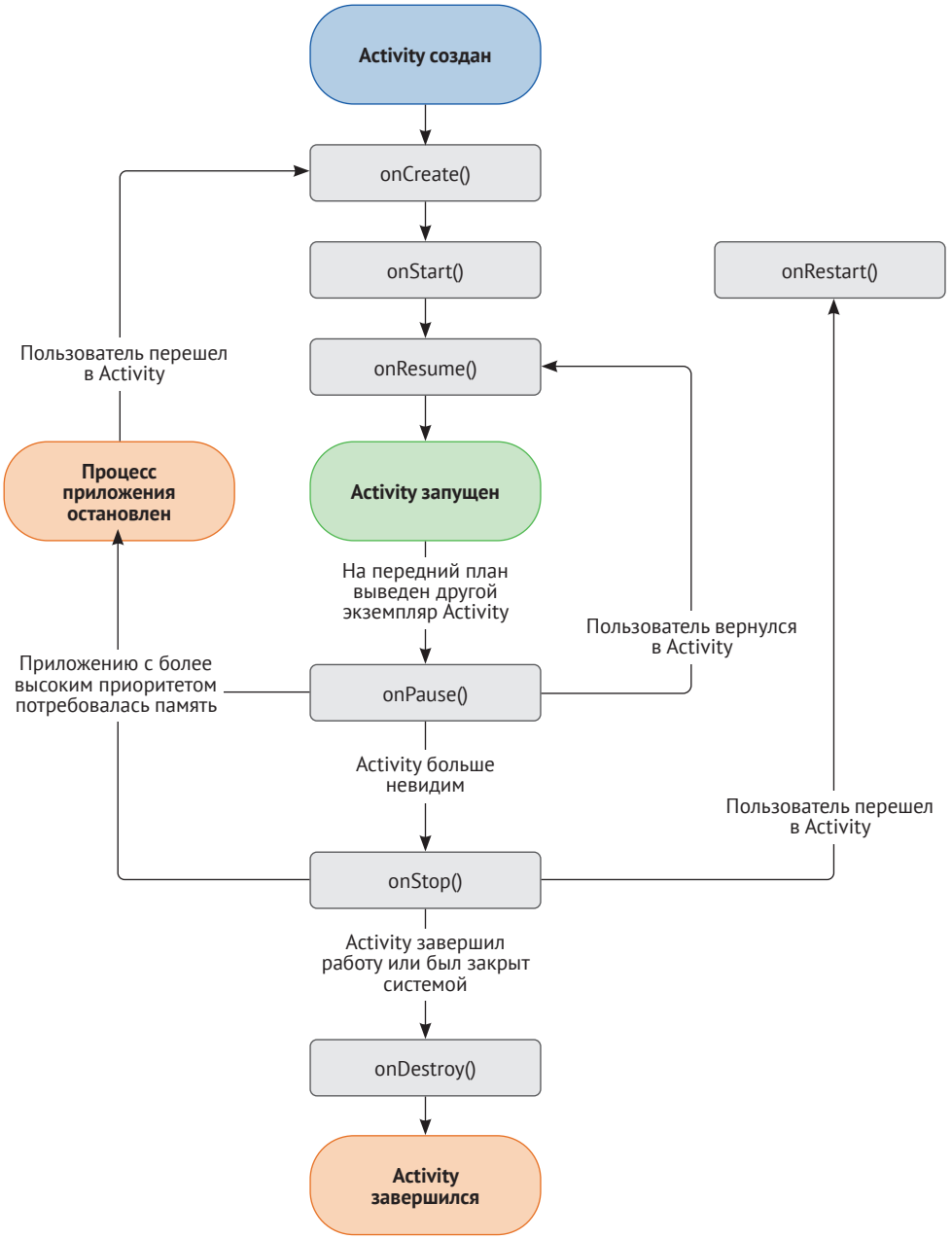

Рис. 1.1 ❖ Жизненный цикл Activity

Когда `Activity` создается в первый раз, этому методу передается единственный параметр – пустая ссылка на `Bundle`. Когда он воссоздается после утилизации ресурсов (как это происходит при «изменении конфигурации», например при повороте устройства или подключении нового экрана), методу `onCreate` будет передана непустая ссылка на `Bundle`.

Когда экземпляр Activity становится видимым (например, после появления из-за другого экземпляра Activity), вызывается метод onStart. Вызов onStart всегда следует за вызовом onCreate, но не всем вызовам onStart предшествует вызов onCreate.

Метод onResume вызывается каждый раз, когда Activity получает фокус ввода. Экземпляр Activity может потерять фокус, если содержащее его приложение будет свернуто, на передний план будет выведено другое приложение, произойдет телефонный звонок или даже если поверх содержимого Activity появится диалоговое окно, несмотря на то что большая часть этого содержимого все еще будет видна. После восстановления фокуса ввода – при закрытии другого приложения, прекращении телефонного звонка или закрытии диалога – будет вызван метод onResume. Метод onResume всегда следует за onStart, но не всем событиям onResume предшествуют события onStart.

Теперь пройдемся по другой ветке, ведущей к уничтожению.

Каждый раз, когда Activity теряет фокус ввода (см. onResume), вызывается метод onPause.

Ситуация с методом onStop сложнее, его смысл часто искажается в ходе обычного диалога. Метод onStop вызывается при уничтожении Activity, но он может быть воссоздан заново, например когда система отберет и вновь вернет ресурсы приложению. За onStop следует событие onDestroy (см. ниже) или onRestart, означающее, что Activity был восстановлен из сохраненных «подсказок» после остановки. Всем событиям onStop предшествует onPause, но не за всеми событиями onPause следует onStop. За более полной информацией об этом событии обращайтесь к документации (https://oreil.ly/POytI). Вот выдержка из документации:

> Когда Activity больше не виден пользователю, он переходит в состояние «приостановлен», и система вызывает onStop(). Это может произойти, например, когда вновь запущенный экземпляр Activity охватывает весь экран. Система может также вызвать onStop(), когда Activity завершил работу и вскоре будет уничтожен.

Метод onDestroy вызывается перед уничтожением экземпляра Activity, когда его повторное восстановление не предполагается. Если, находясь в Activity, вы коснетесь кнопки **Назад**, будет вызван метод onDestroy. Это подходящее место для освобождения ресурсов. Всем событиям onDestroy предшествует onStop, но не за всеми событиями onStop следует onDestroy.

В документации четко указано, что нельзя рассчитывать на onDestroy для освобождения объемных ресурсов или выполнения асинхронных операций. Это верно, но многие интерпретируют эти слова как то, что *можно* рассчитывать на onStop или onPause, однако это не так. Представьте, что ваше устройство попало под колесо грузовика (или, что более вероятно, разрядился аккумулятор). Ваше приложение будет закрыто немедленно, без вызова любых методов обработки событий жизненного цикла и без всяких шансов на освобождение ресурсов. Выполнять такую работу в onPause ничуть не безопаснее, чем в onDestroy. Тем не менее, поскольку вызов onDestroy, как правило, означает, что Activity вскоре будет уничтожен и утилизирован сборщиком мусора, можно не беспокоиться о занятых ресурсах, которые и так будут освобождены.

Экземпляры `Fragment` имеют очень похожий жизненный цикл, включающий дополнительные вызовы `onCreateView` (очень важный – значение, возвращаемое этим методом, представляющее экземпляр `View` с пользовательским интерфейсом этого фрагмента) и `onDestroyView`. Существуют также `onActivityCreated` и другие методы, которые вызываются, когда фрагмент добавляется (`onAttached`) в пользовательский интерфейс или удаляется из него (`onDetached`) с использованием методов `FragmentTransaction`.

Обратите внимание, что классы `Fragment`, `FragmentManager` и `FragmentTransaction` претерпели изменения с течением времени. Для согласованности и чтобы гарантировать совместимость с последними версиями ОС, мы рекомендуем использовать классы из библиотеки поддержки. В большинстве случаев они взаимозаменяемы – для этого достаточно просто импортировать `android.support.v4.app.Fragment` вместо `android.app.Fragment`; вызывая `new Fragment();`, вы получите `Fragment` из пакета библиотеки поддержки. Точно так же используйте `android.support.v7.app.AppCompatActivity` вместо `android.app.Activity`, который имеет метод `getSupportFragmentManager`, возвращающий `FragmentManager` с расширенным API, пригодным для использования с экземплярами `Fragment` из библиотеки поддержки.

Кроме того, доступны также версии AndroidX этих (и некоторых новых) классов, но на самом деле, даже спустя год разработки, их нельзя назвать стабильными (хотя среди них есть несколько классов, выпущенных с пометкой «стабильный»). Библиотеки Jetpack могут выполнять многие из этих функций, и Google предлагает использовать их в новых проектах, если это возможно, но не забывайте, что разработка с нуля – явление гораздо более редкое, чем сопровождение и дальнейшее развитие. Не стесняйтесь исследовать эти альтернативы, возможно, какие-то из них лучше подойдут для вас и вашей команды; мы (авторы) решили продолжать использовать имеющиеся у нас библиотеки и наборы инструментов просто потому, что они обеспечивают большинство необходимых нам возможностей. Но со временем ситуация обязательно изменится, поэтому, как в случае с любой технологией, старайтесь идти в ногу со временем и следовать последним рекомендациям.

iOS

UIKit, фреймворк пользовательского интерфейса, на который опираются почти все приложения для iOS, основан на архитектуре MVC. В iOS контроллер пользовательского интерфейса (символ «C» в аббревиатуре MVC) – это класс `UIViewController`. В типичном приложении есть несколько экземпляров и подклассов `UIViewController`, вместе осуществляющих управление поведением иерархии объектов, формирующих представления.

Как создать начальный контроллер пользовательского интерфейса приложения

Прежде чем углубиться в детали создания начального контроллера пользовательского интерфейса приложения, нам нужно обсудить представления, окна

и контроллеры, как они связаны с функциональностью, которую мы собираемся охватить.

Представления и контроллеры пользовательского интерфейса

Представления и контроллеры `UIViewController` в iOS неразрывно связаны друг с другом, поэтому перед обсуждением контроллеров важно обсудить представления. Более подробно представления рассматриваются в главе 2, но мы решили отметить их здесь, потому что корень иерархии контроллеров представлений в приложении начинается с единого свойства специализированного представления – окна приложения, экземпляра `UIWindow`. Каждое приложение для iOS имеет единственный экземпляр `UIWindow`, который представляет экземпляр приложения `UIApplication`. Свойство, где находится ссылка на корневой контроллер представления, имеет говорящее имя `rootViewController`. Запись ссылки на определенный контроллер в свойство `rootViewController` экземпляра `UIWindow` можно выполнить одной строкой кода:

```
window.rootViewController = viewController
```

Установка корневого контроллера представления почти всегда происходит во время запуска приложения, обычно в `application(_:didFinishLaunchingWithOptions:)`. Однако если в Xcode создать новый проект приложения с одним представлением (Single View Application), будет создан делегат приложения со следующим кодом в этом методе:

```
func application(_ application: UIApplication,
  didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) ->
  Bool {
    // Переопределите для настройки после запуска приложения
    return true
}
```

Обратите внимание, что нигде в теле этого метода не настраиваются свойства `rootViewController`. На самом деле здесь нет даже упоминания о `UIWindow` – только возврат значения `true`. И все же приложение запускает и отображает контроллер представления, созданный в раскадровке (storyboard), который, как кажется, нигде не настраивается. Выглядит очень таинственно.

В XCode нет никакого волшебства, но как тогда все это работает? Что ж, если вы заглянете в некоторые другие важные файлы в этом примере проекта XCode, вы довольно быстро раскроете тайну.

Расследование

Начнем наше детективное расследование с файла *Info.plist* в проекте. Это специальный файл, который определяется в настройках проекта XCode. Он содержит параметры конфигурации приложения в форме хорошо знакомых ключей XML. Вот как определяются значения свойств в этом файле:

```
<key>UIMainStoryboardFile</key>
<string>Main</string>
```

Элемент `key` с именем свойства `UIMainStoryboardFile` указывает, что определяется имя файла раскадровки (storyboard), который приложение должно ис-

пользовать при запуске. Значение этого свойства – `Main` – напрямую отображается в имя файла, в этом примере в файл с именем *Main.storyboard*. Продолжим расследование в данном файле.

Если открыть *Main.storyboard* в визуальном редакторе Xcode, мы увидим единственную сцену с большой стрелкой, указывающей на нее. Каждой сцене в раскадровке (storyboard) соответствует `UIViewController`, заданный в инспекторе идентичности (Identity inspector) в правой части экрана. По умолчанию это стандартный экземпляр `UIViewController`, но в инспекторе можно выбрать пользовательский подкласс, введя имя подкласса в поле `Class`. В нашем примере проекта используется собственный класс, установленный в View-Controller, который является подклассом `UIViewController` и определяется в файле *ViewController.swift* (рис. 1.2).

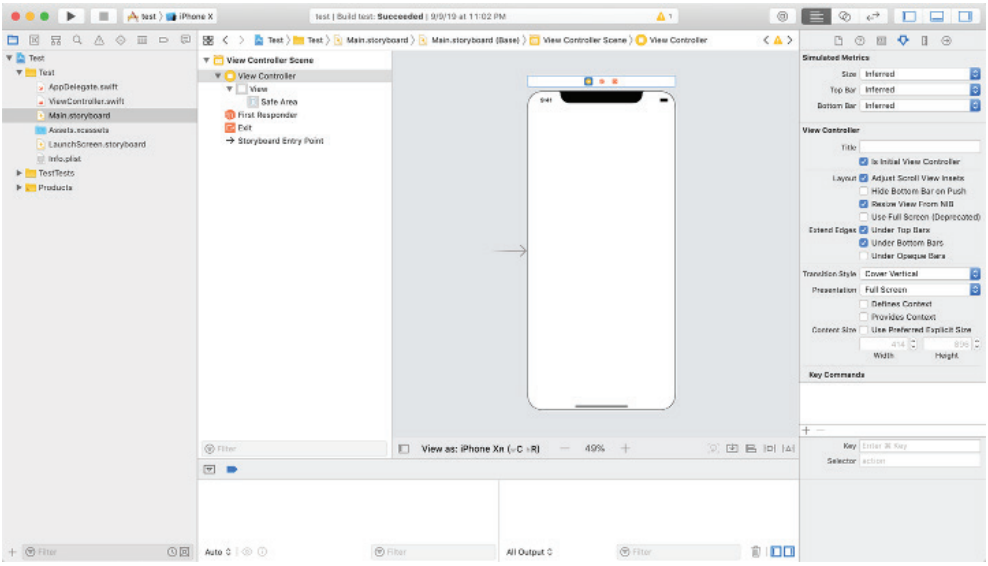

Рис. 1.2 ❖ Редактор раскадровки в Xcode

Теперь о большой стрелке слева от сцены контроллера представления: это просто «дымящийся ствол пистолета», на который мы наткнулись в ходе поиска корневого контроллера представления. В инспекторе атрибутов (Attributes inspector) в XCode есть флажок с подписью Is Initial View Controller (Является начальным контроллером представления), который в настоящий момент установлен. Если снять этот флажок, большая стрелка исчезнет. Создайте и запустите приложение, предварительно сняв флажок, и вы получите несколько предупреждений и следующую ошибку в консоли Xcode:

```
Failed to instantiate the default view controller
    for UIMainStoryboardFile 'Main' - perhaps the designated entry point is not
    set?
(Перевод:
Ошибка создания экземпляра контроллера представления по умолчанию
```

```
    для UIMainStoryboardFile 'Main' – возможно, требуемая точка входа не
    установлена?
)
```

Отлично! Мы узнали, откуда берется корневой контроллер представления. Но как объединить полученные знания, чтобы задать свой корневой контроллер представления в окне приложения?

Это просто. При запуске приложение ищет ключ `UIMainStoryboardFile` в своем файле *Info.plist*. Внутри главного файла раскадровки находит контроллер представления для сцены с установленным флажком Is Initial View Controller (Является начальным контроллером представления) и создает его. Поскольку этот начальный контроллер представления определен в главной раскадровке, приложение добавит его в свойство `rootViewController` окна приложения, и дело в шляпе! Теперь в приложении есть корневой контроллер, который отображается и активен.

При желании того же результата можно добиться, добавив следующий код внутрь делегата приложения:

```
func application(_ application: UIApplication,
  didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) ->
  Bool {
    window = UIWindow(frame: UIScreen.main.bounds)
    window?.rootViewController = UIStoryboard(name: "Main", bundle: nil).
                                 instantiateInitialViewController()
    window?.makeKeyAndVisible()
    return true
}
```

Рассмотрим его подробнее.

Сначала инициализируется переменная `window`, которая определена как часть протокола `UIApplicationDelegate`, и ей присваивается ссылка на экземпляр окна `UIWindow`, имеющего те же размеры, что и размеры главного и, вероятно единственного, экрана устройства (`UIScreen.main.bounds`). Затем назначается корневой контроллер. Это может быть любой имеющийся контроллер, но в данном примере мы используем начальный контроллер, как определено в файле *Main.storyboard*; для этого вызывается метод `instantiateInitialViewController()` объекта `UIStoryboard`.

Наконец, мы отображаем данное окно, вызывая `makeKeyAndVisible()`. Этот метод принимает объект окна и назначает его основным окном приложения, вытесняя все другие окна, отображаемые в данный момент.

Вообще говоря, в каждый конкретный момент приложения для iOS отображают только одно окно, но это *не всегда* так. Приложениям, которые должны выводить видео на другой экран, может потребоваться больше одного окна; хорошим примером могут служить приложения, подобные Keynote. Однако это, скорее, исключение из правил.

Какой способ лучше, код или раскадровка?

В настоящее время в любых простых приложениях рекомендуется использовать конфигурацию в *Info.plist* и главную раскадровку, как было описано выше. Од-

нако в более сложных приложениях удобнее может оказаться способ настройки в программном коде. Возможно также, что вы сами *предпочтете* настраивать контроллер в коде, а не в раскадровке. На самом деле нет по-настоящему «правильного» способа настройки начального контроллера пользовательского интерфейса; все сводится к личным предпочтениям и требованиям проекта.

Однако в процессе развития приложение с единственным контроллером пользовательского интерфейса быстро столкнется с ограничениями или станет невероятно сложным. Поэтому давайте теперь посмотрим, как переключить контроллер пользовательского интерфейса, отображаемый в данный момент, и как дать пользователю более богатые впечатления от работы с приложением.

Как изменить активный контроллер пользовательского интерфейса

В iOS есть несколько способов переключения активного контроллера пользовательского интерфейса. Одни из них реализуются в программном коде, а другие – за счет определения «переходов» в редакторе раскадровки. Скорее всего, вам встретятся оба подхода, нередко в одной и той же кодовой базе. Сначала рассмотрим способ реализации в программном коде – это поможет понять суть происходящего за кулисами и освоить магию переходов.

Шоу начинается!

Допустим, у нас есть два контроллера представлений: один с именем `primaryViewController` и другой с именем `secondaryViewController`. В этом примере текущим активным контроллером является `primaryViewController`. Самый простой способ отобразить `secondViewController` – воспользоваться методом `show(_:sender:)`, унаследованным от `UIViewController`. Вот как это делается:

```
// Создать контроллеры представлений
let primaryViewController = ...
let secondaryViewController = ...

// Назначить вторичный контроллер активным
primaryViewController.show(secondaryViewController, sender: nil)
```

В этом простом примере вызов метода `show(_:sender:)`, вероятно, приведет к тому, что `secondaryViewController` отобразится модально в нижней части экрана перед `primaryViewController`. Ключевое слово в предыдущем предложении – «вероятно». Мы не можем быть на 100 % уверены без дополнительного контекста – метод `show(_:sender:)` отделяет процесс отображения контроллера представления от контроллера представления, который вызывается для отображения. Большую часть времени такой вызов имеет простую логику. Например, рассмотрим следующий код, который не использует `show(_:sender:)`:

```
let primaryViewController = UIViewController(nibName: nil, bundle: nil)
let secondaryViewController = UIViewController(nibName: nil, bundle: nil)

// Добавить первичный контроллер представления в контроллер навигации
let _ = UINavigationController(rootViewController: primaryViewController)

...
```

```
// Убедиться, что контроллер представления является частью
// контроллера навигации
if let navigationController = primaryViewController.navigationController {
    // Втолкнуть контроллер представления в стек навигации
    navigationController.pushViewController(secondaryViewController, animated: true)
} else {
    // Отобразить контроллер представления модально,
    // потому что стека навигации не существует
    primaryViewController.present(secondaryViewController, animated: true, completion: nil)
}
```

Первое, что бросается в глаза, – новый класс `UINavigationController`. Этот класс помогает управлять стеком контроллеров представлений; обычно вталкивание в стек или выталкивание из стека навигационного контроллера визуально отображается как переход вбок, вправо или влево. Это, пожалуй, наиболее распространенный способ смены активного контроллера представления в iOS, уступающий, возможно, только использованию контроллера вкладок. В нашем предыдущем примере `primaryViewController` добавляется в корень стека навигации в `UINavigationController` при создании экземпляра.

Теперь, как показано в нашем примере `show-less`, допустим, что нам нужно добавить новый контроллер представления в стек и сделать его активным. Прежде чем сделать это, мы должны проверить, имеет ли свойство `navigationController` экземпляра `primaryViewController` значение `nil`. Если это не так, значит, контроллер представления является частью иерархии контроллера навигации, поэтому мы можем продолжить и добавить в стек новый контроллер представления, в данном случае `secondaryViewController`, получив ссылку из свойства `navigationController` и вызвав ее метод `push(_:animated:completion:)`. Но если текущий активный контроллер представления отсутствует в стеке контроллера навигации, мы должны отобразить новый контроллер представления другим способом. В этом примере мы используем более прямолинейный и старый способ, заключающийся в вызове `present(_:animated:completion:)`.

Только что показанный способ дает возможность более точного управления, но он значительно сложнее – и это только простой пример! Более того, `show(_:sender:)` позволяет внести некоторые изменения в отображение контроллера представления, как показано ниже:

```
let primaryViewController = UIViewController(nibName: nil, bundle: nil)
let secondaryViewController = UIViewController(nibName: nil, bundle: nil)

// Изменить стиль отображения и перехода
secondaryViewController.modalPresentationStyle = .formSheet
secondaryViewController.modalTransitionStyle = .flipHorizontal

// Сменить активный контроллер пользовательского интерфейса
primaryViewController.show(secondaryViewController, sender: nil)
```

Здесь с помощью свойства `modalPresentationStyle` изменяется состояние отображения контроллера представления, а с помощью `modalTransitionStyle` изменяется стиль перехода к этому состоянию. В данном примере устанавливается стиль отображения Form Sheet (лист формы), специально предназначенный для iPad, занимающий только часть экрана. Стиль перехода – переворот стра-

ницы по горизонтали, когда представление как бы переворачивается при появлении.

В iPhone или других устройствах размерного класса `.compact` стиль представления `.formSheet` игнорируется, а UIKit адаптирует его к полноэкранному отображению. На более крупных iPhone, таких как iPhone XS Max или iPhone 8 Plus, в альбомной ориентации стиль Form Sheet отображается так же, как на планшете, потому что в альбомной ориентации эти устройства имеют размерный класс `.regular`; в книжной ориентации те же устройства имеют размерный класс `.compact` и Form Sheet отображается в полноэкранном режиме, как на небольших устройствах. Мы обращаем ваше внимание на этот факт, потому что всегда есть исключения и крайние случаи. Важно проводить тестирование на самых разных симуляторах или устройствах.

Мы лишь слегка коснулись темы переключения активного контроллера представления программным способом. Прежде чем двинуться дальше, рассмотрим альтернативный способ, основанный на использовании конфигурации, который в iOS называется переходы (segues).

Переходы

Смену активного контроллера, показанную выше в коде, можно реализовать внутри раскадровки с использованием переходов. Переходы определяют связь между двумя контроллерами представлений и используются для отображения контроллеров представлений в приложении. Самый простой способ определить переход – воспользоваться редактором раскадровки в Xcode.

Чтобы создать новый переход, сначала нужно получить две сцены с контроллерами представлений, между которыми будет осуществляться переход. Удерживая клавишу **Ctrl**, щелкните на сцене с исходным контроллером представления и перетащите указатель мыши на целевой контроллер представления в редакторе раскадровки. При этом целевая сцена будет выделена синим цветом, что помогает визуально контролировать выбор целевой сцены. Отпустите кнопку мыши, и перед вами появится всплывающее окно, где можно выбрать тип перехода. На выбор будут представлены уже знакомые варианты: с использованием `show(_:sender:)` за кулисами и возможностью для UIKit определить лучший переход или явно использовать модальный переход, кроме всего прочего.

После создания перехода, если он связывает контроллеры представлений, его нужно вызвать программно. Для этого щелкните на самом переходе (например, на линии, соединяющей сцены в раскадровке), откройте инспектор атрибутов и добавьте уникальный идентификатор. В нашем примере будем использовать имя `ExampleSegue`.

Идентификаторы переходов должны быть уникальными в пределах раскадровки, где определяются контроллеры представлений.

Вызов перехода осуществляется так:

```
primaryViewController.performSegue(withIdentifier: "ExampleSegue", sender: nil)
```

Метод `performSegue(withIdentifier:sender:)` принимает строку (`ExampleSegue`, как мы определили выше) и отправителя `sender`, которым может быть любой

объект. Обычно во втором аргументе передается ссылка на кнопку, если переход вызван нажатием кнопки, но допускается передать `nil`, как сделано в этом примере.

Также можно подключить кнопку или другой элемент управления, чтобы явно вызвать переход. Это делается с помощью того же механизма **Ctrl**+щелчок в редакторе раскадровки, но, вместо того чтобы щелкать и перетаскивать всю сцену, можно щелкнуть и перетащить определенную кнопку в исходном контроллере представления. Такой подход облегчит задачу, потому что избавит от необходимости вызывать переход программно, с использованием `performSegue(withIdentifier:sender:)`.

Иногда при переходе между контроллерами представлений требуется передать дополнительные данные. Контроллеры представлений имеют ряд специальных методов, которые вызываются всякий раз, когда выполняется переход, что позволяет передавать данные или состояние и помогает настроить целевой контроллер представления или выполнить некоторое действие. Вот пример контроллера представления, отображающего другой контроллер представления с идентификатором `ExampleSegue`, который мы определили выше:

```
class ViewController: UIViewController {

  func buttonPressed(button: UIButton) {
    // Код для вызова перехода. То же самое можно реализовать более
    // непосредственно в редакторе раскадровки
    performSegue(withIdentifier: "ExampleSegue", sender: button)
  }

  override func shouldPerformSegue(withIdentifier identifier: String,
    sender: Any?) -> Bool {
      // Необязательный метод, по умолчанию возвращающий true
      // Вернув false, можно отменить переход
      return true
  }

  override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // Получить ссылку на целевой контроллер представления
    let destinationViewController = segue.destination

    // Передать ему некоторые данные
    ...
  }
}
```

В этом примере в подклассе `UIViewController` определен метод `buttonPressed(_:)`, который вызывается при каждом нажатии кнопки. Этот код вызывает `performSegue(withIdentifier:sender:)`, чтобы выполнить переход. (То же самое можно реализовать, напрямую связав кнопку с целевым контроллером представления в редакторе раскадровки, но здесь мы решили явно показать, что происходит в действительности.)

Далее перед началом перехода вызывается метод `shouldPerformSegue(withIdentifier:sender:)`. Это необязательный метод в контроллере представления, который можно переопределить, чтобы выполнить какие-либо проверки, прежде чем принять решение о том, следует ли выполнять переход. По умолчанию воз-

вращается значение `true` – переход разрешен. Целевой контроллер представления не будет создан до вызова этого метода. Вернув `false`, можно отменить переход, и больше ничего не произойдет. На практике метод `shouldPerformSegue(withIdentifier:sender:)` редко используется для отмены переходов, однако иногда такая возможность может пригодиться.

Следующее и последнее событие в цепочке: `prepare(for:sender:)`. На этом этапе целевой контроллер представления уже создан и находится на расстоянии одного шага от появления. Это последний шанс исходного контроллера представления передать некоторую информацию о состоянии или контекст, которая может помочь целевому контроллеру во время или после перехода.

Теперь мы знаем, как создать и настроить начальный контроллер представления в приложении и как производить смену активного контроллера с помощью переходов. Давайте отступим на шаг назад и рассмотрим жизненный цикл контроллера представления в iOS.

Основные этапы жизненного цикла контроллера пользовательского интерфейса

Создать контроллер пользовательского интерфейса в iOS можно несколькими способами, но на практике для этих целей чаще всего используется раскадровка.

Создание контроллеров пользовательского интерфейса из раскадровки

Чтобы создать контроллер представления из раскадровки, сначала нужно определить сцену контроллера представления в раскадровке. Это можно сделать в Xcode, добавив контроллер представления на стадии редактирования. После этого обязательно откройте инспектор идентичности (Identity inspector) и добавьте любой пользовательский подкласс в поле Class. Кроме того, присвойте контроллеру представления конкретный идентификатор раскадровки. Этот идентификатор используется для выбора сцены контроллера представления, которая будет применяться при создании контроллера представления из раскадровки программным способом. Обычно идентификатором служит имя класса, например:

```
let viewController = UIStoryboard(name: "Main", bundle: nil).
    instantiateViewController(withIdentifier: "ExampleViewController")
```

Несмотря на простоту использования строк, применяя их, вы можете быстро потерять контроль. Чтобы этого не произошло и для предотвращения проблем сопровождения в будущем идентификаторы раскадровки лучше хранить отдельно – в константной структуре, в перечислении или с применением любой другой абстракции, обеспечивающей безопасность на этапе компиляции.

Когда контроллеры представлений создаются через раскадровку, UIKit использует специальный метод (его можно переопределить в своем классе), помогающий выполнить инициализацию, – метод `init(coder:)`, который является лучшим местом для настройки и конфигурации представления перед загруз-

кой в класс и размещением в иерархии контроллеров представления. Вот как можно переопределить этот метод:

```
class ViewController: UIViewController {

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)

        // Выполнить дополнительные настройки
    }

}
```

Метод `init(coder:)` легко переопределить, но ему нельзя передать нестандартные параметры. Изменить свойства контроллеров представлений намного проще в Android, при инициализации через конструктор объекта, чем в iOS с использованием раскадровок. Обычно изменение свойств выполняется непосредственным присваиванием значений или вызовом метода настройки после создания экземпляра представления. Оба подхода имеют свои плюсы и минусы, и часто в проектах используются оба.

Жизненный цикл контроллера пользовательского интерфейса тесно связан с жизненным циклом представления, которым он управляет. Помимо события инициализации, контроллер представления получает еще множество событий от представления и других объектов, которые контролируют его самого, помогают упростить управление представлением и другими зависимыми объектами. Давайте поговорим о некоторых из них.

viewDidLoad

Этот метод вызывается после загрузки представления и только один раз в течение всего жизненного цикла контроллера. Он отлично подходит для настройки представления. Все выходы (outlets) и действия, описанные внутри раскадровки, уже подключены и готовы к использованию. Как правило, в этом методе выполняются такие операции, как установка цвета фона представления, шрифтов меток и другие стилистические операции. Иногда здесь настраиваются уведомления (см. главу 11). Если вы тоже собираетесь выполнить такие настройки, обязательно отмените подписку на уведомления в `deinit` или другим способом, чтобы предотвратить сбои или утечки памяти.

viewWillAppear и viewDidAppear

Эта пара методов вызывается до и после включения представления в иерархическое дерево представлений. На этом этапе обычно уже известен размер представления (но не всегда – размер модальных представлений остается неизвестным до вызова `viewDidAppear`), и здесь можно выполнить окончательную коррекцию размера. Это также хорошее место для подключения ресурсов, интенсивно использующих память или процессор, таких как GPS или акселерометр.

viewWillDisappear и viewDidDisappear

Эти методы похожи на `viewWillAppear` и `viewDidAppear`, но вызываются до и после удаления представления из иерархии представлений, когда оно уже не отобра-

жается. Это отличное место для отключения ресурсов, подключенных в предыдущей паре методов.

Интерактивные жесты «возврат назад» (swipe back), выполняемые пользователем, не приводят к вызову `viewDidDisappear`. Обязательно протестируйте это поведение, касаясь системной кнопки **Назад** и выполняя жест «возврата назад» для выталкивания представления с экрана.

didReceiveMemoryWarning

Обработка предупреждений о нехватке памяти играет важную роль в iOS, потому что мобильные устройства могут иметь очень ограниченный объем памяти. Получив предупреждение, удалите ненужное кеширование ресурсов, очистите выходы, созданные из раскадровки, и т. д. Если этого окажется недостаточно, рано или поздно приложение потерпит сбой и будет закрыто.

Вот пример класса, обрабатывающего все события, перечисленные выше:

```
class ViewController: UIViewController {
    var hugeDataFile: Any?

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        // Здесь настраиваются операции, не зависящие от представления
        // Например, настройка объекта hugeDataFile для работы в фоновом режиме
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Представление уже загружено из раскадровки
        title = "Awesome View Controller Example"
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        // Представление готово к отображению на экране
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)
        // Представление появилось на экране
    }

    override func viewWillDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        // Представление готово к удалению с экрана
    }

    override func viewDidDisappear(_ animated: Bool) {
        super.viewDidDisappear(animated)
        // Представление удалено с экрана
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
```

```
        // Ой! Лучше освободить память, занятую этим большим файлом
        hugeDataFile = nil
    }
}
```

Обратите внимание, что все методы снабжены спецификатором `override` и вызывают унаследованные реализации в суперклассе. *Это важно*, потому что иначе последующие контроллеры представлений в иерархии не получат соответствующих событий. Обсуждение, почему эта ситуация не обрабатывается компилятором, как вызовы `retain` и `release`, выходит за рамки данной книги. Просто не забывайте включить эти вызовы методов в свои переопределенные версии!

Обе системы, Android и iOS, поддерживают архитектуру MVC. Иногда ее уничижительно называют «Massive View Controller» (массивный контроллер представления), потому что при неосторожном применении логика управления представлениями выливается в определения классов в тысячи строк. Важно стараться писать классы, следуя принципу единственной ответственности, и правильно использовать контейнеры представлений.

Контроллеры навигации, вкладок и раздельных представлений

В iOS есть специальные классы с особым поведением, специально предназначенные для управления контроллерами представлений. Вы непременно столкнетесь с тремя из них: контроллерами навигации (`UINavigationController`), контроллерами панелей вкладок (`UITabBarController`) и контроллерами раздельных представлений (`UISplitViewController`).

Контроллеры навигации поддерживают стеки контроллеров представлений и обеспечивают их согласованную работу, упрощают пространственную навигацию и ее реализацию, по сравнению с серией контроллеров модальных представлений, визуально расположенных друг над другом.

Контроллеры панелей вкладок – это специальный класс, управляющий активными контроллерами представлений с помощью закрепленной панели вкладок в нижней части экрана. Это распространенный метод сегментирования разделов в приложениях (например, вкладки **Поиск**, **Оформление заказа** и **Заказы** в приложении для покупок).

Контроллеры раздельных представлений первоначально появились в iPad, а затем мигрировали в iPhone. Они используются для отображения основного набора данных, как правило, в виде списка, и более подробной информации о выбранном элементе в основном наборе.

showDetail(_:sender:)

Класс `UISplitController` позволяет вместо `show(_:sender:)` переопределить метод `showDetail(_:sender:)` для вывода сведений в контроллере представления детальной информации. Этот контроллер адаптируется к полноэкранному модальному представлению, когда `UISplitController` недоступен на устройстве (например, на устройствах с размерным классом `.compact`, таких как iPhone с маленьким экраном размера).

Что мы узнали

В этой главе мы подробно рассмотрели контроллеры пользовательского интерфейса:

- поговорили о различных архитектурах, поддерживаемых в Android и iOS, и показали, как `Activity` накладывается на `UIViewController`;
- рассмотрели логику отображения представления на экране при запуске приложения на обеих платформах. Выяснили, что Android предлагает больше возможностей для настройки, по сравнению с более традиционным подходом в iOS;
- рассмотрели переходы между сценами и смену активного представления, а также некоторые инструменты, доступные в Android, такие как объекты `Fragment`, помогающие упростить управление представлениями;
- обсудили разные методы жизненного цикла контроллеров пользовательского интерфейса в Android и iOS;
- познакомились с раскадровками (storyboards) в iOS и их ролью в соединении различных сцен.

Удивительно, что даже при таком широком охвате многое осталось неосвещенным. Подробнее о некоторых деталях представлений, не имеющих отношения к контроллерам пользовательского интерфейса, мы поговорим в главе 2. Кроме того, во второй части книги мы представим пример создания приложения для обеих платформ и приведем дополнительную информацию.

Если вы готовы продолжить исследование представлений, переходите к следующей главе!