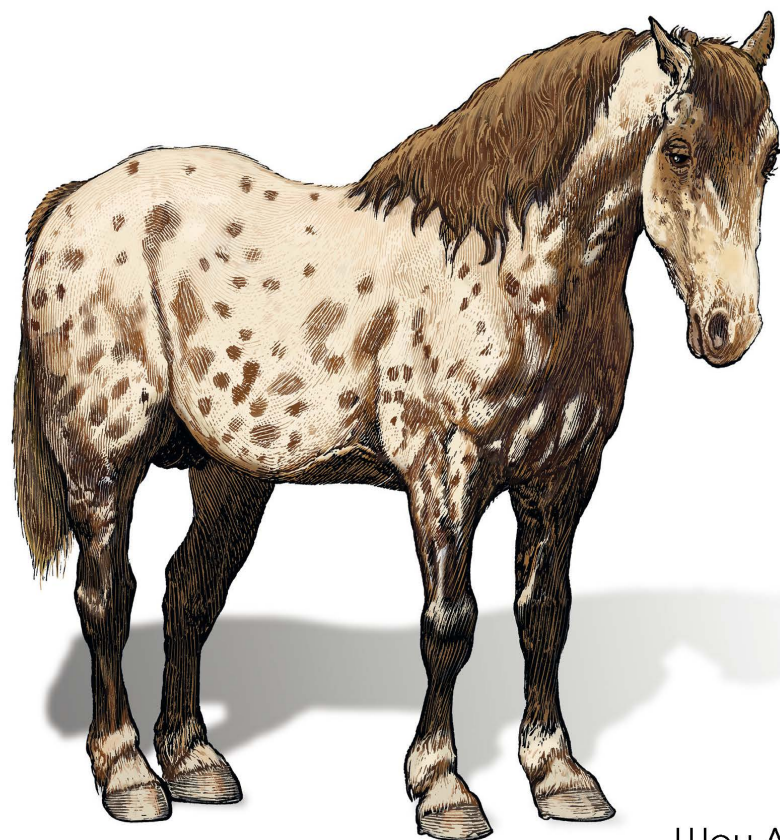




O'REILLY®

Нативная разработка мобильных приложений

Перекрестный справочник
для iOS и Android



Шон Льюис
Майк Данн

ДМК
ИЗДАТЕЛЬСТВО

Нативная разработка
мобильных приложений

O'REILLY®

Шон Льюис и Майк Данн

Нативная разработка мобильных приложений

Native Mobile Development

**A Cross-Reference
for iOS and Android**

Shaun Lewis and Mike Dunn

Нативная разработка мобильных приложений

Перекрестный справочник
для iOS и Android

Шон Льюис и Майк Данн



Москва, 2020

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

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same. Russian language edition copyright © 2020 by DMK Press. Authorized Russian translation of the English edition of Native Mobile Development ISBN 9781492052876. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-492-05287-6 (англ.)

ISBN 978-5-97060-845-6 (рус.)

Copyright © 2020 Shaun Lewis
and Mike Dunn

© Оформление, издание, перевод,
ДМК Пресс, 2020

Содержание

Об авторах	11
О колофоне	12
Вступление	13
Часть I. ЗАДАЧИ И ОПЕРАЦИИ	17
Примечание о текущем положении дел в сфере разработки мобильных приложений	17
Глава 1. Контроллеры пользовательского интерфейса	19
Задачи	19
Android	20
Как создать начальный контроллер пользовательского интерфейса приложения	20
Как изменить активный контроллер пользовательского интерфейса	22
Основные этапы жизненного цикла контроллера пользовательского интерфейса	27
iOS	30
Как создать начальный контроллер пользовательского интерфейса приложения	30
Как изменить активный контроллер пользовательского интерфейса	34
Основные этапы жизненного цикла контроллера пользовательского интерфейса	38
Что мы узнали	42
Глава 2. Представления	43
Задачи	43
Android	43
Создание нового представления	44
Вложение представлений друг в друга	49
Изменение состояния представлений	50
iOS	50
Создание нового представления	51
Вложение представлений друг в друга	53
С помощью Interface Builder	56
Изменение состояния представлений	57
Изменение позиции	58
Что мы узнали	59

Глава 3. Пользовательские компоненты	60
Задачи.....	60
Android.....	60
Как создать свое представление.....	61
Как использовать свое представление.....	66
iOS.....	68
Как создать свое представление.....	68
Как использовать свое представление.....	70
Что мы узнали.....	72
Глава 4. Пользовательский ввод	73
Задачи.....	73
Android.....	73
Получение события касания и реакция на него.....	74
Получение события ввода с клавиатуры и реакция на него.....	78
Обработка сложных жестов.....	81
iOS.....	84
Получение события касания и реакция на него.....	84
Получение события ввода с клавиатуры и реакция на него.....	85
Обработка сложных жестов.....	87
Что мы узнали.....	89
Глава 5. Передача сообщений	90
Задачи.....	90
Android.....	90
Использование обратных вызовов для реакции на действия.....	91
Передача сообщений подписчикам, заинтересованным в их получении.....	95
Получение и обработка сообщений.....	96
iOS.....	98
Использование обратных вызовов для реакции на действия.....	98
Передача сообщений подписчикам, заинтересованным в их получении.....	104
Получение и обработка сообщений.....	106
Замыкания вместо селекторов.....	107
Отмена подписки на уведомления.....	108
Что мы узнали.....	109
Глава 6. Файлы	111
Задачи.....	111
Android.....	111
Определение характеристик файла, таких как размер или дата последнего изменения.....	112
Чтение и запись данных в файлы.....	113

Копирование данных из одного файла в другой.....	118
iOS.....	119
Определение характеристик файла, таких как размер или дата последнего изменения.....	119
Чтение и запись данных в файлы.....	122
Копирование данных из одного файла в другой.....	123
Что мы узнали.....	125
Глава 7. Хранение данных	126
Задачи.....	126
Android.....	126
Соединение с базой данных.....	127
Создание таблицы или хранимого объекта.....	128
Запись данных в таблицу или хранимый объект.....	129
Чтение данных из таблицы или хранимого объекта.....	130
iOS.....	133
Настройка соединения со слоем хранения данных.....	133
Определение и создание таблицы или хранимого объекта.....	135
Запись хранимых данных в SQLite.....	136
Чтение данных из SQLite.....	137
Что мы узнали.....	138
Глава 8. Конкурентное (многопоточное) выполнение	140
Задачи.....	140
Android.....	140
Запуск задачи в фоновом потоке.....	141
Передача результатов из фонового потока в главный.....	144
Завершение потока выполнения.....	145
iOS.....	150
Запуск задачи в фоновом потоке.....	151
Передача результатов из фонового потока в главный.....	152
Что мы узнали.....	153
Глава 9. Сетевые взаимодействия	155
Задачи.....	155
Android.....	156
Загрузка текстового файла с удаленного сервера и его вывод.....	156
Создание запроса HTTP POST.....	157
Загрузка двоичного файла.....	159
iOS.....	160
Загрузка текстового файла с удаленного сервера и его вывод.....	161
Создание запроса HTTP POST.....	162
Загрузка двоичного файла.....	166
Что мы узнали.....	171

Глава 10. Обратная связь с пользователем	172
Задачи.....	172
Android.....	172
Отображение обратной связи с использованием системных инструментов.....	172
Snackbar	173
Изменение строки состояния	175
iOS.....	177
Отображение обратной связи с использованием системных инструментов	177
Изменение строки состояния	179
Что мы узнали.....	180
Глава 11. Предпочтения пользователя	182
Задачи.....	182
Android.....	182
Сохранение предпочтений пользователя.....	183
Чтение предпочтений пользователя.....	184
Работа с предпочтениями в многопользовательских приложениях	184
iOS.....	185
Сохранение предпочтений пользователя.....	185
Чтение предпочтений пользователя.....	188
Работа с предпочтениями в многопользовательских приложениях	189
Что мы узнали.....	190
Глава 12. Сериализация и транспорты	192
Задачи.....	192
Android.....	192
Сериализация и десериализация экземпляров объектов.....	192
iOS.....	200
Сериализация и десериализация экземпляров объектов.....	200
Дополнительные замечания для iOS.....	205
Что мы узнали.....	206
Глава 13. Расширения	207
Задачи.....	207
Android.....	207
Добавление новых возможностей в существующие API	207
iOS.....	209
Добавление новых возможностей в существующие API	209
Что мы узнали.....	212
Глава 14. Тестирование	213
Задачи.....	213
Android.....	213

Как писать и запускать модульные тесты.....	217
Как писать и запускать интеграционные тесты.....	220
iOS.....	222
Как писать и запускать модульные тесты.....	222
Что мы узнали.....	225
Часть II. ПРИМЕР ПРИЛОЖЕНИЯ	226
Глава 15. Добро пожаловать и настройка окружения	227
Сравнение нативных и кросс-платформенных инструментов разработки мобильных приложений	227
Веб-разработка	228
Другие подходы	228
Настройка окружения.....	229
Настройка окружения разработки для Android	229
Настройка окружения разработки для iOS	235
Что мы узнали.....	236
Глава 16. Создание приложения	237
Создание нового проекта.....	237
Android Studio	238
Xcode.....	242
Архитектура приложения	245
Создание первого экрана.....	246
Android	247
iOS.....	249
Что мы узнали.....	254
Глава 17. Вывод списка с данными	255
Оформление представлений	255
Android	255
iOS.....	265
Добавление кнопки	270
iOS.....	271
Списки, списки и еще раз списки!.....	271
Добавление нового представления каталога.....	272
Подключение кнопки	273
Книги	274
Заполнение представления списка	278
Android	278
iOS.....	285
Что мы узнали.....	288
Глава 18. Моделирование каталога библиотеки.....	290
Динамические данные в представлениях списков	290
Android	291

iOS.....	294
Пришло время вернуть объекты модели в реальность.....	298
JSON для одного, JSON для всего	298
Переключение слоя данных на использование JSON.....	300
Android	300
iOS.....	306
Что мы узнали.....	308
Глава 19. Сохранность данных.....	309
Детализация информации о книгах.....	309
Android	309
iOS.....	314
Сохранение книг для последующего использования.....	318
Android	319
iOS.....	320
Запись книг в хранилище.....	321
Android	322
iOS.....	331
Сохранение книг в закладках	339
Android	340
Что мы узнали.....	341
Глава 20. Сетевые операции в приложении.....	342
Поиск в сети	342
Android	345
iOS.....	347
Создание службы поиска.....	350
Установка Node и Express	350
Файл JSON с местоположениями библиотек	351
Вызов службы.....	352
Android	352
iOS.....	356
Что мы узнали.....	365
Предметный указатель	366

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 пространственных 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).

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. Поточковый 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 широко-вещательные 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:supportRtl="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 представление `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 `FragmentManager` 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

```
FragmentManager 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()
```

`FragmentManager` 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

```
FragmentManager 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.

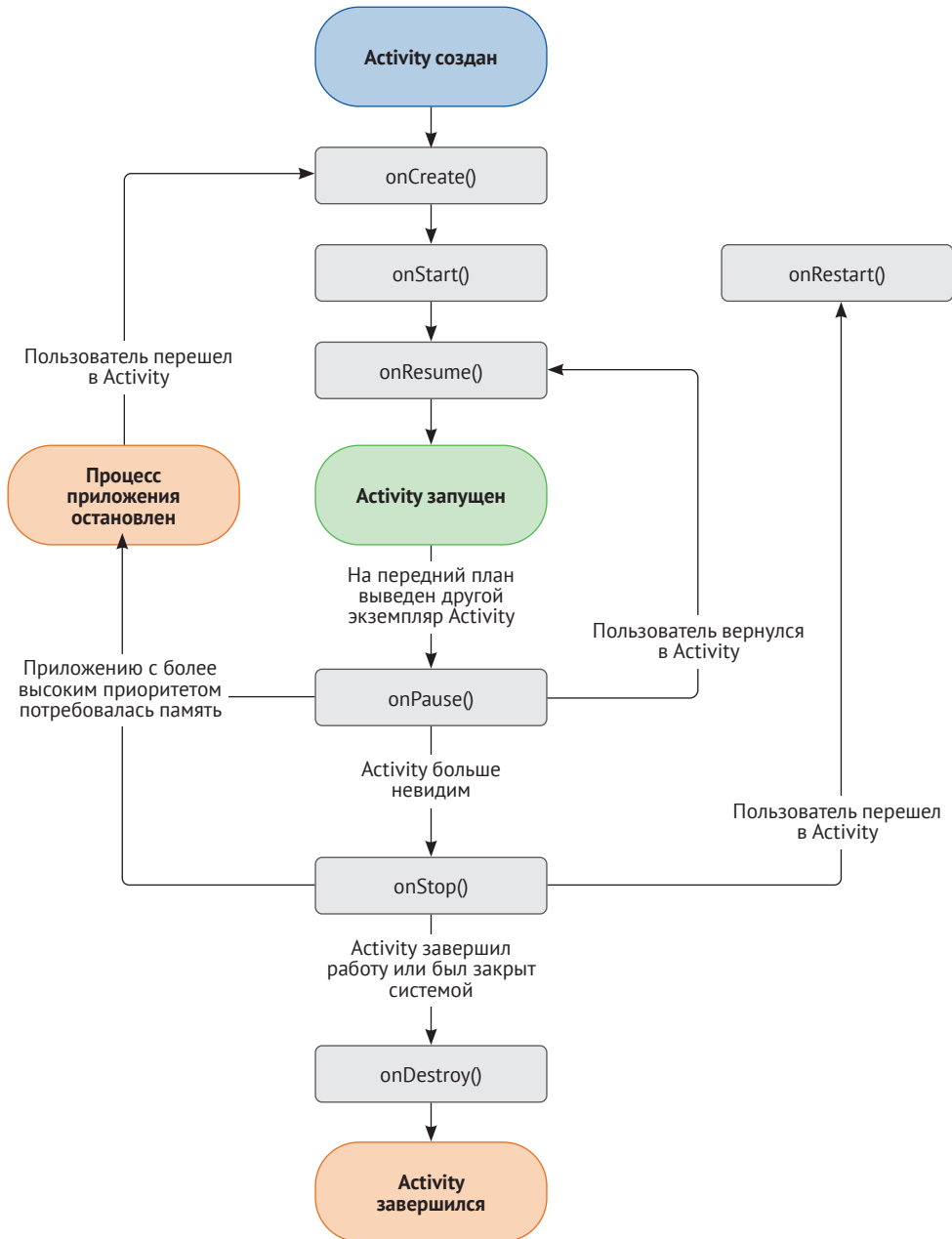


Рис. 1.1 ❖ Жизненный цикл Activity

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/POytl>). 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 `FragmentManager`.

Обратите внимание, что классы `Fragment`, `FragmentManager` и `FragmentManager` 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 *View-Controller.swift* (рис. 1.2).

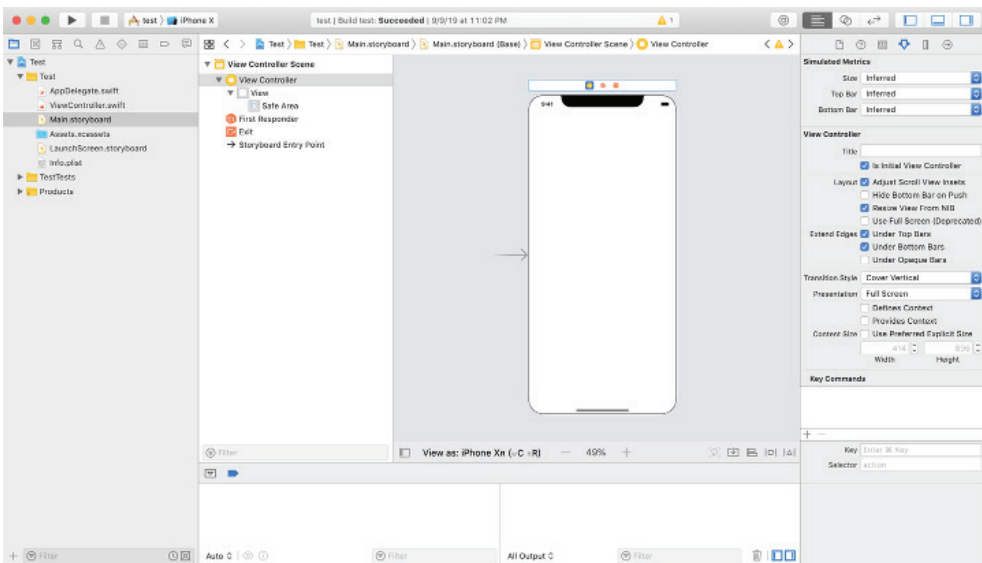


Рис. 1.2 ❖ Редактор раскадровки в Xcode

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 UIStoryboardFile '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 `UINavigationController`.

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 `secondaryViewController` – 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 визуальное 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.

i 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 сбой 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 получают 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 отдельных 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 отдельных 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 отдельных 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!

Глава 2

Представления

В большинстве фреймворков графического пользовательского интерфейса (GUI) визуальные элементы на экране представлены объектами, которые часто называют «представлениями». В веб-приложениях представление может быть элементом HTML, но в некоторых веб-платформах представлению соответствует целая веб-страница или ее фрагмент. В Java Swing и некоторых других фреймворках для обозначения части представления приложения используется термин «визуальные компоненты».

В нативной разработке мобильных приложений представлением является экземпляр подкласса `View` (Android) или `UIView` (iOS).

Представления могут быть атомарными и состоять из единственного визуального компонента, например фрагмента текста или изображения, но также могут иметь более сложную иерархическую структуру, как, например, список строк или еще сложнее, как, допустим, виджет календаря со сложным поведением, встроенным в сам виджет.

Представления также используются для получения пользовательского ввода. Кнопки, переключатели, флажки, раскрывающиеся списки и поля ввода текста – все это примеры представлений, которые дают пользователю возможность взаимодействовать с вашим приложением.

Задачи

В этой главе вы узнаете:

- 1) как создать новое представление;
- 2) как вкладывать представления друг в друга;
- 3) как обновлять состояние представлений.

ANDROID

В Android базовым классом является класс `View`, и это *не абстрактный* класс – вы можете создать экземпляр `View`, хотя необходимость в этом возникает нечасто. Простые экземпляры `View` обычно используются как линии или фигуры, для предоставления размеров и цвета фона, или как области для определения попаданий без визуального представления. В числе распространенных примеров подклассов `View` в Android можно назвать `TextView`, `ImageView`, `Button` и `EditText`.

Базовый класс `View` не может содержать других представлений, для этого предназначен иной класс: `ViewGroup` (`ViewGroup` наследует класс `View`, поэтому сам является представлением). В библиотеке Android есть несколько классов, наследующих `ViewGroup`, которые предназначены только для формирования макета: `LinearLayout`, `FrameLayout`, `ConstraintLayout` и др. Эти подклассы `ViewGroup` способны удовлетворить основные требования к макетам, но, поскольку они являются классами Java, вы можете создавать свои подклассы `ViewGroup` и определять в них свою логику; я делаю это очень часто.

В числе других подклассов `ViewGroup`, не имеющих прямого отношения к макетам, можно назвать `ScrollView`, `RecyclerView`, `Spinner` и `ViewPager`. Каждый из этих визуальных компонентов обладает широкими возможностями, помимо отображения и размещения контента. Например, `RecyclerView` управляет прокручиваемым списком элементов, которые удаляются с экрана (и из памяти) после выхода за пределы экрана и «воссоздаются» («recycled», чем и обусловлено такое имя) после перемещения в видимую область. В экосистеме Android их неофициально называют «виджетами» (не путайте с виджетами домашнего экрана устройства).

Создание нового представления

Экземпляры классов представлений создаются точно так же, как и любые другие объекты, – с использованием ключевого слова `new` и конструктора подкласса `View` (в Java) или простым вызовом конструктора (в Kotlin). В большинстве случаев для этого потребуется, как минимум, параметр `Context`, а в некоторых может понадобиться передать другие параметры:

Java

```
TextView textView = new TextView(context);
```

Kotlin

```
val textView = TextView(context)
```

Но чаще экземпляры `View` и, как правило, иерархии вложенных экземпляров `View` и `ViewGroup` создаются методом развертывания «макета» XML. Под «развертыванием» подразумевается самый простой анализ разметки XML и добавление дерева представлений в существующий пользовательский интерфейс или его возврат.

Макеты XML в Android оформляются в соответствии с традиционными правилами и парой соглашений, о которых вы должны знать.

Объявление XML необязательно, но желательно:

```
<?xml version="1.0" encoding="utf-8"?>
```

Далее должен следовать *один* корневого узла XML. Это почти всегда `ViewGroup` (если определяется дерево представлений) или один `View`, представляющий все содержимое макета. Я говорю «почти», потому что существуют продвинутые механизмы, такие как тег `merge`; технически это не `View`, но он содержит инструкции, определяющие содержимое узла.

Например, вам может встретиться такое дерево представлений:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout...>
  <android.support.design.widget.AppBarLayout...>
    <android.support.v7.widget.Toolbar... />
  </android.support.design.widget.AppBarLayout>
  <FrameLayout...>
  </FrameLayout>
</LinearLayout>
```

или единственное представление:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView... />
```

Файлы макетов хранятся в папке *res/layout/* и должны соответствовать стандартной схеме именования ресурсов в Android (только алфавитно-цифровые символы и символы подчеркивания). Эта папка обрабатывается во время компиляции, и для каждого макета создается ссылка с числовым идентификатором. Сейчас совершенно не важно, как именно это происходит; просто знайте, что если сохранить файл макета как *res/layout/my_activity.xml*, он будет доступен в глобальном статическом объекте конфигурации R как *R.layout.my_activity*, который можно передать любому методу, ожидающему получить идентификатор ресурса.

Вернемся к соглашениям, упоминавшимся выше. Процесс развертывания макетов в Android обладает обширными возможностями, но для доступа к ним требуется использовать пространства имен. Это достаточно просто: добавьте в корневой узел атрибут пространства имен, ссылающийся на схему Android: <http://schemas.android.com/apk/res/android>. Это пространство имен может быть чем угодно, но вообще его принято называть «android»:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  ...>
```

После этого у вас появится доступ к свойствам платформы Android, которые можно использовать, добавляя их после пространства имен, как показано ниже:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  ...>
  <TextView
    android:text="Hello world!"
    ...>
```

После определения пространства имен, когда платформа будет наполнять макет из XML, она создаст *LinearLayout* с *TextView* в нем, и свойство *text* в *TextView* получит значение «Hello World!» (которое будет отображаться на экране системным шрифтом, с цветом и размером по умолчанию).

Обратите внимание, что вполне допустимо (хотя такое вам едва ли встретится на практике) использовать свое имя для пространства имен, достаточно просто указать его в файле макета:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:bob="http://schemas.android.com/apk/res/android"
  bob:layout_width="match_parent"
  bob:layout_height="wrap_content">
  <TextView
    bob:text="Hello world!"
    bob:layout_width="wrap_content"
    bob:layout_height="wrap_content" />
</LinearLayout>
```

Еще одно важное пространство имен, которое необходимо включить, – пространство имен «auto». Это пространство имен должно быть определено при использовании собственных, а также вспомогательных компонентов. Оно должно ссылаться на <http://schemas.android.com/apk/res-auto>, и обычно ему присваивается имя «app» (и снова, вы можете дать ему любое другое имя, которое следует соглашениям выбора названий для пространств имен в XML):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  ...>
```

С этими двумя пространствами имен в корневом узле вы сможете использовать практически любые возможности, предлагаемые макетами XML.

Создавая свои макеты, вы можете заметить, что для всех представлений Android требуется определить свойства `layout_width` и `layout_height` (которые с учетом использования пространства имен XML будут выглядеть как `android:layout_width = "100dp"`). Этого не требуется при создании представлений программным способом, потому что всегда будет действовать флаг `WRAP_CONTENT`, указывающий, что размеры представления должны подбираться так, чтобы отобразить все содержимое.

К допустимым значениям относятся любые размерные значения (100dp, 100px или 100sp) и предопределенные константы: `LayoutParams.WRAP_CONTENT` и `LayoutParams.MATCH_PARENT`. Действие первой не требует объяснений – если у вас есть `TextView` с текстом «Hello World» и обоим измерениям присвоено значение `WRAP_CONTENT`, компонент `TextView` займет столько места, сколько потребуется для отображения символов, составляющих текст. Константа `MATCH_PARENT` означает, что представление попытается заполнить все доступное пространство в пределах своего родителя вдоль этого измерения.

Итак, окончательный файл макета может выглядеть так:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
```

```

xmlns:app="http://schemas.android.com/apk/res-auto"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical">

<android.support.design.widget.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <android.support.v7.widget.Toolbar
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:navigationIcon="?attr/homeAsUpIndicator" />

</android.support.design.widget.AppBarLayout>

<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1">
</FrameLayout>
</LinearLayout>

```

Небольшое примечание к этому коду: обычно требуется, чтобы один элемент (в данном случае `AppBarLayout`, содержащий панель инструментов) занимал столько места, сколько ему нужно, а другой элемент заполнял остальное пространство (это может быть `LinearLayout` или `ScrollView`). В данном случае `FrameLayout`, служащий контейнером для динамически добавляемых и удаляемых представлений, будет занимать все пространство, доступное внутри его родителя.

Вы можете подумать, что в `FrameLayout` достаточно использовать константу `MATCH_PARENT`, но на самом деле родительский элемент имеет высоту, соответствующую высоте экрана (при условии что он является корневым представлением), поэтому `FrameLayout` окажется обрезанным снизу на величину, равную разности высоты экрана и высоты `Toolbar`. Показанный здесь трюк с `LinearLayout` решает эту распространенную проблему: свойству представления, определяющему переменный размер, нужно присвоить значение `WRAP_CONTENT`, а тому же свойству представления, которое должно заполнить оставшуюся часть пространства в родительском элементе, нужно присвоить значение `0dp` и присвоить свойству `layout_weight` значение 1 (на самом деле можно присвоить любое значение). Весовое свойство сообщит `LinearLayout`, какой процент доступного пространства нужно передать представлению.

Теперь мы имеем макет в файле ресурсов, предварительно скомпилированный и готовый к работе. Как его использовать? Есть несколько способов, часть из которых мы затронули в главе 1, когда рассматривали контроллеры пользовательского интерфейса. А сейчас давайте сосредоточимся на паре общих подходов и рассмотрим еще пару, реже встречающуюся на практике.

Один из способов заключается в назначении макета на роль корневого представления в `Activity`. Просто вызовите `Activity setContentView` и передайте идентификатор ресурса макета, например:

Java

```
public class MyActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_activity);
    }
}
```

Kotlin

```
class MyActivity : Activity() {
    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.my_activity)
    }
}
```

Это все, что нужно! Когда `MyActivity` запустится, вы сразу увидите на экране дерево представлений, описанное в XML-файле макета.

Другой способ заключается в добавлении содержимого макета в существующее дерево представлений. Если дерево представлений уже создано любым другим способом, вы сможете довольно легко добавлять или удалять из него экземпляры `View`, используя методы `ViewGroup`, например `ViewGroup.addView` и `ViewGroup.removeView`. Этот способ прекрасно подходит для представлений, создаваемых программно, но его точно так же можно использовать для явного разворачивания макета. Существует системная служба, которая может помочь вам в этом, и есть два идентичных способа получить ссылку на эту службу:

Java

```
LayoutInflater inflater = (LayoutInflater) context.getSystemService(
    Context.LAYOUT_INFLATER_SERVICE);
```

Kotlin

```
val inflater = context.getSystemService(
    Context.LAYOUT_INFLATER_SERVICE) as LayoutInflater
```

Второй способ:

Java

```
LayoutInflater inflater = LayoutInflater.from(context);
```

Kotlin

```
val inflater = LayoutInflater.from(context)
```

Получив ссылку на экземпляр `inflater`, можно вызвать его метод `inflate`. Первый вариант – добавление представлений, извлеченных из макета, в другое представление `ViewGroup`, которое передается как параметр:

Java

```
inflater.inflate(R.layout.my_activity, someViewGroup, true);
```

Kotlin

```
inflater.inflate(R.layout.my_activity, someViewGroup, true)
```

Метод с этой сигнатурой автоматически добавит развернутое дерево представлений в экземпляр, переданный во втором параметре (*someViewGroup*), и вернет прежний корень (*someViewGroup*).

Второй вариант – вызов метода без корня:

Java

```
inflater.inflate(R.layout.my_activity, null);
```

Kotlin

```
inflater.inflate(R.layout.my_activity, null)
```

Преимущество этой версии в том, что она возвращает развернутое дерево представлений, не добавляя его в существующее дерево представлений.

Кроме того, при использовании фрагментов *Fragment* (также описывались в главе 1) ваши экземпляры *Fragment* могут вернуть экземпляр *View* из своего обратного вызова *onCreateView* (который автоматически вызывается при создании фрагмента). Также классы диспетчеров представлений, такие как *RecyclerView* и *ViewPager*, позволяют переопределить обработчики, которые должны возвращать представление *View* по мере необходимости изменения отображения (при прокрутке или листании).

Вложение представлений друг в друга

Любой экземпляр *ViewGroup* может иметь любое количество дочерних экземпляров *View*. Дочерние элементы отображаются внутри содержащего их экземпляра *ViewGroup*, поэтому, если удалить или скрыть контейнер, дочерние элементы тоже будут удалены или скрыты.

Дочерние представления будут располагаться в родительском контейнере *ViewGroup*, согласно явной логике размещения в *ViewGroup*. Например, *LinearLayout* разместит дочерние элементы по вертикали или горизонтали, с учетом размера каждого дочернего элемента, а *FrameLayout* использует абсолютное позиционирование в пикселях.

Представление *View* можно добавить в *ViewGroup* с помощью любой из перегруженных версий метода *ViewGroup.addView*. Есть версии, принимающие индекс вставляемого элемента и инструкции по размещению. Удалить представление можно с помощью метода *ViewGroup.removeView*. Помните, что класс *ViewGroup* наследует *View*, поэтому экземпляры *ViewGroup* тоже могут добавляться и удаляться друг в друга.

Все эти действия автоматически выполняются при использовании макетов XML. Отношения родитель–потомок узлов XML отображаются в дерево представлений, то есть узел *LinearLayout* с двумя дочерними узлами *TextView* будет развернут в экземпляр *LinearLayout* (потомок *ViewGroup*) с двумя дочерними экземплярами *TextView*. Однако после развертывания макета вы сможете изменить эти отношения с помощью методов *addView* и *removeView*.

Изменение состояния представлений

Класс `View` предлагает несколько методов для изменения свойств, управляющих отображением `View`. Например, `View.setLeft` изменит позицию `View` относительно его контейнера, а `View.setAlpha` изменит прозрачность. На практике очень часто используется метод `setVisibility`, чтобы показать или скрыть представление. Для изменения позиции обычно рекомендуется использовать `setTranslationX` или `setTranslationY`, а не `setLeft`, `setTop`, `setX` или `setY`. Методы `setTranslation*` добавляют указанное «смещение» к нормальной позиции представления. Например, если имеется `LinearLayout` с несколькими строками миниатюр и меток и вы вызвали `setTop` для второй строки, это может повлиять на весь список строк. С другой стороны, если вы хотите на короткое время сдвинуть ее вниз, чтобы открыть некоторый пользовательский интерфейс под ней, то можете безопасно использовать метод `setTranslationY`, вызов которого не затронет общей структуры макета контейнера.

Подклассы `View` представляют собой пеструю компанию и, как правило, предлагают специальные методы. Например, `TextView` предлагает такие методы, как `setText` и `setTextSize`, а `ImageView` имеет метод `setImageBitmap` для изменения отображаемого изображения.

Хотя большинство свойств `View` можно инициализировать в XML, изменять их почти всегда приходится программно. Для получения и присваивания значений свойствам платформа Android использует исключительно методы чтения/записи. Вы всегда должны использовать `myView.setVisibility(View.GONE)`; вместо `myView.visibility = View.GONE`; . Эта проблема имеет довольно долгую историю (и столь же долго длятся споры) в сообществе Java, так же как в отношении применения модификаторов доступа (`private` и `public`). Дело в том, что некоторое время тому назад несколько умных парней собрались и сказали, что никогда (или почти никогда) не следует использовать инструкции, извлекающие или присваивающие значения свойствам напрямую, а вместо этого следует использовать методы чтения и записи, чтобы авторы и пользователи API могли перехватывать эти «события» и добавлять свою логику до или после чтения или изменения состояния программы.

Тем не менее Kotlin допускает использование инструкций прямого присваивания:

```
myView.visibility = View.GONE
```

Но за кулисами эта инструкция вызовет метод записи; в действительности, если в Kotlin вы добавите логику в метод записи, а затем непосредственно присвоите значение свойству, как показано в этом примере, *будет* выполнена логика в методе.

iOS

В iOS под термином «представление» обычно понимается экземпляр `UIView` или его подкласса. Представлением может быть все, что отображается на экране, – метка, изображение, карта, встроенный веб-браузер и многое другое! При

этом все представления, в простейшем их виде, занимают прямоугольную область на экране устройства с определенной позицией.

Базовым представлением, в котором запускаются все приложения iOS, является экземпляр `UIWindow`. Каждый `UIViewController` имеет свойство `view`, содержащее ссылку на экземпляр `UIView`. Окно приложения имеет вложенное представление корневого контроллера. Замена самого верхнего представления с помощью перехода или вызовом `show(_:sender:)` просто меняет местами два `UIView` (управляемых своими контроллерами представления) на экране.

Учитывая важность `UIView` в iOS, давайте посмотрим, как работать с этим классом.

Создание нового представления

Прежде всего можно создать новое представление. В iOS это можно сделать, вызвав конструктор `UIView` и передав ему кадр, где это представление будет отображаться:

```
let aView = UIView(frame: CGRect(x: 10.0, y: 30.0, width: 100.0, height: 50.0))
```

Преыдуший код создаст представление шириной 100pt и высотой 50pt; оно будет размещено на 10pt левее начала координат содержащего его представления и на 30pt ниже. Это кадр представления.

Иногда бывает неизвестно, где на экране должно быть размещено представление при его создании. В таком случае можно создать экземпляр представления без известного размера кадра, просто передав экземпляр `CGRect`, все значения которого равны 0. Этот прием настолько часто используется на практике, что была определена статическая переменная `CGRect`, описывающая кадр с нулевыми размерами. Создать представление с использованием этой переменной можно так:

```
let aView = UIView(frame: .zero)
```

Кадры и границы

Поработав с представлениями `UIView` достаточно долго, вы рано или поздно столкнетесь со свойством `bounds`. Это тоже экземпляр `CGRect`, и оно невероятно похоже на свойство `frame`, но имеет важное отличие: свойство `bounds` описывает прямоугольник, позиция которого выражается относительно *собственной* системы координат, тогда как свойство `frame` представления описывает прямоугольник, позиция которого выражается относительно содержащего его представления (или «суперпредставления»). Например, следующий код выведет такие значения свойств `bounds` и `frame` представления из нашего первого примера:

```
let aView = UIView(frame: CGRect(x: 10.0, y: 30.0, width: 100.0, height: 50.0))
print(aView.bounds) // Выведет: x: 0.0, y: 0.0, width: 100.0, height: 50.0
print(aView.frame) // Выведет: x: 10.0, y: 30.0, width: 100.0, height: 50.0
```

Обратите внимание, что в этом примере ширина (`width`) и высота (`height`) одинаковы в обоих свойствах. Разница в том, что `frame` содержит информацию

о положении представления в координатах суперпредставления, в то время как `bounds` – нет.

Раскадровки и XIB

Теперь мы знаем, как инициализировать представление программным способом. Однако чаще представления создаются за кулисами, после определения в раскадровке или в XML Interface Builder (XIB) внутри Xcode.

Представления в раскадровке определяются в контексте контроллера представления непосредственно внутри Interface Builder. Сцена контроллера представления содержит одно или несколько представлений, вложенных в него и отображаемых вместе. Связывание представления с контроллером представления осуществляется с помощью специального флага компилятора `@IBOutlet`. Этот флаг сообщает, что свойство типа `UIView` внутри класса можно подключить к представлению, содержащемуся в XIB или раскадровке. Например, чтобы создать контроллер представления с синим прямоугольным представлением, сначала нужно создать контроллер представления со свойством, отмеченным флагом `IBOutlet`:

```
class ExampleViewController: UIViewController {
    @IBOutlet var blueRectangle: UIView!
    ...
}
```

Затем связать контроллер с представлением, выполнив следующие действия:

- 1) создать новую сцену контроллера представления;
- 2) изменить класс контроллера представления в инспекторе идентичности (Identity inspector) на `ExampleViewController`;
- 3) добавить представление в главное представление в сцене и изменить его цвет фона на синий в инспекторе атрибутов (Attributes inspector);
- 4) удерживая нажатой клавишу `Ctrl`, ухватить контроллер представления левой кнопкой мыши и перетащить его на синий прямоугольник;
- 5) должно появиться всплывающее окно с перечнем доступных выходов (Outlets) в контроллере представления. Выберите `blueRectangle`, и в Interface Builder появится связь между представлением, отображающим синий прямоугольник, и свойством `blueRectangle` в классе `ExampleViewController`.

В XIB представления определяются аналогично. По сути, XIB можно считать эквивалентом единственной сцены контроллера представления в раскадровке, хотя и с некоторыми допущениями. В действительности описания XIB основаны на формате XML (и очень похожи на раскадровки) и используются для хранения информации о представлениях, чтобы их можно было создавать с помощью графического интерфейса, а не только программно. Они являются предшественниками раскадровок и постепенно выходят из употребления.

Создание представления в XIB происходит так же, как в раскадровке; например, операции по настройке представления выполняются в Interface Builder, но само описание XIB настраивается и создается несколько иначе. Чтобы создать представление в XIB, нужно выполнить следующие шаги:

- 1) создать класс `CustomView`, наследующий `UIView`;
- 2) добавить новое описание XIB (*CustomView.xib*) в проект и в инспекторе идентичности назначить ему только что созданный класс `CustomView`;
- 3) внутри контроллера представления или другого объекта создать экземпляр.

Как создать экземпляр? Для этого нужно получить ссылку на XIB, а затем использовать ее для создания экземпляра, например:

```
let nib = UINib(nibName: "CustomView", bundle: nil)
let view = nib.instantiate(withOwner: nil, options: nil).first as? CustomView
```

К сожалению, `instantiate(withOwner:options:)` создает экземпляр обобщенного класса `UIView`, поэтому вам придется привести его к требуемому подклассу.

Мы узнали, как создавать представления, но истинная ценность пользовательского интерфейса заключается в возможности вложения и объединения представлений. Давайте посмотрим, как вложить одно представление в другое.

Вложение представлений друг в друга

Представления могут создаваться и добавляться в представления, которые сами находятся внутри других представлений, которые, в свою очередь, находятся в третьих представлениях, и т. д. – до самого верхнего `UIView`! Давайте создадим представление и добавим его как дочерний компонент в другое представление.

Следующий код создает родительское и дочернее представления, а затем вызывает `addSubview(_:)`, чтобы добавить дочернее представление в массив вложенных в родителе:

```
let parentView = UIView(frame: .zero)
let childView = UIView(frame: .zero)

parentView.addSubview(childView)
```

Многопоточное окружение

Используете разные потоки выполнения? Будьте осторожны! Операции с представлениями должны выполняться только в основном потоке. Для выполнения действий с представлениями в асинхронных операциях лучше всего использовать Grand Central Dispatch (GCD):

```
DispatchQueue.main.async {
    // Здесь выполняются действия с представлениями
}
```

Более подробную информацию по этой теме вы найдете в главе 8.

Мы добавили представление, а теперь удалим его! Опираясь на пример выше, вот как можно заставить дочернее представление удалить себя из родителя:

```
childView.removeFromSuperview()
```

Ограничения

Рано или поздно у вас неизбежно появится необходимость организовать автоматическое изменение размера представления. Обычно для этих целей используется набор ограничений, определяющих размеры относительно другого представления. Допустим, у нас есть кнопка – особый вид представления, способный принимать события, – и нам нужно поместить в 100pt ниже верхнего края экрана и растянуть по ширине так, чтобы левый и правый края кнопки находились в 16pt от соответствующего края экрана. Реализовать это в коде можно с помощью ограничений, например:

```
class ExampleViewController: UIViewController {
    // Настраивает кнопку после загрузки представления
    override func viewDidLoad() {
        super.viewDidLoad()
        setupButton()
    }

    // Это фактический метод настройки кнопки
    func setupButton() {
        // Создать кнопку
        let button = UIButton(frame: .zero)
        button.translatesAutoresizingMaskIntoConstraints = false

        // Определить цвет фона кнопки
        button.backgroundColor = .blue

        // Добавить кнопку в родительское представление
        view.addSubview(button)

        // Задать расстояние до верхнего края представления
        button.topAnchor.constraint(equalTo: view.topAnchor, constant: 100.0)
            .isActive = true

        // Задать расстояние до левого края представления
        button.leadingAnchor
            .constraint(equalTo: view.leadingAnchor, constant: 16.0)
            .isActive = true

        // Задать расстояние до правого края представления
        button.trailingAnchor
            .constraint(equalTo: view.trailingAnchor, constant: -16.0)
            .isActive = true
    }
}
```

Рассмотрим поближе происходящее здесь. Во-первых, мы определили класс `ExampleViewController` контроллера представления для кнопки; это подкласс `UIViewController`, и, как всякий контроллер представления, он хранит ссылку на свое представление в свойстве `view`. Мы добавили в класс метод `setupButton()`, который вызывается после загрузки представления. Внутри `setupButton()` мы создаем экземпляр кнопки с пустым кадром и присваиваем ее свойству `translatesAutoresizingMaskIntoConstraints` значение `false`.

Глядя на установленные ограничения, можно заметить, что мы ссылаемся на верхний якорь (то есть на верхний край) кнопки и размещаем его в 100pt

от верхнего якоря родительского представления. То же самое мы проделываем с передним (`leadingAnchor`) и задним (`trailingAnchor`) якорями, но устанавливаем расстояние, равное `16.0`.

Если запустить этот код, появится представление, как показано на рис. 2.1.

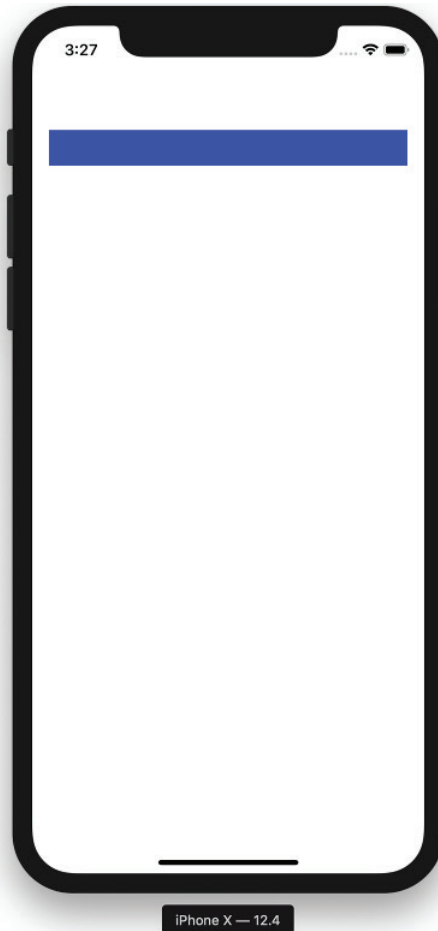


Рис. 2.1 ❖ Кнопка с ограничениями на экране iPhone X

! Создавая представления программным способом и добавляя ограничения, присваивайте свойству `translatesAutoresizingMaskIntoConstraints` значение `false`. Это свойство отключает созданные системой ограничения, управляющие автоматическим изменением размеров, что позволяет нам напрямую указывать свои ограничения. На самом деле желательно всегда присваивать значение `false` этому свойству, когда представление создается программно. Если этого не сделать, иногда могут возникать неприятные и загадочные проблемы с макетом представления.

Главное преимущество ограничений – полное соответствие размеров представления установленным ограничениям. То есть если повернуть устройство, кнопка изменит свои размеры так, что расстояния до левого правого и верхне-

го краев вмещающего представления сохраняются, согласно ограничениям, как показано на рис. 2.2.



Рис. 2.2 ❖ Кнопка с ограничениями, автоматически адаптировавшаяся под ландшафтную ориентацию на экране iPhone X

С помощью Interface Builder

Несмотря на возможность определить ограничения в коде, пользоваться ею становится все сложнее с увеличением числа взаимосвязанных представлений и поддерживаемых размерных классов устройств. Чаще ограничения создаются в Interface Builder. Чтобы создать ту же кнопку, нужно выполнить следующие шаги:

- 1) открыть раскадровку и добавить новую сцену контроллера представления;
- 2) изменить класс контроллера представления в инспекторе идентичности на **ExampleViewController**;
- 3) добавить кнопку в представление, которым управляет этот контроллер;
- 4) щелкнуть на кнопке Add New Constraints (Добавить новые ограничения) внизу справа в редакторе раскадровки;
- 5) настроить отступы в появившемся диалоге. В данном случае отступ сверху следует сделать равным 100pt, а отступы слева и справа – равными 16pt.

Если теперь запустить приложение, оно создаст представление, выглядящее точно так же, как представление, созданное программно.

Автоматическое размещение

Ограничения являются частью технологии Auto Layout автоматического размещения в iOS. Это *невероятно* мощный и сложный инструмент, который при грамотном использовании может уменьшить объем работы по созданию сложных и отзывчивых пользовательских интерфейсов. Сфера применения

Auto Layout слишком велика для этого раздела, поэтому за дополнительной информацией об эффективном использовании этой технологии обращайтесь к документации (<https://oreil.ly/dYJcl>).

Изменение состояния представлений

На одном создании представлений и их добавлении в родительские представления далеко не уедешь. Представления всех видов имеют огромное число свойств, позволяющих сделать их отображение более динамичным и придать им более сложное оформление. Давайте посмотрим, как правильно обновлять некоторые из наиболее часто используемых свойств.

Прозрачность

Изменением значения свойства `alpha` можно управлять прозрачностью представления:

```
myView.alpha = 0.5
```

Эта инструкция сделает представление на 50 % прозрачным. Если вам понадобится полностью скрыть представление (сделать его абсолютно прозрачным), это можно сделать так:

```
myView.alpha = 0.0
```

Скрытие представлений

Присваивание значения `0.0` свойству `alpha` представления не удаляет это представление с экрана. Прозрачное представление все еще будет принимать события касаний и блокировать доступ к другим представлениям, если оно находится над ними. Вот как можно по-настоящему скрыть представление:

```
myView.isHidden = true // Скроет представление
myView.isHidden = false // Отобразит представление
```

Цвет фона

Все представления имеют цвет фона. По умолчанию представления в iOS получают белый фон. Цвет выражается как объект `UIColor`, в данном случае как `UIColor.white`. Вот как можно изменить цвет фона на синий:

```
myView.backgroundColor = UIColor.blue
```

При желании можно установить нестандартный цвет:

```
myView.backgroundColor = UIColor(red: 223.0, green: 23.0, blue: 0.0, alpha: 1.0)
```

Значение `alpha` в `UIColor` влияет только на цвет фона. Это позволяет получить полупрозрачное представление с непрозрачным содержимым, чего нельзя добиться, устанавливая свойство `alpha` самого представления. Иначе говоря:

```
// Сделает цвет фона полупрозрачным
myView.backgroundColor = UIColor(red: 223.0, green: 23.0, blue: 0.0, alpha: 0.5)
```

```
// Сделает все представление полупрозрачным
myView.alpha = 0.5
```

Также важно отметить, что в iOS имеется отдельный прозрачный цвет `clear`. Например, чтобы создать представление без цвета фона, нужно установить цвет фона, как показано ниже:

```
// Сделает фон представления полностью прозрачным
myView.backgroundColor = UIColor.clear
```

Изменение позиции

Требуется изменить позицию представления? Сделать это можно двумя способами, и выбор зависит от использования ограничений. Если ограничения не используются, можно воздействовать на представление через его кадр. Например, вот как можно изменить местоположение представления, воздействуя на кадр:

```
// Создать представление 100×100 в точке (0,0)
let view = UIView(frame: CGRect(x: 0.0, y: 0.0, width: 100.0, height: 100.0))

// Уменьшить размер представления до 50×50 и переместить в точку (130, 55)
view.frame = CGRect(x: 130.0, y: 55.0, width: 50.0, height: 50.0)
```

Чтобы изменить местоположение представления, использующего ограничения, нужно напрямую скорректировать ограничения. Это можно сделать программно, создав ограничение в Interface Builder и присвоив его свойству `@IBOutlet`, созданному в классе для редактирования значения ограничения. Например, если в Interface Builder ограничение ширины определено как `100.0`, его можно заменить значением `50.0`, как показано ниже:

```
class ExampleViewController: UIViewController {
    @IBOutlet var widthConstraint: NSLayoutConstraint!
    ...
    func resizeWidth() {
        widthConstraint.constant = 50.0
    }
}
```

Можно пойти другим путем, чтобы не изменять значений в коде, и добавить два ограничения непосредственно в объект в Interface Builder, отключив одно из них. Затем, когда придет время, ограничения можно переключить:

```
class ExampleViewController: UIViewController {
    @IBOutlet var widthConstraint: NSLayoutConstraint!
    @IBOutlet var otherWidthConstraint: NSLayoutConstraint!
    /// ...
    func resizeWidth() {
        widthConstraint.isActive = false
        otherWidthConstraint.isActive = true
    }
}
```

Это наиболее широко используемые способы корректировки ограничений, но, честно говоря, изменить ограничения, связанные с объектом, можно множеством разных способов. В этом сила и сложность Auto Layout.

Другие свойства

Выше мы коснулись лишь нескольких из множества свойств, доступных в представлениях. Некоторые представления, такие как UILabel, имеют свойства, определяющие характеристики шрифта. Представления типа UIImageView могут иметь связанные с ними UIImage. Представления, такие как WKWebView (веб-представление, встроенное в iOS), имеют еще более сложные наборы свойств, позволяющих включать и переключать их. Как всегда, для выяснения подробностей о доступных свойствах в определенном классе лучше обращаться к онлайн-документации на сайте Apple.

Простейшая анимация

Еще один аспект, который мы не обсудили в этой главе, – это слои Core Animation. Этот механизм предлагает совершенно другой, дополнительный способ изменения внешнего вида представлений. Такие компоненты, как borderRadius и mask, позволяют создавать сложные, анимированные пользовательские интерфейсы.

SwiftUI

На смену UIKit пришла SwiftUI – новая библиотека для создания пользовательских интерфейсов в Swift. Она поддерживает многообещающий декларативный синтаксис для описания пользовательских интерфейсов следующего поколения. В настоящее время в реализации SwiftUI еще имеются некоторые шероховатости, но, учитывая инвестиции Apple в эту технологию, можно с уверенностью сказать, что она станет более актуальным и – если Apple продолжит развивать это направление – преобладающим способом создания приложений через несколько коротких лет.

Что мы узнали

Android и iOS имеют очень разные механизмы отображения представлений на экране устройства, но в целом они преследуют одну и ту же цель: отображение контента для взаимодействия с пользователем. В этой главе мы рассмотрели следующие вопросы:

- как на обеих платформах создать новое представление для отображения на экране;
- различия в создании представлений в макетах XML в Android и раскладках в iOS, которые оба имеют формат XML и хранятся в файлах;
- как с помощью ограничений обеспечить единообразие внешнего вида представлений в iOS на устройствах с экранами разных размеров;
- вложение представлений друг в друга и возможности их изменения.

Эти сведения помогут вам лучше понять особенности создания и использования нестандартных представлений, о которых рассказывается в следующей главе. Поехали!

Глава 3

Пользовательские компоненты

Обе системы, Android и iOS, предлагают большое разнообразие готовых виджетов и компонентов, но иногда возникает желание или необходимость создать свой, нестандартный компонент. Например, может понадобиться создать виджет выбора даты или цвета с настраиваемым внешним видом или поведением, переключатель или выключатель со встроенной подписью, компоненты визуализации данных, такие как диаграммы и графики, или что-то более простое, например настраиваемую метку или значок, придающие характерный вид вашему приложению.

Что бы вам ни потребовалось, у вас есть все возможности создать желаемый компонент в любой из обсуждаемых в книге систем, но процесс создания в них очень отличается и может иметь тонкости, выглядящие порой весьма загадочно.

Задачи

В этой главе вы узнаете:

- 1) как создать свое представление;
- 2) как использовать свое представление.

ANDROID

Большинство макетов в приложениях для Android создаются с использованием данных в формате XML, поэтому вам наверняка захочется придать своему компоненту возможность принимать и соответствующим образом реагировать на произвольные свойства. Например, если вы решите создать свою палитру выбора цвета, вам, вероятно, следует предложить в ней цвет по умолчанию или даже определенное цветовое пространство, такое как HSL или RGB.

Поскольку ресурсы XML компилируются, вам придется убедиться, что система распознает дополнительные свойства, которые вы решите добавить в свой компонент, и определить, какие значения являются действительными. Напри-

мер, если вы добавите атрибут `color`, он не должен принимать любое другое значение, кроме значения цвета или, может быть, идентификатора цвета. Если пользователь попытается ввести значение в неверном формате, программа не сможет скомпилироваться, и Android Studio уведомит его, представив информативное сообщение об ошибке.

Как создать свое представление

Типичным подходом к созданию своего представления является определение класса, наследующего класс `View`, `ViewGroup` или один из существующих их подклассов, и добавление в него своих функциональных возможностей. Возможности пользовательского представления несколько ограничены – он может изменять уже нарисованное (текст, цвета, фигуры) или то, что ему подчинено. Первое можно реализовать с помощью метода `onDraw` класса `View`. Этот метод принимает один параметр – предварительно заполненный экземпляр `Canvas`, имеющий те же размеры, что и само представление. Более подробно о доступных возможностях объекта `Canvas` можно узнать в документации для разработчика, но если говорить кратко, здесь можно делать все, что угодно. Вы можете вызывать методы для рисования, вывода текста и отображения геометрических фигур. Можно также использовать геометрические классы, такие как `Rect` или `Path`, для отображения более сложных структур, и объекты `Paint` для настройки цвета, заливки или текстуры.

Например, следующий простой подкласс `View` рисует красный круг (точнее, овал), вписанный в границы представления:

Java

```
public class Oval extends View {
    private Paint mPaint = new Paint();
    {
        mPaint.setColor(Color.RED);
        mPaint.setStyle(Paint.Style.FILL);
    }
    public Oval(Context context) {
        super(context);
    }
    public Oval(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    public Oval(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }
    @Override
    protected void onDraw(Canvas canvas) {
        canvas.drawOval(0, 0, getWidth(), getHeight(), mPaint);
    }
}
```

Kotlin

```
class Oval @JvmOverloads constructor(context: Context, attrs: AttributeSet? = null,
    defStyleAttr: Int = 0, defStyleRes: Int = 0) : View(context, attrs, defStyleAttr,
    defStyleRes) {

    private val paint = Paint()

    init {
        paint.color = Color.RED
        paint.style = Paint.Style.FILL
    }

    override fun onDraw(canvas: Canvas) {
        canvas.drawOval(0f, 0f, width.toFloat(), height.toFloat(), paint)
    }
}
```

Как видите, есть возможность определить свои методы для настройки цвета или добавить свое свойство `Paint` для вывода растрового изображения, создания теней или градиентов.

Аналогично можно определить подкласс `TextView`, который всегда выводит рамку вдоль нижней границы, например для использования в `LinearLayout` или `RecyclerView`:

Java

```
public class BottomBorderTextView extends TextView {

    private Paint mPaint = new Paint();
    {
        mPaint.setStyle(Paint.Style.STROKE);
        mPaint.setColor(Color.BLACK);
    }

    public BottomBorderTextView(Context context) {
        super(context);
    }

    public BottomBorderTextView(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
    }

    public BottomBorderTextView(Context context, @Nullable AttributeSet attrs,
        int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        // не забывайте вызывать super.onDraw для вывода текста, фона, смежных
        // элементов и пр.
        super.onDraw(canvas);
        canvas.drawLine(0, getHeight(), getWidth(), getHeight(), mPaint);
    }
}
```

Kotlin

```

class BottomBorderTextView @JvmOverloads constructor(context: Context,
    attrs: AttributeSet? = null,
    defStyleAttr: Int = 0, defStyleAttrRes: Int = 0) :
    TextView(context, attrs, defStyleAttr, defStyleAttrRes) {

    private val paint = Paint()

    init {
        paint.style = Paint.Style.FILL
        paint.color = Color.RED
    }

    override fun onDraw(canvas: Canvas) {
        // не забывайте вызывать super.onDraw для вывода текста, фона, смежных
        // элементов и пр.
        super.onDraw(canvas)
        canvas.drawLine(0, height, width, height, paint)
    }
}

```

Также можно добавить новый метод для изменения цвета рамки или ее перерисовывания:

Java

```

public void setBorderColor(int color) {
    mPaint.setColor(color);
    // вызов invalidate уведомляет систему о необходимости перерисовать
    // представление при отображении следующего кадра
    invalidate();
}

```

Kotlin

```

fun setBorderColor(color: Int) {
    paint.color = color
    // вызов invalidate уведомляет систему о необходимости перерисовать
    // представление при отображении следующего кадра
    invalidate()
}

```

С другой стороны, пользовательская версия `ViewGroup` может реализовать новую стратегию размещения или включать набор дочерних экземпляров `View` и `ViewGroup`, образующих сложный компонент, такой как средство выбора даты или медиапроигрыватель. Подобный подход к использованию `ViewGroup` обычно называют созданием пользовательского «компонента», он требует более глубокого погружения. Мы затронем эту тему далее, но вы обязательно должны ознакомиться с дополнительными подробностями в документации (<https://oreil.ly/ugTKF>).

Суть этого подхода сводится к переопределению двух методов: `onMeasure` и `onLayout`. `onMeasure` сообщает родителю, сколько места *требуется* пользовательский компонент. Иногда это все доступное пространство; иногда ровно столько, сколько требуется для отображения содержимого.

`onMeasure` ожидает два параметра типа `int`: `widthMeasureSpec` и `heightMeasureSpec`. Эти значения включают биты, определяющие «режим» (например, флаг `MATCH_PARENT`, указывающий, что компоненту должно быть отведено все пространство его родителя), а также размеры в пикселях.

Для чтения этих значений из параметров можно использовать методы `MeasureSpec.getMode` и `MeasureSpec.getSize`.

Второй метод – `onLayout`. Существуют разнообразные события, которые вызывают повторное размещение элементов в дереве представлений, например изменение размера родительского или дочернего элемента, добавление/удаление дочернего элемента или переупорядочение дочерних элементов. Кроме того, есть возможность явно потребовать выполнить повторное размещение элементов вызовом `View.requestLayout`.

Реализация по умолчанию метода `onLayout` ничего не делает (хотя конкретные подклассы `ViewGroup`, такие как `FrameLayout` и `LinearLayout`, переопределяют этот метод). У вас есть возможность выбирать, как `ViewGroup` будет размещать свои дочерние элементы. Например, вертикально ориентированный `LinearLayout` сначала измерит все дочерние элементы (в `onMeasure`); затем, в `onLayout`, разместит первый дочерний элемент вверху, второй дочерний элемент – под ним, третий – под вторым и т. д. Вертикальный размер этого компонента будет равен сумме вертикальных размеров его дочерних элементов. `FrameLayout` действует проще (и эффективнее): все дочерние элементы размещаются независимо друг от друга, причем для каждого дочернего элемента явно определяется параметр `LayoutParams` со значениями координат левого верхнего угла элемента.

Например, следующий код использует `FrameLayout` для отображения содержимого в `Activity` и располагает `TextView` на расстоянии ста пикселей от верхнего и левого края. Появление других дочерних экземпляров `View` в этом контейнере не повлияет на положение `TextView`, потому что при выполнении операции `onLayout` класс `FrameLayout` проверяет только значения координат дочерних элементов в `LayoutParams`:

Java

```
public class MyActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        FrameLayout container = new FrameLayout(this);
        setContentView(container);
        FrameLayout.LayoutParams lp = new FrameLayout.LayoutParams(
            LayoutParams.WRAP_CONTENT,
            LayoutParams.WRAP_CONTENT);

        lp.leftMargin = 100;
        lp.topMargin = 100;
        TextView textView = new TextView(this);
        textView.setText("Hello world!");
        container.addView(textView, lp);
    }
}
```

Kotlin

```
class MyActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        val container = FrameLayout(this)
        setContentView(container)
        val lp = FrameLayout.LayoutParams(FrameLayout.LayoutParams.WRAP_CONTENT,
                                          FrameLayout.LayoutParams.WRAP_CONTENT)

        lp.leftMargin = 100
        lp.topMargin = 100
        val textView = TextView(this)
        textView.text = "Hello world!"
        container.addView(textView, lp)
    }
}
```

Вот пример реализации `onLayout`, которая располагает дочерние элементы по горизонтали, пока не останется свободного места, после чего переходит к следующей строке. Иногда такое размещение называют «потокowym» («FlowLayout»):

Java

```
@Override
protected void onLayout(boolean changed, int left, int top,
                        int right, int bottom) {
    int x = 0;
    int y = 0;
    int tallest = 0;
    for (int i = 0; i < getChildCount(); i++) {
        View child = getChildAt(i);
        int childWidth = child.getMeasuredWidth();
        int childHeight = child.getMeasuredHeight();
        tallest = Math.max(tallest, childHeight);
        if (childWidth + x > getWidth()) {
            x = 0;
            y += tallest;
            tallest = 0;
        }
        child.layout(x, y, x + childWidth, y + childHeight);
        x += childWidth;
    }
}
```

Kotlin

```
override fun onLayout(changed: Boolean, left: Int, top: Int,
                    right: Int, bottom: Int) {
    var x = 0
    var y = 0
    var tallest = 0
    for (i in 0 until childCount) {
        val child = getChildAt(i)
        val childWidth = child.measuredWidth
```

```

    val childHeight = child.measuredHeight
    tallest = Math.max(tallest, childHeight)
    if (childWidth + x > width) {
        x = 0
        y += tallest
        tallest = 0
    }
    child.layout(x, y, x + childWidth, y + childHeight)
    x += childWidth
}
}

```

Также можно добавлять свои методы и свойства, чтобы получить функциональность, необходимую вашему компоненту.

Как использовать свое представление

Выше мы узнали довольно много о пользовательских представлениях и компонентах, но наше описание по большей части было упорядочено и достаточно логично. Оно освещает основную механику, характерную для фреймворка, и может показаться не особенно ценным. Чтобы приложение могло творить свое волшебство, нам нужно перепрыгнуть через несколько ступенек – не расстраивайтесь, если что-то останется для вас непонятным; мы сами иногда сталкиваемся с малопонятными проявлениями и ищем ответы на возникающие в связи с этим вопросы.

Для начала определим основу для последующего обсуждения. У нас есть подкласс `TextView` из предыдущего примера: `BottomBorderTextView`. Он будет представлен в XML, как любой другой компонент `TextView`, и для него будут определены свойства из пространства имен `android`, такие как `android:text` и `android:textSize`. Но нам также нужно добавить новое свойство: `borderColor`.

Этот новый атрибут не является частью пространства имен `android` – он относится к пространству имен XML нашего приложения (<http://schemas.android.com/apk/res/com.yourapp>). В предыдущей главе, посвященной представлениям, мы видели, что можно использовать «автоматическое» пространство имен <http://schemas.android.com/apk/res/auto> и привязать его непосредственно к пространству имен текущего приложения.

Сначала определим свой атрибут как компилируемое значение `values`. Традиционно такие определения атрибутов помещаются в файл `res/values/attrs.xml`, но вообще его можно поместить в любой подкаталог внутри `res/values`.

Определение должно находиться в узле `resources` в виде узла `declare-styleable` с атрибутом `name`, содержащим имя пользовательского компонента. Каждое свойство (в данном примере мы определяем только одно свойство `borderColor`) должно быть представлено узлом `attr` с атрибутом `name`, содержащим имя свойства, и атрибутом `format`, представляющим приемлемые типы данных (`color`, `boolean`, `dimension`, `integer` и др.).

Вот как выглядит такое определение:

```

<resources>
  <declare-styleable name="BottomBorderTextView">

```

```

    <attr name="borderColor" format="color" />
</declare-styleable>
</resources>

```

Создав определение пользовательского свойства, на него можно сослаться в макете XML своего компонента:

```

<com.myapp.BottomBorderTextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Hello world!"
    app:borderColor="#FFFF9900" />

```

Затем нужно получить значение этого атрибута во время создания представления. Для этого приходится использовать малопонятный код. Взгляните на следующий пример, который поможет вам понять то, что сложно объяснить на простом языке:

Java

```

public BottomBorderTextView(Context context,
                            @Nullable AttributeSet attrs,
                            int defStyleAttr) {

    super(context, attrs, defStyleAttr);
    TypedArray a = context.getTheme().obtainStyledAttributes(
        attrs,
        R.styleable.BottomBorderTextView,
        0, 0);
    try {
        int color = a.getBoolean(R.styleable.BottomBorderTextView_borderColor,
            Color.BLACK);
        setBorderColor(color);
    } finally {
        a.recycle();
    }
}

```

Kotlin

```

constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int) :
    super(context, attrs, defStyleAttr) {
    val a = context.theme.obtainStyledAttributes(attrs,
        R.styleable.BottomBorderTextView, 0, 0)
    try {
        borderColor = a.getInt(R.styleable.BottomBorderTextView_borderColor, Color.BLACK)
    } finally {
        a.recycle()
    }
}
}

```

Откуда взялось это: `R.styleable.BottomBorderTextView`? А это: `R.styleable.BottomBorderTextView_borderColor`? Ответ прост: магия. Система творит волшебство

за кулисами, но вы можете быть уверены, что, создав ресурс XML, как было показано выше, вы получите эти значения в глобальном экземпляре R. Узел `declare-styleable` генерирует имя из своего атрибута `name` и затем добавляет в конец символ подчеркивания и значение атрибута `name` вложенного узла `attr`. Тип `TypedArray` тоже немножко волшебный, и мы очень сомневаемся, что вам когда-либо доведется использовать `context.getTheme().obtainStyledAttributes` в других ситуациях. Как обычно, мы рекомендуем обратиться к документации для разработчиков, а также заглянуть в исходный код, но в данном случае (как это характерно для операций компиляции на уровне платформы) мы призываем вас просто довериться нам.

❗ Возможно, вы захотите завернуть эту пользовательскую логику обработки атрибута в метод `initialize` и вызывать его в каждом конструкторе своего представления, или можно использовать удобный трюк, вызывая каждый конструктор уровнем выше со значениями по умолчанию или `null`.

iOS

Настроить свое представление в iOS и добавить его в сцену можно в раскладовке или непосредственно в файле XIB в XCode. К сожалению, подобный подход быстро становится неустойчивым из-за сложности повторного использования такого представления. Как правило, под «пользовательским представлением» в iOS подразумевается пользовательский класс, который наследует стандартный класс представления из iOS. Такие представления часто можно использовать повторно, и они, как правило, обладают более широкими функциональными возможностями, чем при соблюдении принципа единственной ответственности и непосредственном добавлении этих возможностей в контроллер представления. Давайте поближе познакомимся с некоторыми приемами создания пользовательских представлений в iOS и с использованием UIKit.

Как создать свое представление

Представлением в iOS является экземпляр `UIView`. Это может быть экземпляр самого класса `UIView` или его подкласса. Создать свое представление не намного сложнее, чем создать подкласс `UIView`:

```
class SomeView: UIView {
    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }

    override init(frame: CGRect) {
        super.init(frame: frame)
    }

    // Далее следуют нестандартные реализации методов
}
```

В классе должны быть определены два метода инициализации, если вы планируете добавить больше свойств и хотите, чтобы объект был готов к работе

сразу после его инициализации: `init(coder:)` и `init(frame:)`. В этих методах следует реализовать настройки, необходимые для подготовки объекта к работе.

Если, например, требуется создать представление с красным фоном и кнопкой с надписью «Click Me!», реализовать его настройку можно примерно так:

```
class SomeView: UIView {
    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        setupView()
    }

    override init(frame: CGRect) {
        super.init(frame: frame)
        setupView()
    }

    private func setupView() {
        backgroundColor = .red
        let button = UIButton(type: .custom)
        button.titleLabel?.text = "Click Me!"

        addSubview(button)
    }
}
```

Метод `setupView` совместно используется обоими методами инициализации и вызывается при создании каждого экземпляра представления. Такой подход позволяет обеспечить согласованность процесса настройки. Внутри `setupView` свойству `backgroundColor` присваивается ссылка на экземпляр `.red` класса `UIColor`, а затем создается новый экземпляр кнопки и добавляется как подпредставление.

Также в свое представление можно добавлять дополнительные свойства. Например, давайте сделаем текст на кнопке настраиваемым:

```
class SomeView: UIView {
    var buttonText: String = "Click Me!" {
        didSet {
            button.titleLabel?.text = self.buttonText
        }
    }

    lazy var button: UIButton = {
        let button = UIButton(type: .custom)
        addSubview(button)
        return button
    }()

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

    override init(frame: CGRect) {
        super.init(frame: frame)
        setupView()
    }
}
```

```

convenience init(frame: CGRect, buttonText: String) {
    self.init(frame: frame)
    self.buttonText = buttonText
}

private func setupView() {
    backgroundColor = .red
    button.titleLabel?.text = buttonText
}
}

let noClicky = SomeView(frame: CGRect.zero, buttonText: "Don't click me!")

```

Здесь мы добавили новое свойство `buttonText` для хранения текста надписи на кнопке. Это позволяет инициализировать кнопку и одновременно передавать текст надписи. Также мы немного изменили метод `setupView`: мы убрали из него инициализацию кнопки и перенесли ее в свойство `lazy`, чтобы позже его можно было изменить, не создавая новую кнопку.

Теперь посмотрим, как можно использовать наше новое представление.

Как использовать свое представление

Чтобы использовать свое представление, не требуется больших усилий:

```

class SomeViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let clickMeButton =
            SomeView(frame: CGRect(x: 0.0, y: 0.0, width: 100.0, height: 50.0),
                    buttonText: "Click Me!")
        view.addSubview(clickMeButton)
    }
}

```

Этот код создает новый контроллер, новый экземпляр представления и добавляет его в контроллер. Как видите, все просто!

С другой стороны, добавить экземпляр кнопки можно с помощью `Interface Builder`. В раскладку или файл XIB добавьте новый объект представления из библиотеки. Затем в инспекторе идентичности выберите для объекта свой класс представления **SomeView**.

Уже неплохо, но можно сделать еще лучше, используя флаги `@IBInspectable` и `@IBDesignable`, относящиеся к `Interface Builder`. Эти флаги позволяют `Interface Builder` настраивать и отображать представление так, будто оно отображается в работающем приложении.

Для этого нужно декорировать класс представления этими флагами:

```

@IBDesignable class SomeView: UIView {
    @IBInspectable var buttonText: String = "Click Me!" {
        didSet {
            button.titleLabel?.text = self.buttonText
        }
    }

    lazy var button: UIButton = {

```

```

        let button = UIButton(type: .custom)
        addSubview(button)
        return button
    }()

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

    override init(frame: CGRect) {
        super.init(frame: frame)
        setupView()
    }

    convenience init(frame: CGRect, buttonText: String) {
        self.init(frame: frame)
        self.buttonText = buttonText
    }

    private func setupView() {
        backgroundColor = .red
        button.titleLabel?.text = buttonText
    }
}

```

Теперь, вернувшись в раскладовку или представление, вы увидите в инспекторе атрибутов новое поле **Button Text**, в которое можно вписать свой текст для отображения на кнопке. Если изменить его, Interface Builder обновит текст кнопки, отображаемый на экране (см. рис. 3.1)!

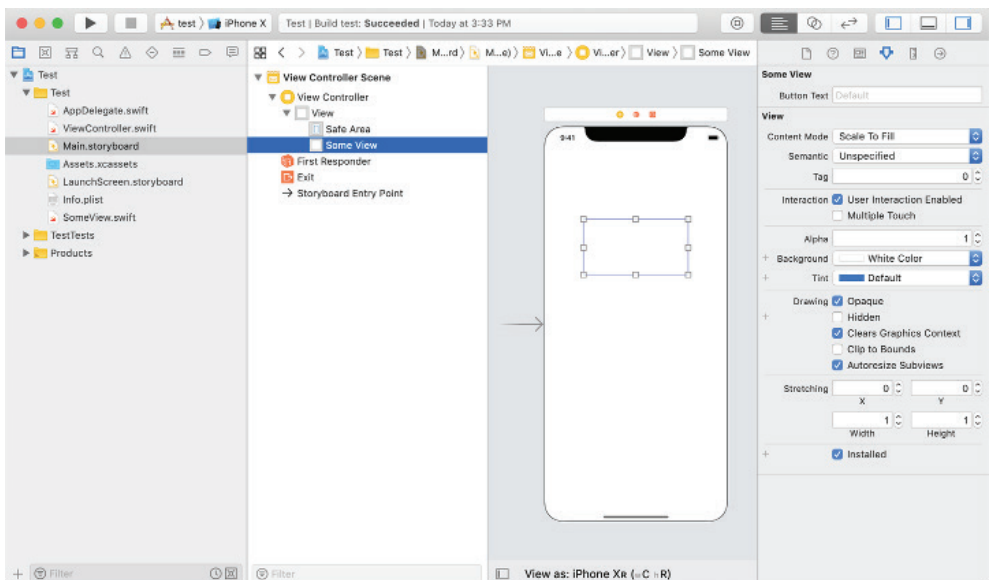


Рис. 3.1 ❖ Пользовательское представление с редактируемыми полями в Interface Builder

Interface Builder, UIKit и Xcode – это мощная комбинация инструментов. С их помощью можно создавать любые представления и их вариации – единственным ограничением является ваше воображение. Возможности настолько обширны, что их не получится перечислить в единственной главе книги. За дополнительной информацией мы, как обычно, советуем обратиться к документации для разработчиков UIView (<https://oreil.ly/QtOVC>).

Что мы узнали

Эта глава завершает описание контроллеров пользовательского интерфейса (глава 1), представлений (глава 2) и пользовательских представлений. В этой главе мы узнали много нового:

- в обеих системах, Android и iOS, для создания своего представления требуется определить и настроить подкласс стандартного представления;
- процедуры определения, создания и настройки представлений имеют некоторые отличия;
- мы обсудили подходы и способы использования пользовательских представлений для создания пользовательских интерфейсов;
- обе платформы обладают обширными и мощными наборами инструментов для создания интерфейсов, которые могут радовать и удивлять пользователей.

Глава 4

Пользовательский ввод

В течение десятилетий основным средством взаимодействия с компьютерами были клавиатура и мышь. Пользователь был буквально привязан к устройству при работе с ним. Единственный способ выполнить работу – сесть за рабочую станцию и начать работу. Спустя какое-то время появились ноутбуки, обеспечившие некоторую мобильность, но они использовали практически те же механизмы ввода.

Затем появились сенсорные экраны.

Современные устройства на Android и iOS больше не находятся на расстоянии вытянутой руки от пользователя. Они находятся в тесном физическом контакте с пользователями. Когда пользователь нажимает кнопку, он касается ее непосредственно, а не трекпада или клавиатуры. Это делает ввод одним из наиболее важных аспектов превращения любого старого приложения в динамическое произведение искусства.

Ввод может принимать разные виды и формы: касание ссылки в веб-представлении, ввод пароля в форме входа в систему или перелистывание экранов в поисках лиц, вызывающих симпатию, с кем могли бы начаться новые отношения, которые потом могут даже перерасти в любовь. Ставки очень высоки, и поэтому платформы готовы поддержать вас надежными наборами инструментов, помогающими получить ввод пользователя и превратить его в действие, результат которого он сможет увидеть, услышать или коснуться.

Задачи

В этой главе вы узнаете:

- 1) как получить событие касания и реализовать реакцию на него;
- 2) как получить событие нажатия клавиши на клавиатуре и реализовать реакцию на него;
- 3) как обрабатывать сложные жесты.

ANDROID

Поддержка жестов в Android имеет немного сложный API, но она достаточно прозрачна, и, как разработчик, вы будете иметь всю информацию, необходимую для удовлетворения даже самых требовательных приложений, предполагающих большое число касаний.

Получение события касания и реакция на него

Касание – это, пожалуй, самая распространенная форма пользовательского ввода в большинстве современных мобильных приложений. Это может быть касание кнопки для отправки формы, касание поля ввода текста для передачи ему фокуса ввода, долгое касание с целью открыть контекстные параметры или двойное касание для увеличения/уменьшения масштаба карты. Все эти события являются интуитивно понятными выражениями желаний пользователя.

Поэтому неудивительно, что Android делает захват сигналов касаний простым и доступным.

☑ По историческим причинам в платформе Android все еще используется термин «щелчок». В большинстве окружений с сенсорным экраном под «щелчком» подразумевается «касание».

Все экземпляры `View` (включая `ViewGroup`) имеют настраиваемое свойство `View.OnClickListener` (через `setOnClickListener`). После его настройки система возьмет на себя все сложности, связанные с организацией получения и доставки события касания, и всякий раз, распознав жест, будет вызывать метод `onClick` слушателя событий. Чтобы удалить реакцию на касание из данного представления, достаточно записать в это свойство значение `null`, вызвав `myView.setOnClickListener(null)`;

Обратите внимание, что `View.OnClickListener` – это простой функциональный интерфейс с единственным методом `onClick(View view)`. Вот его определение, буквально скопированное из исходного кода, имевшегося в момент написания этой книги:

```
public interface OnClickListener {
    void onClick(View v);
}
```

Этот интерфейс может быть реализован практически на любом уровне – в контроллере `Activity` или `Fragment`, в самом экземпляре `View`, в анонимном классе или в виде лямбда-выражения или ссылки на метод. Кроме того, обработчики событий касания можно назначать в макетах XML. Далее мы рассмотрим все эти подходы по очереди.

Реализация `View.OnClickListener` в контроллере:

Java

```
public class MyActivity extends Activity implements View.OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Button button = new Button(this);
        button.setText("Click me!");
        button.setOnClickListener(this);
        setContentView(button);
    }
    @Override
    public void onClick(View view) {
```

```

        Log.d("MyTag", "View was clicked " + view.toString());
    }
}

```

Kotlin

```

class MyActivity : Activity(), View.OnClickListener {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val button = Button(this)
        button.text = "Click me!"
        button.setOnClickListener(this)
        setContentView(button)
    }

    override fun onClick(view: View) {
        Log.d("MyTag", "View was clicked $view")
    }
}

```

В ВИДЕ ССЫЛКИ НА МЕТОД:

Java

```

public class MyActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Button button = new Button(this);
        button.setText("Click me!");
        button.setOnClickListener(this::myClickMethod);
        setContentView(button);
    }

    public void myClickMethod(View view) {
        Log.d("MyTag", "View was clicked " + view.toString());
    }
}

```

Kotlin

```

class MyActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val button = Button(this)
        button.text = "Click me!"
        button.setOnClickListener(::onClick)
        setContentView(button)
    }

    fun onClick(view: View) {
        Log.d("MyTag", "View was clicked $view")
    }
}

```

В виде лямбда-выражения:

Java

```
button.setOnClickListener(view -> Log.d("MyTag", "clicked!"));
```

Kotlin

```
button.setOnClickListener { Log.d("MyTag", "clicked!") }
```

В анонимном классе:

Java

```
button.setOnClickListener(new View.OnClickListener() {  
    private boolean mToggled;  
  
    @Override  
    public void onClick(View view) {  
        mToggled = !mToggled;  
        Log.d("MyTag", "toggled? " + mToggled);  
    }  
});
```

Kotlin

```
button.setOnClickListener(object:View.OnClickListener() {  
    private var toggled:Boolean = false  
  
    override fun onClick(view:View) {  
        toggled = !toggled  
        Log.d("MyTag", "toggled? $toggled")  
    }  
})
```

В подклассе View, который всегда будет одинаково реагировать на события касания:

Java

```
public class MyButton extends Button implements View.OnClickListener {  
  
    public MyButton(Context context) {  
        this(context, null);  
    }  
  
    public MyButton(Context context, @Nullable AttributeSet attrs) {  
        this(context, attrs, 0);  
    }  
  
    public MyButton(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {  
        super(context, attrs, defStyleAttr);  
        setOnClickListener(this);  
    }  
  
    @Override  
    public void onClick(View view) {  
        mToggled = !mToggled;  
        Log.d("MyTag", "toggled? " + mToggled);  
    }  
}
```

Kotlin

```
class MyButton @JvmOverloads constructor(context: Context, attrs: AttributeSet? = null,
    defStyleAttr: Int = 0) : Button(context, attrs, defStyleAttr), View.OnClickListener {
    var toggled = false

    init {
        setOnClickListener(this)
    }

    override fun onClick(view: View) {
        toggled = !toggled
        Log.d("MyTag", "toggled? $toggled")
    }
}
```

Наконец, можно указать имя метода (как строку) в макете XML. Соответствующий контроллер Activity должен иметь общедоступный метод с этим именем и с сигнатурой, соответствующей сигнатуре метода View.OnClickListener.onClick:

```
<!-- содержимое файла res/layout/myactivity_layout.xml -->
<Button xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click me!"
    android:onClick="myClickHandler" />
```

Обратите внимание, что Activity автоматически определит связь и создаст логику привязки слушателя, не требуя явных ссылок на метод или на View:

```
public class MyActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.myactivity_layout);
    }

    public void myClickHandler(View view) {
        Log.d("MyTag", "View was clicked " + view.toString());
    }
}
```

Обратите внимание, что в каждый конкретный момент времени представление View может иметь не более одного слушателя OnClickListener. Чтобы подключить несколько обработчиков событий касания, нужно либо изменить основного слушателя, чтобы он вызывал других слушателей, либо создать небольшую инфраструктуру для поддержки нескольких слушателей. Например, вот как можно организовать управление списком обработчиков в одном слушателе:

Java

```
public class CompositeClickListener extends ArrayList<View.OnClickListener>
    implements View.OnClickListener {
```

```

@Override
public void onClick(View view) {
    for (View.OnClickListener listener : this) {
        listener.onClick(view);
    }
}
}

```

Kotlin

```

class CompositeClickListener : ArrayList<View.OnClickListener>(), View.OnClickListener {
    override fun onClick(view: View) {
        for (listener in this) {
            listener.onClick(view)
        }
    }
}

```

И использовать его:

Java

```

CompositeClickListener listener = new CompositeClickListener();
listener.add(view -> Log.d("MyTag", "hello!"));
listener.add(view -> Log.d("MyTag", "hola!"));
listener.add(view -> Log.d("MyTag", "bonjour!"));
myView.setOnClickListener(listener);

```

Kotlin

```

val listener = CompositeClickListener()
listener.add(View.OnClickListener{ Log.d("MyTag", "hello!") })
listener.add(View.OnClickListener{ Log.d("MyTag", "hola!") })
listener.add(View.OnClickListener{ Log.d("MyTag", "bonjour!") })
myView.setOnClickListener(listener)

```

Спектр доступных вариантов может показаться очень широким, но на самом деле это только верхушка айсберга. Платформа Android предоставляет доступ к событиям касания на нескольких уровнях, где можно реализовать свою логику обработки касания, например запускать обработку касания только после некоторой задержки, или более либерально (или консервативно) ограничить область «блуждания» (насколько далеко от представления может произойти событие касания, чтобы его можно было считать касанием). К счастью, необходимость управления жестами возникает в очень редких случаях, тем не менее мы поговорим об этом далее в этой главе.

Получение события ввода с клавиатуры и реакция на него

Фреймворк Android обрабатывает события от клавиатуры совершенно иначе, чем другие фреймворки пользовательского интерфейса, с которыми вы могли иметь дело. Даже класс `KeyEvent`, с которым вы, наверное, ожидаете иметь дело, очень редко доступен напрямую разработчику. Обратите внимание, что и текущая документация гласит:

Программные методы ввода могут использовать несколько разных способов ввода текста, поэтому нет никакой гарантии, что любое нажатие клавиши на программной клавиатуре сгенерирует событие нажатия клавиши: этот аспект остается на усмотрение IME (метода ввода), и фактически отправка таких событий не рекомендуется. Вы не должны полагаться на получение событий `KeyEvent` для любых клавиш при использовании программного метода ввода.

Здесь просто утверждается, что генерирование событий от «программной» (экранной) клавиатуры не гарантируется. Эти события гарантируются для «аппаратных» клавиатур (физических клавиатур, имеющихся у небольшого круга современных смартфонов, или портативных клавиатур, подключенных через Bluetooth или USB); однако в этом мало пользы, потому что подавляющее большинство событий ввода с клавиатуры, на которые вам понадобится реагировать, будут генерироваться программной клавиатурой. Кроме того, подключение к этим событиям требует довольно сложной настройки, включая привязку к «IME» (методу ввода), регистрацию для передачи фокуса ввода, расширение и сжатие клавиатуры по мере необходимости и т. д.

Углубившись в документацию для разработчиков, можно обнаружить раздел, озаглавленный «Обработка действий клавиатуры». Звучит многообещающе, но нам сразу же показывают предупреждение:

При обработке событий клавиатуры с помощью класса `KeyEvent` и других родственных API следует ожидать, что такие события генерируются только аппаратной клавиатурой. Вы никогда не должны полагаться на получение событий от программного метода ввода (экранной клавиатуры).

И как же нам быть? У нас на выбор есть пара стратегий...

Во-первых, нас, как правило, больше интересуют события изменения редактируемого текста, а не фактическое нажатие клавиши. В этих случаях у нас есть доступ к интерфейсу `TextWatcher`, который требует реализации трех методов:

- `onTextChanged`;
- `beforeTextChanged`;
- `afterTextChanged`.

Объекты `TextView`, включая `EditText`, которые реализуют интерфейс `TextWatchers`, можно настроить на получение событий изменения текста вызовом `addTextChangedListener`.

i Это один из немногих методов, позволяющих подключить несколько обработчиков. Существует также парный ему метод `removeTextChangedListener`.

Используя `TextWatcher`, можно определить момент изменения содержимого текстового поля ввода. Часто это именно то, что нам нужно. Сигнатуры методов интерфейса `TextWatcher` существенно отличаются, тем не менее каждый из них предоставляет доступ к измененному тексту как к экземпляру `Editable` или `CharSequence`:

Java

```
EditText editText = new EditText(this);
editText.addTextChangedListener(new TextWatcher() {
```



```

@Override
public void onTextChanged(CharSequence s, int start, int before, int count) {
    Log.d("MyTag", "onTextChanged: " + s);
}

@Override
public void beforeTextChanged(CharSequence s, int start, int count,
    int after) {
    Log.d("MyTag", "beforeTextChanged: " + s);
}

@Override
public void afterTextChanged(Editable s) {
    Log.d("MyTag", "afterTextChanged: " + s);
}
});

```

Kotlin

```

val editText = EditText(this)

editText.addTextChangedListener(object : TextWatcher {
    override fun onTextChanged(s: CharSequence, start: Int, before: Int, count: Int) {
        Log.d("MyTag", "onTextChanged: $s")
    }

    override fun beforeTextChanged(s: CharSequence, start: Int, count: Int,
        after: Int) {
        Log.d("MyTag", "beforeTextChanged: $s")
    }

    override fun afterTextChanged(s: Editable) {
        Log.d("MyTag", "afterTextChanged: $s")
    }
})

```

Помимо изменения текста и учитывая, что физическую клавиатуру будет использовать очень незначительное число пользователей, мы должны признать, что основной интерес для нас представляет поведение программной клавиатуры и некоторое знакомство с идеей «IME». Аббревиатура «IME» расшифровывается как «Input Method Editor» (редактор методов ввода) и с технической точки зрения может представлять все, что угодно, способное обрабатывать события от аппаратных компонентов, но на самом деле почти всегда относится к управлению программной клавиатурой, обычно через `TextView`, и часто через экземпляр класса `EditText`, наследующего `TextView`, который имеет встроенные функции редактирования.

Как и большинство представлений, IME может настраиваться в макетах XML или программно. К наиболее широко используемым IME API относятся `android:imeOptions` и `TextView.setImeOptions`. Оба принимают целое число, представляющее различные флаги IME, такие как «go», «next», «previous», «search», «done» и «send». Иногда семантика флагов выражает поведение, но это не всегда так. Например, «next» и «previous» изменяют фокус ввода, а «go», «done» и «send» могут явно ничего не делать, но должны передавать разные значения подключенным обработчикам.

Например, можно создать экземпляр `EditText` с флагом `android:imeOptions="actionSend"`. Когда этот экземпляр получает фокус ввода, он откроет экранную клавиатуру с отдельной кнопкой для действия **Send (Отправить)**, часто отображается в виде кнопки с надписью **Send (Отправить)** на языке устройства). Нажатие этой кнопки вызывает `TextView.OnEditorActionListener` для обработки события `onEditorAction` (об этом чуть ниже).

Аналогично можно создать экземпляр `EditText` с флагом `android:imeOptions="actionNext"`, который предлагает отображение программной клавиатуры с кнопкой **Next** (часто со стрелкой вправо). Нажатие этой кнопки обычно передает фокус ввода следующему доступному ИМЕ (возможно, экземпляру `EditText`) в дереве представлений.

Чтобы получить более полный контроль над поведением кнопок ИМЕ, можно использовать `TextView.OnEditorActionListener`. Установить этот обработчик событий в ИМЕ (например, в `EditText`) можно с помощью метода `setOnEditorActionListener` (а удалить обработчик можно вызовом того же метода с аргументом `null`).

Экземпляры `OnEditorActionListener` реализуют единственный метод: `public Boolean onEditorAction(TextView view, int actionId, KeyEvent event)`. Вы можете свободно использовать любые аргументы, передаваемые обработчику, но обычно наибольший интерес представляет флаг `actionId`. В последнем примере, если нажать кнопку со стрелкой вправо, все подключенные экземпляры `OnEditorActionListener` вызовут свои методы `onEditorAction` со следующими параметрами: экземпляр `View`, открывший клавиатуру, целочисленная константа `EditorInfo.IME_ACTION_NEXT` и `KeyEvent`, описывающий событие нажатия клавиши **Next** (<https://oreil.ly/pOZn8>).

Обработка сложных жестов

Для реализации поддержки нестандартных жестов можно использовать несколько механизмов. Самый простой способ, как нам кажется, – просто переопределить метод `onTouchEvent` в экземпляре `ViewGroup` (или `Activity`!) и управлять каждым событием в соответствии с потребностями. Каждое событие перемещения пальца по экрану имеет флаг типа (например, начало выполнения жеста, когда палец касается экрана [`ACTION_DOWN`], продолжение жеста – перемещение пальца по экрану [`ACTION_MOVE`], окончание жеста [`ACTION_UP`] или другие подобные флаги). С помощью этой информации и разумно используя значения времени можно реализовать поддержку любого нестандартного поведения, которое может потребоваться вашему приложению.

Также доступны дополнительные API, помогающие упростить реализацию поддержки нестандартных жестов, например класс `Scroller`, несмотря на свое имя, на самом деле не обслуживает никаких движений прокрутки, но имеет некоторые очень удобные методы для вычисления ускорения и замедления инерциальной прокрутки. `VelocityTracker` позволяет записывать события движения и предоставляет информацию о скорости и ускорении вдоль любой оси.

Если этого недостаточно или вам не нужен полный контроль, можно использовать более простой способ доступа к жестам – `GestureDetector` (или `GestureDetectorCompat` из библиотеки поддержки). Экземпляру `GestureDetector` можно передать `GestureListener` и реализовать обработчики сенсорных событий, в том числе:

- onDown;
- onFling;
- onLongPress;
- onScroll (можно рассматривать как событие «перетаскивания»);
- onShowPress;
- onSingleTapUp.

Для этого вам понадобится экземпляр `GestureDetector`, которому нужно передать экземпляры `Context` и `GestureListener`:

Java

```
public class MyActivity extends Activity implements GestureDetector.OnGestureListener {  
    private GestureDetector mGestureDetector = new GestureDetector(this, this);  
  
    @Override  
    void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.my_activity);  
    }  
  
    @Override  
    public boolean onTouchEvent(MotionEvent e){  
        return mGestureDetector.onTouchEvent(e) || super.onTouchEvent(e);  
    }  
  
    @Override  
    public boolean onDown(MotionEvent event){  
        return true;  
    }  
  
    @Override  
    public boolean onFling(MotionEvent e1, MotionEvent e2, float vX, float vY){  
        return true;  
    }  
  
    @Override  
    public void onLongPress(MotionEvent e) {}  
  
    @Override  
    public boolean onScroll(MotionEvent e1, MotionEvent e2, float dx, float dY){  
        return true;  
    }  
  
    @Override  
    public void onShowPress(MotionEvent e) { }  
  
    @Override  
    public boolean onSingleTapUp(MotionEvent e){  
        return true;  
    }  
}
```

Kotlin

```
class MyActivity : Activity(), GestureDetector.OnGestureListener {  
    private val gestureDetector: GestureDetector = GestureDetector(this, this)
```

```

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

override fun onTouchEvent(e: MotionEvent): Boolean {
    return gestureDetector.onTouchEvent(e) || super.onTouchEvent(e)
}

override fun onDown(event: MotionEvent): Boolean {
    return true
}

override fun onFling(e1: MotionEvent, e2: MotionEvent, vX: Float,
                    vY: Float): Boolean {
    return true
}

override fun onLongPress(e: MotionEvent) {}

override fun onScroll(e1: MotionEvent, e2: MotionEvent, dX: Float,
                     dY: Float): Boolean {
    return true
}

override fun onShowPress(e: MotionEvent) {}

override fun onSingleTapUp(e: MotionEvent): Boolean {
    return true
}
}

```

Экземпляр `GestureDetector` берет на себя основную рутинную работу; он использует системные значения, определяющие величину и направление силы тяготения и силу касания, поэтому вы можете быть уверены, что ваше приложение будет действовать так же, как при использовании `ScrollView` или `RecyclerView`.

Если родительское представление `ViewGroup` содержит дочерние элементы `View`, ожидающие получения сенсорных событий (даже таких простых, как `View.OnClickListener`), управлять и без того сложной системой жестов становится еще сложнее. Вообще говоря, можно использовать `onInterceptTouchEvent` в сочетании с `onTouchEvent` (см. документацию для разработчиков: <https://oreil.ly/qCLNx>); с их помощью вы почти наверняка сможете получить доступ к сенсорным событиям в любом контейнере.

В классе `View` также имеются другие методы для обработки событий¹:

- `onKeyDown(int, KeyEvent)`: вызывается, когда появляется новое событие от клавиатуры;
- `onKeyUp(int, KeyEvent)`: вызывается, когда появляется событие отпускания клавиши;
- `onTrackballEvent(MotionEvent)`: вызывается, когда появляется событие от трекбола;

¹ Из документации для разработчиков Android (<https://oreil.ly/HvEKV>).

- `onTouchEvent(MotionEvent)`: вызывается, когда появляется событие жеста;
- `onFocusChanged(boolean, int, Rect)`: вызывается, когда представление получает или теряет фокус ввода.

Узнать больше об обнаружении жестов можно в замечательном руководстве по Android: <https://oreil.ly/tFb5K>.

iOS

В 2007 году Apple представила новый iPhone и вместе с ним мультисенсорную технологию Multi-Touch. Несмотря на повсеместное распространение этой технологии в наши дни, в ту пору возможность выполнять жесты несколькими пальцами произвела революцию и изменила пользовательский интерфейс. В настоящее время сенсорные методы ввода являются основой взаимодействия со смартфоном, но, безусловно, не единственными. В этой главе рассматриваются два наиболее распространенных метода ввода: с помощью касаний и клавиатуры. Приступим.

Получение события касания и реакция на него

Поддержка сенсорных событий в iOS является, пожалуй, лучшей в отрасли. Она немного изменилась с течением времени, но в основном осталась той же, которая появилась в iOS 4. На сегодняшний день она предлагает самый простой способ перехвата сенсорных событий. Вот, например, как можно организовать получение события касания в представлении с изображением в контроллере представления:

```
class SomeViewController: UIViewController {
    var imageView: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()
        imageView = UIImageView(image: ...)
        let gestureRecognizer =
            UITapGestureRecognizer(target: self, action: #selector(handleTap(_:)))
        gestureRecognizer.numberOfTapsRequired = 1
        imageView.addGestureRecognizer(gestureRecognizer)
    }

    @objc func handleTap(_ gestureRecognizer: UIGestureRecognizer) {
        print("Image tapped!")
    }
}
```

Здесь объявляется класс `SomeViewController`, наследующий `UIViewController`. Большая часть операций в этом классе выполняется внутри `viewDidLoad()`. Это метод жизненного цикла представлений в iOS, и здесь обычно выполняют настройку представления в контроллере. Загляните в главу 2, где более подробно рассказывается о представлениях.

Внутри этого метода настраивается представление с изображением `imageView`. В следующей строке мы объявляем распознаватель жестов типа `UITap-`

`GestureRecognizer`, которому передаются ссылка `self` на этот экземпляр и метод `handleTap(_:)` для вызова этим распознавателем жестов.

После записи значения 1 в свойство `numberOfTapsRequired` распознавателя, указывающего, что должно распознаваться одно касание, мы добавляем распознаватель жестов в созданный перед этим экземпляр представления с изображением. Это необходимо для включения распознавателя в работу. В нашем примере это означает, что всякий раз, когда пользователь коснется представления с изображением, это представление просмотрит список своих распознавателей и определит, какой из них следует вызвать для обработки события.

Затем выбранный распознаватель жестов сам вызовет метод `handleTap(_:)`, который мы сами указали.

i Обратите внимание, что `handleTap(_:)` объявлен как метод `@objc`. Это связано с тем, что `UIGestureRecognizer` и его подклассы требуют передачи `#selector(...)` в качестве действия, выполняемого при активации распознавателя жестов.

В нашем примере использован типичный шаблон, который фактически сводится к двум строкам:

```
let gestureRecognizer = UITapGestureRecognizer(target: self,
action: #selector(handleTap(_:)))
imageView.addGestureRecognizer(gestureRecognizer)
```

Мы создали экземпляр распознавателя жестов и подключили его к представлению.

Распознаватели обладают очень широкими возможностями. Мы поговорим о них ниже, в этой же главе, а пока рассмотрим другой основной источник ввода в iOS – клавиатуру.

Получение события ввода с клавиатуры и реакция на него

В отличие от устройств с Android, компания Apple никогда не выпускала iPhone или iPad со встроенной физической клавиатурой. Теоретически такие клавиатуры могут появиться в будущем, но это крайне маловероятно, учитывая позицию Apple в прошлом. Однако существуют внешние клавиатуры для iPad (встроенные в оригинальные чехлы), и, конечно, к устройству может быть подключена Bluetooth-клавиатура, заменяющая экранную клавиатуру. Тем не менее для экосистемы, столь зависимой от «программных клавиатур», библиотеки поддержки клавиатуры и текстовых полей в UIKit разочаровывающе – и шокирующе – сложны, учитывая, насколько просты в использовании некоторые другие сферы в UIKit.

Например, преимущественный способ редактирования текста в iOS основан на использовании `UITextField` и `UITextView`. Для каждого из этих элементов пользовательского интерфейса существуют отдельные протоколы делегатов. Они немного отличаются функциональными возможностями, но несущественно. Каждый из этих протоколов делегатов, хотя и является надежным, не имеет специального метода для получения событий при изменении содержимого текстового поля.

Есть и другие подходы. Например, к текстовому полю можно подключить обработчик событий редактирования, например:

```

class SomeViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        textField = UITextField(frame: ...)
        textField.addTarget(self, action: #selector(textFieldDidChange(_)),
                           for: .editingChanged)
    }

    @objc func textFieldDidChange(_ textField: UITextField) {
        print(textField.text)
    }
}

```

В этом примере в контроллере представления `SomeViewController` мы определили поле `UIText` с именем `textField`, которое добавляет целевое действие `textFieldDidChange(_:)` для события `.editingChanged`. Всякий раз, когда пользователь будет изменять текст в этом поле, для каждого добавляемого или изменяемого символа будет вызываться метод `textFieldDidChange(_:)`; в нашем примере мы просто выводим содержимое текстового поля вызовом `print(textField.text)`.

Этот способ нормально работает, пока содержимое текстового поля не изменяется программно. После этого наш метод `textFieldDidChange(_:)` замолкает, и текст изменяется без уведомления.

Более надежный способ обнаружения изменений в текстовом поле дает добавление наблюдателя, как показано ниже:

```

class SomeViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        textField = UITextField(frame: ...)
        NotificationCenter.default
            .addObserver(self, selector: #selector(textFieldDidChange(_)),
                       name: UITextField.textDidChangeNotification,
                       object: textField)
    }

    @objc func textFieldDidChange(_ notification: Notification) {
        let textField = notification.object as! UITextField
        print(textField.text)
    }
}

```

Этот пример напоминает предыдущий, но имеет несколько важных отличий. Прежде всего после создания экземпляра `UITextField` мы не устанавливаем обработчик события `.editingChanged`, а организуем прослушивание уведомления `UITextField.textDidChangeNotification`. Теперь наш метод `textFieldDidChange(_:)` будет вызываться всякий раз, когда поступит соответствующее уведомление; однако, чтобы прочитать текст в следующей строке `print(textField.text)`, мы приводим `notification.object` к типу `UITextField`.

До сих пор мы работали только с `UITextField`. А как быть, если понадобится следить за несколькими текстовыми полями ввода и сочетанием объектов `UITextField` и `UITextView`? Ваш код быстро может превратиться в нечто вроде этого:

```

class SomeViewController: UIViewController {
    var textField1: UITextField!
    var textField2: UITextField!
    var textField3: UITextField!
    var textView1: UITextView!
    var textView2: UITextView!
    var textView3: UITextView!

    override func viewDidLoad() {
        super.viewDidLoad()
        NotificationCenter.default
            .addObserver(self, selector: #selector(textFieldDidChange(_:)),
                name: UITextField.textDidChangeNotification,
                object: nil)
        NotificationCenter.default
            .addObserver(self, selector: #selector(textViewDidChange(_:)),
                name: UITextView.textDidChangeNotification,
                object: nil)
    }

    @objc func textFieldDidChange(_ notification: Notification) {
        let textField = notification.object as! UITextField
        doSomething(textField.text!)
    }

    @objc func textViewDidChange(_ notification: Notification) {
        let textView = notification.object as! UITextView
        doSomething(textView.text!)
    }

    private func doSomething(for text: String?) {
        print(text)
    }
}

```

Как сложно!

Но давайте оставим эти меланхоличные мысли и сосредоточимся на чем-то другом. Вернемся снова к сенсорному вводу и обсудим более сложные средства распознавания жестов. Эта сфера принадлежит UIKit и прекрасно масштабируется от простой до весьма сложной логики, не требуя особых трудозатрат от разработчика.

Обработка сложных жестов

Распознаватели жестов отлично справляются с простыми жестами, выполняемыми одним пальцем. Но их также с успехом можно использовать для реализации сложных цепочек взаимодействий. Рассмотрим следующий пример:

```

let doubleTapRecognizer = UITapGestureRecognizer(target: self,
    action: #selector(handleTap(_:)))
doubleTapRecognizer.numberOfTapsRequired = 2

```

Этот код напоминает наш предыдущий пример распознавателя одного касания. Однако, просто изменив значение одного свойства, мы смогли превратить его в распознаватель касания двумя пальцами.

В UIKit есть также другие встроенные распознаватели жестов. Если вам понадобится распознать жест панорамирования тремя пальцами, его можно создать, как показано ниже:

```
let panGestureRecognizer = UIPanGestureRecognizer(target: self,
                                                action: #selector(handlePan(_:)))
panGestureRecognizer.minimumNumberOfTouches = 3
```

Или, если вы решите определить физически сложно осуществимый жест, вот вам пример пятипальцевого жеста тройного касания:

```
let fiveFingerTapRecognizer = UITapGestureRecognizer(target: self,
                                                  action: #selector(handleTap(_:)))
fiveFingerTapRecognizer.numberOfTapsRequired = 3
fiveFingerTapRecognizer.numberOfTouchesRequired = 5
```

Вы едва ли будете часто встречать нечто подобное в приложениях. Однако все сенсорные интерфейсы имеют общую проблему: в одном представлении часто требуется обрабатывать несколько сенсорных событий. Но как обработать жесты одиночного и двойного касания без риска случайно запустить сначала обработчик одиночного касания? Вот как может выглядеть такая реализация:

```
// Создать распознаватель двойного касания
let doubleTapRecognizer = UITapGestureRecognizer(target: self,
                                              action: #selector(handleTap(_:)))
doubleTapRecognizer.numberOfTapsRequired = 2

// Создать распознаватель одного касания
let singleTapRecognizer = UITapGestureRecognizer(target: self,
                                              action: #selector(handleTap(_:)))
singleTapRecognizer.numberOfTapsRequired = 1
singleTapRecognizer.require(toFail: doubleTapRecognizer)
```

Здесь сначала создается распознаватель `doubleTapRecognizer` жестов двойного касания. В свойство `numberOfTapsRequired` распознавателя записывается значение 2. Затем создается распознаватель `singleTapRecognizer` жестов одиночного касания. Количество касаний устанавливается равным 1, а затем вызывается отдельный метод `require(toFail:)`, которому передается распознаватель жестов двойного касания, созданный ранее.

Метод `require(toFail:)` имеет во всех распознавателях жестов и позволяет им запускаться только в случае отказа другого распознавателя. Такое связывание распознавателей позволяет распознавателю одиночного касания дожидаться, когда распознаватель двойного касания потерпит неудачу, и только потом вызвать свой обработчик. Без подобной связи, когда пользователь выполнит жест двойного касания, после первого касания сработает распознаватель одиночного касания, а после второго – распознаватель двойного касания.

Это позволяет увидеть, как можно связать несколько составных жестов и придать им определенные приоритеты выполнения. Количество комбинаций распознавателей жестов, которые можно создать, фактически бесконечно; однако их перечисление выходит за рамки этой книги, но если вам интересно, обращайтесь к описанию `UIGestureRecognizer` в документации для разработчиков.

Программный интерфейс сенсорных событий

Одной из особенностей цепочки респондентов в iOS является программный интерфейс сенсорных событий касания, доступный всем респондентам (например, представлениям и контроллерам представлений). Это невероятно мощный набор методов, которые вызываются всякий раз, когда сенсорный жест начинается, выполняется и, наконец, заканчивается или отменяется. Однако, учитывая простоту и широкие возможности механизма распознавателей жестов, практически всегда лучше использовать последний, за исключением особых случаев, когда пользовательский интерфейс требует более тонкого взаимодействия с сенсорным экраном. В этих случаях вы можете использовать сенсорные события, доступные в объектах `UIResponder`.

Что мы узнали

В этой главе мы увидели сходства и различия в способах получения и обработки пользовательского ввода в Android и iOS. Пользовательский ввод может иметь форму простых касаний, сложных жестов или поступать с экранных и внешних клавиатур.

- Обе платформы имеют похожие механизмы приема и обработки простых сенсорных событий.
- Обе платформы могут получать ввод текста из различных источников, но в iOS требуется написать некоторый код вручную из-за несколько громоздкой схемы приема ввода.
- Существуют механизмы обнаружения и обработки встроенных сложных жестов, но каждая из платформ имеет жесты, которые обычно не используются на другой платформе.

Сенсорный ввод делает устройства на Android и iOS чрезвычайно дружелюбными и интуитивно понятными. Важно понимать, как получать и обрабатывать входные данные, чтобы получилось удобное приложение.

В следующей главе мы познакомимся с объектами и шаблонами, с которыми пользователь сталкивается не так непосредственно, как с теми, что рассматривались до сих пор. Поехали!

Глава 5

Передача сообщений

Передача сообщений – очень обширная и иногда противоречивая область информатики, и к настоящему времени разработано множество моделей и систем передачи сообщений, таких как pub/sub (от англ. publisher/subscriber – издатель/подписчик), диспетчеры событий, обратные вызовы, наблюдатели, очереди сообщений и т. д. Проблема в том, что они часто очень похожи, и вам будет сложно определить практические различия между некоторыми из них. Тем не менее это очень важная функция в любом приложении, и в разработке мобильных приложений используется несколько согласованных стратегий, которые мы обсудим здесь.

Задачи

В этой главе вы узнаете:

- 1) как использовать обратные вызовы для реакции на действия;
- 2) как передать сообщения подписчикам, заинтересованным в их получении;
- 3) как организовать получение и обработку сообщений.

ANDROID

В Android для прямой передачи сообщений и событий обычно используются обратные вызовы с привлечением статистически доступного и потокобезопасного `LocalBroadcastManager`. Обратите внимание, что `LocalBroadcastManager` посылает экземпляры `Intent` объектам `BroadcastReceiver` – это тот же механизм, который используется для передачи сообщений другим приложениям или компонентам операционной системы, только `LocalBroadcastManager` передает сообщения объектам `BroadcastReceiver`, находящимся в одном с ним приложении, что является хорошим (с точки зрения безопасности) и не особенно обременительным ограничением. Если вам понадобится организовать обмен сообщениями между приложениями или с самой системой, вам придется выйти за пределы простого и компактного API `LocalBroadcastManager`.

Использование обратных вызовов для реакции на действия

Самый прямолинейный способ передачи сообщения в Java и Android – обращение к обратному вызову. Обратный вызов – это экземпляр объекта, созданный исключительно для получения результата и его обработки, и часто это простой анонимный экземпляр функционального интерфейса. Например:

Java

```
public class Callbacks {

    public static void requestData(Callback callback) {
        // выполнить некоторый ввод/вывод с файлом или сетевым соединением...
        Object data = // результат операции
        if (data != null && callback != null) {
            callback.onSuccess(data);
        }
    }

    public interface Callback {
        void onSuccess(Object data);
    }
}
```

Kotlin

```
object Callbacks {

    fun requestData(callback: (data: Any?) -> Unit) {
        // выполнить некоторый ввод/вывод с файлом или сетевым соединением...
        val data = Any()
        callback(data)
    }
}
```

Какой-то другой код может вызвать его, используя интерфейс `Callback`, лямбда-выражение или ссылку на метод:

В лямбда-выражении в Java и затем в Kotlin

```
Callbacks.requestData(data -> Log.d("MyApp", "result received: " + data));
Callbacks.requestData { data-> Log.d("MyApp", "result received: $data") }
```

Ссылка на метод в Java и затем в Kotlin

```
private void handleResult(Object data) {
    Log.d("MyApp", "result received: " + data);
}
...
Callbacks.requestData(this::handleResult);

private fun handleResult(data: Any?) {
    Log.d("MyApp", "result received: $data")
}
...
Callbacks.requestData(::handleResult)
```

Интерфейс, только в Java

```
Callbacks.requestData(new Callback(Object data) {  
    Log.d("MyApp", "result received: " + data);  
});
```

В Kotlin нет прямого эквивалента, потому что вместо интерфейсов параметры функции определяются как лямбда-выражения. Вместо этого используйте синтаксис лямбда-выражений и обязательно загляните в описание функций высшего порядка в документации Kotlin (<https://oreil.ly/lr8Ja>).

Как видите, обратные вызовы предлагают довольно прямолинейный способ определения реакции программы на результат операции.

Вот тривиальный, но вполне рабочий пример. На этот раз мы используем обратный вызов для обработки успешного и неудачного выполнения операции:

Java

```
public class FiftyFifty {  
  
    public static void play(SuccessHandler successHandler, FailureHandler failureHandler) {  
        float random = new Random().nextFloat();  
        if (random >= 0.5f) {  
            successHandler.onSuccess();  
        } else {  
            failureHandler.onFailure();  
        }  
    }  
  
    public interface SuccessHandler {  
        void onSuccess();  
    }  
  
    public interface FailureHandler {  
        void onFailure();  
    }  
}
```

Kotlin

```
fun playFiftyFifty(success: () -> Unit, failure: () -> Unit) {  
    val random = Random().nextFloat()  
    if (random >= 0.5f) {  
        success()  
    } else {  
        failure()  
    }  
}
```

Вот как можно их использовать:

Java

```
public class PlayFiftyFifty {  
  
    public PlayFiftyFifty() {  
        FiftyFifty.play(this::onSuccess, this::onFailure);  
    }  
}
```

```

private void onSuccess() {
    Log.d("FiftyFifty", "We won!");
}

private void onFailure() {
    Log.d("FiftyFifty", "We lost :(");
}
}

```

Kotlin

```

class PlayFiftyFifty {
    init {
        playFiftyFifty({ this.onSuccess() }, { this.onFailure() })
    }

    private fun onSuccess() {
        Log.d("FiftyFifty", "We won!")
    }

    private fun onFailure() {
        Log.d("FiftyFifty", "We lost :(")
    }
}

```

А так можно вызвать этот код:

Java

```
new PlayFiftyFifty();
```

Kotlin

```
PlayFiftyFifty()
```

Более практичным примером может служить выполнение множества операций, способных генерировать исключения, например сетевых запросов, требующих аутентификации и возвращающих ответы JSON, которые необходимо проанализировать, а затем сохранить результаты на локальный диск или в базу данных. У вас может быть экземпляр Activity, выполняющий всю эту логику, и вам нужна общая логика обработки сбоев (например, показывающая пользователю Snackbar с описанием сбоя и не прерывающая работу пользователя). В таком случае вам может пригодиться, например, такой класс:

Java

```

public class Safely {
    public static void handle(Attempter attempter, SuccessHandler successHandler,
                             ErrorHandler errorHandler) {
        try {
            attempter.attempt();
            successHandler.onSuccess();
        } catch (Exception e) {
            if (errorHandler != null) {
                errorHandler.onException(e);
            }
        }
    }
}

```

```
    }  
}  
public interface Attempter {  
    void attempt();  
}  
public interface SuccessHandler {  
    void onSuccess();  
}  
public interface ErrorHandler {  
    void onException(Exception e);  
}  
}
```

Kotlin

```
object Safely {  
    fun handle(attempter: () -> Unit, success: () -> Unit, error: (e: Exception) -> Unit) {  
        try {  
            attempter()  
            success()  
        } catch (e: Exception) {  
            error(e)  
        }  
    }  
}
```

Вот как можно использовать эти утилиты в контроллере:

Java

```
public class MainActivity extends Activity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        setContentView(R.layout.activity_main);  
        Safely.handle(this::fetchAndSave, this::handleSuccess, this::handleFailure);  
    }  
    private void fetchAndSave() throws Exception {  
        // представьте, что этот метод устанавливает HTTP-соединение,  
        // передает учетные данные, запрашивает ответ в формате JSON,  
        // анализирует этот ответ и записывает получившиеся значения  
        // в локальную базу данных SQLite  
    }  
    private void handleSuccess() {  
        // здесь можно организовать вывод сообщения о благополучном  
        // получении и сохранении данных;  
        // но пока мы просто запишем сообщение в журнал ;)  
        Log.d("MyApp", "just fetched and saved some data!");  
    }  
    private void handleFailure(Exception e) {  
        // здесь можно организовать вывод Snackbar с описанием  
        // возникшей проблемы или дать возможность повторить
```

```

        // попытку...
        Log.d("MyApp", "Oops! Something went wrong: " + e.getMessage());
    }
}

```

Kotlin

```

class MyActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Safely.handle(::fetchAndSave, ::handleSuccess, ::handleFailure)
    }

    @Throws(Exception::class)
    private fun fetchAndSave() {
        // представьте, что этот метод устанавливает HTTP-соединение,
        // передает учетные данные, запрашивает ответ в формате JSON,
        // анализирует этот ответ и записывает получившиеся значения
        // в локальную базу данных SQLite
    }

    private fun handleSuccess() {
        // здесь можно организовать вывод сообщения о благополучном
        // получении и сохранении данных;
        // но пока мы просто запишем сообщение в журнал ;)
        Log.d("MyApp", "just fetched and saved some data!")
    }

    private fun handleFailure(e: Exception) {
        // здесь можно организовать вывод Snackbar с описанием
        // возникшей проблемы или дать возможность повторить
        // попытку...
        Log.d("MyApp", "Oops! Something went wrong: $e.getMessage()")
    }
}

```

Это были основы использования обратных вызовов! Это довольно простой шаблон, и вы наверняка уже видели или даже использовали нечто подобное.

Передача сообщений подписчикам, заинтересованным в их получении

Менее известный API, характерный для Android и недоступный за пределами этой платформы, – `LocalBroadcastManager`. Давайте рассмотрим его.

`LocalBroadcastManager` – это синглтон, единственный экземпляр, существующий и используемый в приложении. Доступ к этому экземпляру можно получить, передав любой объект `Context` в вызов статического метода `getInstance`: `LocalBroadcastManager lbm = LocalBroadcastManager.getInstance(context);`. Вам никогда не придется беспокоиться о вызове конструктора или настройке. Кроме того, он изначально является потокобезопасным!

Самая замечательная особенность этого синглтона заключается в возможности использовать его для передачи сообщений между объектами, ничего не

знающими друг о друге. Например, с его помощью можно рассылать сообщения из класса `Adapter`, управляющего несколькими разными `RecyclerView`, не заботясь о том, что произойдет при получении этих сообщений. Одно представление списка может обновить свой список в ответ на сообщение конкретного типа; другое – отреагировать как-то иначе, может быть, даже вызвать `finish` своего `Activity`. Экземпляру `Adapter` не нужно знать о существовании экземпляров `RecyclerView` или `Activity`, в которых они размещены, и наоборот – `LocalBroadcastManager` посылает сигнал, а любая заинтересованная сторона сможет отреагировать на него по вашему выбору.

Экземпляр `LocalBroadcastManager` имеет два метода, используемых особенно часто: `sendBroadcast` и `registerReceiver`. Как вы уже наверняка догадались, `sendBroadcast` посылает сообщение и уведомляет любые другие классы, зарегистрировавшиеся как получатели сообщений этого типа. Также важную роль играет метод `unregisterReceiver`, позволяющий прекратить прием сообщений, если получатель был удален из приложения (и тем самым предотвратить утечки памяти!).

Сообщениями в данном случае являются экземпляры класса `Intent`, того же класса, который используется для запуска новых экземпляров `Activity` и описания общесистемных сообщений. Экземпляр `Intent` имеет свойство, описывающее «действие», которое можно инициализировать строкой. Также есть возможность добавить дополнительные действия позже с помощью метода `addAction`.

Синглтон `LocalBroadcastManager` позволяет послать сообщение `Intent`, предлагая для этого метод `sendBroadcast`:

Java

```
Intent intent = new Intent("data-received");
LocalBroadcastManager.getInstance(context).sendBroadcast(intent);
```

Kotlin

```
val intent = Intent("data-received")
LocalBroadcastManager.getInstance(context).sendBroadcast(intent)
```

После вызова этого метода для всех экземпляров `BroadcastReceiver`, зарегистрировавшихся в том же глобальном экземпляре `LocalBroadcastManager` в качестве получателей этого действия, будет вызван метод `onReceive` с соответствующим экземпляром `Intent` во втором аргументе. Обратите внимание, что уведомляться будут только экземпляры `BroadcastReceiver`, зарегистрировавшиеся с помощью `IntentFilter`, включающего действие, соответствующее действию в экземпляре `Intent`. То есть если в предыдущем примере послать `Intent` с другим действием, например `Intent intent = new Intent("userlogin")`, `BroadcastReceiver` не получит сообщения, потому что действия в его фильтре `IntentFilter` и в сообщении `intent` не совпадают.

Получение и обработка сообщений

Давайте посмотрим, как зарегистрироваться для получения рассылаемых сообщений.

Прежде всего нужно создать экземпляр `BroadcastReceiver` и переопределить метод `onReceive`, добавив в него обработку сообщений:

Java

```
BroadcastReceiver receiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d("MyApp", "We received a broadcast for " + intent.getAction());
    }
}
```

Kotlin

```
var receiver:BroadcastReceiver = object:BroadcastReceiver() {
    override fun onReceive(context:Context, intent:Intent) {
        Log.d("MyApp", "We received a broadcast for " + intent.getAction())
    }
}
```

Затем следует создать экземпляр `IntentFilter`, который в действительности просто хранит список строк с названиями действий, нас интересующих:

Java

```
IntentFilter intentFilter = new IntentFilter("data-received");
```

Kotlin

```
val intentFilter = IntentFilter("data-received")
```



`IntentFilter` обладает более широкими возможностями, чем показано здесь, но чаще всего он используется как простой фильтр строк с названиями действий.

Созданный экземпляр `intentFilter` нужно передать в вызов метода `registerReceiver`, после чего `BroadcastReceiver` начнет получать рассылаемые сообщения с действием `"data-received"`:

Java

```
LocalBroadcastManager.getInstance(context).registerReceiver(receiver,
    intentFilter);
```

Kotlin

```
LocalBroadcastManager.getInstance(context).registerReceiver(receiver,
    intentFilter)
```

Обратите внимание, что `LocalBroadcastReceiver` является потокобезопасным, поэтому посылать сообщения с его помощью можно даже из фонового потока и безопасно их получать в потоке пользовательского интерфейса. Однако сама рассылка производится асинхронно, то есть операторы, следующие за вызовом `sendBroadcast`, не обязательно будут выполнены после доставки сообщения получателям. Однако есть метод `broadcastSync`, который гарантирует немедленную доставку сообщений в синхронном стиле.

Также очень важно не забывать отменять регистрацию получателей `BroadcastReceiver`, например перед их удалением в методе `finished` экземпляра `Activity`. В противном случае возможны утечки памяти.

Существует также ряд сторонних систем обмена сообщениями, доступных в Java и Android, включая Otto от Square и EventBus от GreenRobot. Имеется также класс Observable, используемый в очень популярной библиотеке RxJava для Android, но, как нам кажется, шаблон проектирования Наблюдатель довольно сильно отличается от традиционного шаблона Издатель/подписчик и поэтому заслуживает отдельного рассмотрения.

Наконец, без труда можно написать свою шину сообщений (систему отправки и приема событий) на Java; например, мы реализовали такую шину, написав всего 40 строк кода.

iOS

Рассылать сообщения в iOS можно несколькими способами. Так же как в Android, самые распространенные из них – обратные вызовы и рассылка уведомлений. Давайте рассмотрим их различия и определим, когда лучше использовать каждый из этих проверенных шаблонов.

Использование обратных вызовов для реакции на действия

Самый распространенный способ передачи сообщений в Swift и iOS основан на использовании замыканий.

Замыкания

Замыкания – это автономные функции, действующие как объекты. Они могут быть свойствами другого объекта, передаваться методам в параметрах или сохраняться в переменных для последующего использования. В простейшем виде замыкание выглядит примерно так:

```
var someClosure = { print("I'm a closure!") }
someClosure() // Вызов замыкания, которое выведет сообщение в консоль
```

По сути, замыкание – это фрагмент кода, который можно вызвать в любой момент. Иногда замыкания называют «анонимными функциями», и они очень полезны!

Использование замыканий в роли обратных вызовов особенно эффективно и целесообразно в асинхронных операциях, например при ожидании завершения сетевого вызова. Вот пример использования замыкания гипотетическим клиентом для взаимодействия с сервером, чтобы предотвратить приостановку приложения в ожидании ответа:

```
class NetworkService {
    var completion: ((Bool, Error?) -> ())?

    func fetchData(for url: URL) {
        ...
    }

    func onSuccess() {
```

```

        completion?(true, nil)
    }
    func onError(error: Error) {
        completion?(false, error)
    }
}
let api = NetworkService()
api.completion = { success, error in
    if success {
        print("Success!")
    } else {
        print("Uh-oh!")
    }
}
api.fetchData()

```

Разберем этот код подробнее.

Во-первых, свойство `completion` – это хранимый экземпляр замыкания, который может использоваться вмещающим классом позже. Это свойство должно быть отмечено как допускающее возможность хранения значения `nil`, и оно будет иметь значение `nil`, если программа не должна будет как-то реагировать на завершение сетевого вызова.

Метод `fetchData` – это метод-заготовка, который вызывает `onSuccess` или `onError` после получения ответа от гипотетического API.

Далее следует самая важная часть. Методы `onSuccess` и `onError` вызывают хранимое замыкание и передают некоторые данные через параметры. В замыкание можно передать любые данные, но в этом примере передаются логическое значение, описывающее успех или неудачу операции, и необязательная ошибка.

Далее демонстрируется порядок использования этого класса: сначала создается экземпляр `api` класса, и в переменную `completion` записывается замыкание, которое выводит в консоль сообщение об успехе или неудаче. Это то самое замыкание, которое вызывается в методах `onSuccess` и `onError` класса `NetworkService`.

В последней строке мы вызываем метод `fetchData` созданного нами объекта, чтобы запустить сетевой вызов и получить ответ.

Экранированные и неэкранированные замыкания

Одним из более сложных аспектов замыканий, которые важно понимать, является управление памятью. Всякий раз, когда создается замыкание, переменные и экземпляры, содержащиеся в нем, «замыкаются», и создаются сильные ссылки на эти объекты. Поэтому объекты остаются в памяти даже после выхода из области видимости кода, благодаря чему могут использоваться замыканиями. Например:

```

class Incrementor {
    var count = 0

    func increment() {
        count += 1
        print(count)
    }
}

```

```

    }
}
let incrementor = Incrementor()
let closure = {
    incrementor.increment() // 1
    incrementor.increment() // 2
}
closure() // Выведет: "1\n2"
```

Здесь сначала определяется класс, увеличивающий значение своего счетчика на единицу при каждом вызове метода `increment`. Затем создается экземпляр этого класса. Потом этот объект передается в замыкание, в результате чего создается сильная ссылка на него. Это позволяет объекту оставаться в текущем состоянии после выхода из области видимости. В конечном итоге, когда вызывается метод `closure()`, объект все еще хранится в памяти и может увеличить свой счетчик на единицу при вызове метода `increment` (в данном случае метод вызывается дважды).

Актуальность объекта обеспечивается особенностями механизма управления памятью. Swift и его предшественник Objective-C подвержены целому классу ошибок, известных как циклические ссылки, когда объект нельзя удалить после выхода из области видимости, потому что другие объекты удерживают ссылку на него. Поскольку замыкания создают сильную ссылку на хранимые в них объекты, они фактически добавляют «метку» к объектам, которыми владеют. Она говорит компилятору, что тот не должен удалять такие объекты из памяти, потому что они могут продолжать использоваться другими объектами. В нашем примере это пригодилось в замыкании, потому что `incrementor` продолжает оставаться в области видимости и может вызываться позже, при обращении к замыканию.

Замыкание, которое передается в параметре и не живет дольше вызываемой функции, называют *неэкранированным* замыканием. Замыкания этого типа используются по умолчанию, когда передаются в качестве аргумента. Вот пример неэкранированного замыкания:

```

class Incrementor {
    var count = 0

    func increment(with closure: () -> ()) {
        count += 1
        closure()
    }
}

let incrementor = Incrementor()
let printCount = {
    print(incrementor.count)
}

incrementor.increment(with: printCount) // 1
incrementor.increment(with: printCount) // 2
```

Мы изменили метод `increment()` в нашем классе `Incrementor`, чтобы он принимал замыкание в параметре, поэтому теперь сигнатура метода имеет вид

`increment(with:)`. Это замыкание вызывается сразу после увеличения переменной `count` на единицу. В нашем примере мы передаем замыкание, которое просто выводит значение `count`, вызывая его непосредственно из объекта `incrementor`. Мы обращаемся к `incrementor` непосредственно, а это означает, что компилятор создаст сильную ссылку и `incrementor` будет оставаться в памяти до завершения приложения.

В нашем примере аргумент `closure`, передаваемый в `increment(with:)`, является неэкранированным замыканием. Оно никогда не живет дольше вызываемой функции. Давайте теперь рассмотрим экранированное замыкание, чтобы узнать, как оно выглядит:

```
class Incrementor {
    var count = 0
    var printingMethod: (() -> ())?

    func increment(with closure: () -> ()) {
        if printingMethod == nil {
            printingMethod = closure
        }
        count += 1
        printingMethod?()
    }
}

let incrementor = Incrementor()
let printCount = {
    print(incrementor.count)
}

incrementor.increment(with: printCount)
incrementor.increment(with: printCount)
```

Как видите, мы добавили новое свойство `printingMethod` для сохранения полученного замыкания. Далее, в `increment(with:)`, мы записываем полученное замыкание в переменную `printingMethod`. Попробовав скомпилировать этот код, вы получите ошибку компиляции в строке, где выполняется присваивание переменной `printingMethod`, потому что мы не указали, что это замыкание является экранированным. Эта проблема решается просто: достаточно добавить ключевое слово `escape` в объявление параметра `closure` в метод `increment(with:)`, например:

```
func increment(with closure: @escaping () -> ()) {
    if printingMethod == nil {
        printingMethod = closure
    }
    count += 1
    printingMethod?()
}
```

Полный пример теперь выглядит так:

```
class Incrementor {
    var count = 0
    var printingMethod: (() -> ())?
```

```

func increment(with closure: @escaping () -> ()) {
    if printingMethod == nil {
        printingMethod = closure
    }
    count += 1
    printingMethod?()
}
}

let incrementor = Incrementor()
let printCount = {
    print(incrementor.count)
}

incrementor.increment(with: printCount)
incrementor.increment(with: printCount)

```

Это объявление предотвращает ошибку компиляции. Но мы создали другую досадную ошибку, известную как циклическая ссылка. Такие ошибки легко создаются, но с трудом выявляются. Это не очень заметно в нашем небольшом примере, но если у вас имеется большой объект или объект, который создается много раз, вы можете быстро исчерпать доступную память или столкнуться с нежелательными и неожиданными побочными эффектами. Итак, в чем же наша ошибка?

Выше мы упоминали, что при создании замыкания создается сильная ссылка, мешающая удалению замыкания из памяти, пока не исчезнут все ссылки на него. В нашем примере мы создаем сильную ссылку на `incrementor` в замыкании `printCount`. Но эта сильная ссылка на замыкание создается при сохранении его в `printingMethod` внутри метода `increment(with:)`. То есть объект хранит сильную ссылку на самого себя, из-за чего он никогда не выйдет из области видимости и не будет удален из памяти!

К счастью, в Swift есть способ превратить сильную ссылку в слабую. Давайте внутри замыкания объявим `incrementor` как слабую ссылку, например:

```

let printCount = { [weak incrementor] in
    guard let incrementor = incrementor else { return }
    print(incrementor.count)
}

```

Обратите внимание: мы использовали объявление `[weak incrementor]`, чтобы подсказать компилятору, что `incrementor` является слабой ссылкой. Мы также добавили спецификатор `guard` в объявление `incrementor`, чтобы гарантировать невозможность обращения к ссылке, если она равна `nil`. Так мы разорвали цикл, потому что теперь `Incrementor` больше не сохраняет сильную ссылку на себя при сохранении замыкания в `printingMethod` внутри метода `increment(with:)`; он сохраняет слабую ссылку. То есть после последнего обращения к `incrementor` ссылка обнулится, и объект будет благополучно удален из памяти.

В настоящее время замыкания, безусловно, являются наиболее современным способом передачи сообщений между объектами, но иногда их возможности избыточны. Кроме того, использование замыканий чревато ошибками, как только что было показано. Как оказывается, в Cocoa Touch имеются другие способы передачи сообщений между объектами. Давайте теперь обратим наше

внимание на делегатов и посмотрим, чем они отличаются от обратных вызовов на основе замыканий.

Делегаты

Делегаты поддерживались в Cocoa Touch с самого начала создания этой библиотеки. Они обеспечивают логическую простоту, но, как правило, для их использования требуется написать больше кода, чем в случае с замыканиями. Вот тот же самый класс `NetworkService`, но теперь вместо замыкания он использует делегата:

```
protocol NetworkServiceDelegate: class {
    func fetchDidComplete(success: Bool, with error: Error?)
}

class NetworkService {
    weak var delegate: NetworkServiceDelegate?

    func fetchData(for url: URL) {
        ...
    }

    func onSuccess() {
        delegate?.fetchDidComplete(success: true, with: nil)
    }

    func onError(error: Error) {
        delegate?.fetchDidComplete(success: false, with: error)
    }
}

class APIClient {
    init() {
        let api = NetworkService()
        api.delegate = self
        api.fetchData()
    }
}

extension APIClient: NetworkServiceDelegate {
    func fetchDidComplete(success: Bool, with error: Error?) {
        if success {
            print("Success!")
        } else {
            print("Uh-oh!")
        }
    }
}
```

Как видите, здесь мы определили протокол с именем `NetworkServiceDelegate`. Сигнатура метода обратного вызова в этом примере аналогична сигнатуре замыкания `completion` – она включает параметры `Bool` и `Error`, но теперь имеет имя.

В `NetworkService` мы добавили свойство `delegate` для хранения нашего делегата. Это свойство снабжено спецификатором `weak`, чтобы предотвратить образование сильной циклической ссылки. В нем делегат хранит ссылку на родитель-

ский объект; если ссылка будет объявлена сильной, это помешает удалению любого объекта из памяти, и эта ошибка легко может остаться незамеченной.

Обратите внимание, что большая часть кода осталась неизменной, только вместо вызова замыкания в `NetworkService` мы не вызываем делегата напрямую через `delegate?.fetchDidComplete(success:with:)`.

Чтобы фактически вызвать наш API, нужно создать объект, который создаст экземпляр `NetworkService`, установит себя в качестве делегата и вызовет `fetchData()`, чтобы получить данные из сети. В нашем примере эту задачу решает класс `APIClient`.


Наконец, фактическая реализация метода делегата, требуемая протоколом `NetworkServiceDelegate`, реализована как расширение класса `APIClient`. Каждый раз, когда вызывается метод `fetchData`, вызывается метод `fetchDidComplete(success:with:)`, и наше сообщение об успехе или неудаче выводится в консоль.

Как вы наверняка заметили, этот подход требует больше кода, чем подход на основе замыканий, но сами вызовы выглядят довольно просто. Однако этот код быстро может стать слишком громоздким, тем не менее, учитывая, насколько глубоко укоренился этот шаблон в `Socoa Touch`, рано или поздно вы неизбежно столкнетесь с ним. От себя мы можем посоветовать всегда использовать замыкания, когда важна асинхронность или есть только одна ветвь в коде, где можно вызывать замыкание, а делегатов использовать, только когда нужно обеспечить синхронную передачу сообщений.

В некоторых ситуациях использование замыканий и делегатов сопряжено с дополнительными и ненужными сложностями. Когда возникнет необходимость в нескольких замыканиях, выполняемых в тандеме, или в нескольких делегатах, получающих одно и то же сообщение, задумайтесь о возможности использовать `NotificationCenter` и родственные ему механизмы.

Передача сообщений подписчикам, заинтересованным в их получении

Уведомления – это удобный встроенный механизм передачи сообщений в другие части приложения, позволяющий ослабить связь между компонентами. Для получения сообщений с определенными именами объект должен подключиться к общему объекту, в котором другие объекты будут публиковать эти сообщения. К сообщениям могут прикрепляться дополнительные данные и передаваться любым объектам в приложении, подписавшимся на их получение. В терминологии `Socoa Touch` такие сообщения называются «уведомлениями», а в других языках и системах этот шаблон иногда называют `pub/sub` (публикация/подписка) или `observer` (наблюдатель). Эквивалентом такого общего объекта в `Android` является `LocalBroadcastManager`.

 Под «уведомлениями» мы подразумеваем объект `Notification`. Еще большую неразбериху вносит тот факт, что «уведомление» в `iOS` означает нечто совершенно иное, чем в `Android`, и скорее соответствует системным событиям.

Давайте начнем с публикации уведомлений. Имена уведомлений являются строками, поэтому опубликовать уведомление можно передачей самой простой строки:

```
NotificationCenter.default.post(name: Notification.Name("didFinish"), object: nil)
```

Этот код обращается к общему объекту `NotificationCenter.default`, чтобы опубликовать уведомление с именем "didFinish". Это отличный способ, но он редко используется в приложениях, распространяемых через App Store. Проблема в хрупкости такой реализации, использующей простые строки. Объекты, подписавшиеся на ваше уведомление, будут использовать его имя для принятия решений о дальнейших действиях. Если, допустим, вы решите изменить имя уведомления на другое, например `didFinishDownload`, тогда объекты, подписанные на `didFinish`, не получат его.

Эту проблему легко исправить. На самом деле метод `post` принимает не саму строку – мы предварительно обертываем ее перечислением с именем `Notification.Name`. Чтобы решить нашу проблему, можно в классе или структуре создать свойство с типом этого перечисления, использовать это свойство для хранения уже обернутой строки и передавать подписчикам уже это значение. Это изменение позволяет любому объекту предварительно прочитать это свойство перед подпиской, и любые изменения в значении свойства (например, в имени уведомления) будут автоматически обнаруживаться подписчиками. Вот как это может выглядеть:

```
class SomeObject {
    public static let didFinishNotification = Notification.Name("didFinish")
}
```

```
NotificationCenter.default.post(name: SomeObject.didFinishNotification, object: nil)
```

В нашем примере класс `SomeObject` определяет новое статическое свойство с именем `didFinishNotification`, хранящее значение перечисления, которое мы использовали в вызове `post` в первом примере. Затем в вызове метода публикации уведомлений `post` мы используем это свойство вместо объявления значения непосредственно. Это позволяет нам изменить имя уведомления с `didFinish` на любое другое, например `SomeObjectDidFinishNotification`, чтобы предотвратить конфликты имен, допустим.

Также возможно, и на практике это используется довольно часто, прикрепить данные к уведомлению. Это можно сделать с помощью другой версии метода `post`, которая принимает дополнительный аргумент со словарем значений:

```
NotificationCenter.default
    .post(name: SomeObject.didFinishNotification, object: nil,
        userInfo: ["downloadCount": 3])
```

В этом примере мы передаем словарь, содержащий ключ "downloadCount" со значением 3. Таким способом можно передавать любые данные, которые можно поместить в словарь (то есть если они соответствуют интерфейсу `Hashable`). Например, вот как можно передать объект:

```
let anObject = SomeObject()
let count = 3

NotificationCenter.default
    .post(name: SomeObject.didFinishNotification, object: nil,
        userInfo: ["someObject": anObject, "downloadCount": count])
```

Обратите внимание, что в своем примере мы снова использовали обычные строки. Это упрощает иллюстрацию описываемых идей, но на практике этого лучше избегать из-за хрупкости такой реализации. Мы можем исправить данный недостаток, применив тот же прием, что использовался для устранения проблем с именами уведомлений, – включив в класс свойства, соответствующие ключам, например:

```
class SomeObject {
    ...
}

// Инкапсулировать код уведомлений в расширение
extension SomeObject {
    public static let didFinishNotification =
        Notification.Name("SomeObjectDidFinishNotification")
    public static let didFinishNotificationObjectKey = "someObjectKey"
    public static let didFinishNotificationDownloadCountKey = "downloadCount"
}

NotificationCenter.default
    .post(name: SomeObject.didFinishNotification, object: nil,
          userInfo: [SomeObject.didFinishNotificationObjectKey: anObject,
                    SomeObject.didFinishNotificationDownloadCountKey: count])
```

Используя статические свойства для ключей в словаре, можно повисить безопасность уведомлений на этапе компиляции и сделать код более стабильным. Когда число уведомлений становится больше двух-трех, обычно принято выделять код в расширение, как показано выше.

Теперь мы можем публиковать уведомления и посылать данные, но их пока никто не принимает, поэтому поговорим о принимающей стороне. Для начала посмотрим, как подписаться на эти уведомления.

Получение и обработка сообщений

Есть два способа подписаться на уведомление в Swift: с помощью селекторов (методы @objc) и с помощью замыканий. Оба имеют очень похожую природу, но присутствуют небольшие отличия в коде и логике. Вот как выглядит подход на основе селектора (это, пожалуй, наиболее типичный способ):

```
class SomeObject {
    func listenForNotifications() {
        // Подписаться на уведомления
        NotificationCenter.default
            .addObserver(self, selector: #selector(didFinishDownload(_)),
                        name: SomeObject.didFinishNotification, object: nil)
    }

    @objc func didFinishDownload(_ notification: Notification) {
        // Получить данные из уведомления
        let downloadCount =
            notification
                .userInfo?[SomeObject.didFinishNotificationDownloadCountKey]
```

```

        print(downloadCount)
    }
}

```

Разберем этот код.

Сначала мы добавили метод `listenForNotifications` в наш класс. В этом методе мы добавляем себя в список наблюдателей уведомления `SomeObject.didFinishNotification`, созданного ранее. Добавить в качестве наблюдателя можно любой подконтрольный нам объект, но здесь, ради примера, мы передали в первом аргументе ссылку `self`, подписав объект на получение своих же уведомлений. В качестве селектора мы указали `didFinishDownload(_:)` – метод в стиле Objective C, который определяется ниже с использованием спецификатора `@objc`.

В метод `didFinishDownload(_:)` передается объект `Notification`. Этот объект имеет свойство `userInfo` – необязательный словарь, хранящий данные, пересылаемые вместе с уведомлением, как было показано в предыдущем примере. Используя определенный выше ключ, `SomeObject.didFinishNotificationDownloadCountKey`, мы извлекаем данные из уведомления и выводим их в консоль.

Замыкания вместо селекторов

Подход с использованием замыкания для получения уведомления похож на способ с использованием селекторов. Вот как можно организовать получение того же уведомления с применением замыкания вместо селектора:

```

class SomeObject {
    private var observer: AnyObject?

    func listenForNotifications() {
        // Подписаться на уведомления
        self.observer = NotificationCenter.default.addObserver(
            forName: SomeObject.didFinishNotification,
            object: nil, queue: nil) { notification in
                let downloadCount =
                    notification.userInfo?[
                        SomeObject.didFinishNotificationDownloadCountKey]
                print(downloadCount)
            }
    }
}

```

Прежде всего отметим важное отличие наблюдателей-замыканий от наблюдателей-селекторов: наблюдатель-замыкание сохраняется в переменной `observer`. Этот объект создается в результате вызова метода `addObserver(forName:object:queue:using:)` внутри `listenForNotifications()`. Нам нужно сохранить ссылку на этот объект, потому что иначе он будет уничтожен сразу после завершения метода и никогда не будет вызываться. Позже, когда мы отменим подписку на уведомления, мы запишем в это свойство значение `nil`, чтобы освободить память, занимаемую объектом-наблюдателем.

Тело замыкания совпадает с телом селектора `didFinishDownload(_:)` из предыдущего примера; в нем мы извлекаем данные, переданные вместе с уведомлением, и выводим значение в консоль.

Отмена подписки на уведомления

Итак, мы показали, как посылать уведомления и как организовать их получение. Теперь нам осталось показать, как отменить подписку на уведомления. В отличие от отправки, подписки и приема уведомлений, отмена подписки реализуется удивительно просто:

```
// С селекторами
NotificationCenter.default.removeObserver(self)

// С замыканиями
NotificationCenter.default.removeObserver(observer)
self.observer = nil
```

Пример кода для варианта с селекторами отменяет подписку объекта на все уведомления, которые он может прослушивать, и является довольно грубым, но действенным способом разрыва всех связей с NotificationCenter. Однако в реализациях с использованием селекторов лучше отписываться от уведомлений по их именам, например:

```
NotificationCenter.default
    .removeObserver(self, name: SomeObject.didFinishNotification, object: nil)
```

Этот код отменит подписку только на уведомление `SomeObject.didFinishNotification` и оставит все остальные подписки без изменений.

! Важность удаления объекта из списка наблюдателей уведомления может быть неочевидна. Удаляя объект, вы фактически уменьшаете счетчик ссылок на него и позволяете удалить его из памяти, если на него не ссылаются какие-то другие объекты. Утечки памяти – распространенная ошибка, которую так легко допустить. Это особенно верно при использовании замыканий. Всегда удаляйте объекты-наблюдатели и не забывайте присваивать `nil` свойствам, ссылающимся на замыкания.

Получение уведомлений от конкретного издателя

Выше об этом не упоминалось, но вообще есть возможность подписаться на получение уведомлений только от определенных объектов. Для этого нужно передать выбранный объект в вызов метода `addObserver`, как показано ниже:

```
let objectToObserve = SomeObject()

// Нас интересуют только уведомления, посылаемые объектом objectToObserve
NotificationCenter.default
    .addObserver(self, selector: #selector(didFinishDownload(_)),
                name: SomeObject.didFinishNotification,
                object: objectToObserve)
```

Доступно также еще несколько вариантов:

```
// Получать все уведомления от конкретного объекта
NotificationCenter.default
    .addObserver(self, selector: #selector(didFinishDownload(_)),
                name: nil, object: objectToObserve)
```

```
// Получать уведомления didFinishNotification от всех объектов
NotificationCenter.default
    .addObserver(self, selector: #selector(didFinishDownload(_:)),
                 name: SomeObject.didFinishNotification, object: nil)

// Получать все уведомления от всех объектов (не делайте так!)
NotificationCenter.default
    .addObserver(self, selector: #selector(didFinishDownload(_:)),
                 name: nil, object: nil)
```

Уведомления в многопоточном окружении

Уведомления передаются подписчикам в том же потоке выполнения, в каком они были отправлены. То есть если после получения уведомления в фоновом потоке потребуется обновить пользовательский интерфейс, необходимо использовать что-то вроде `DispatchQueue`, чтобы передать событие в основной поток. Это можно сделать, обернув код, который должен выполняться в главном потоке, в блок `DispatchQueue.main.async{...}`, например:

```
class SomeObject {
    ...
    @objc func listenForNotifications(_ notification: Notification) {
        DispatchQueue.main.async {
            updateUI()
        }
    }
}
```

Наблюдение за изменением значений свойств

Socoa Touch имеет целую систему передачи сообщений, которую мы не затронули в этой главе из-за сложности предмета: наблюдение за изменением значений свойств (Key-Value Observation, KVO). Она позволяет организовать наблюдение за единственным свойством объекта и получать обновления при изменении значения этого свойства. К сожалению, это не очень надежный и старый API; его можно использовать только с объектами, которые наследуют `NSObject`, и при неосторожном применении легко допустить ошибку. Но даже при соблюдении всех условий и мер предосторожности мы, авторы книги, считаем, что этот механизм редко когда оказывается лучше описанных выше методов.

Однако если вам интересно, поближе познакомиться с KVO можно в руководстве по программированию (<https://oreil.ly/j8oQV>).

Что мы узнали

Мы узнали много нового в этой главе:

- несколько способов отправки сообщений в iOS и Android;
- самый простой и прямолинейный способ отправки и обработки сообщений заключается в использовании обратных вызовов;

- как отделить друг от друга отправку и обработку сообщений в Android (`LocalBroadcastManager`) и iOS (`NotificationCenter`).

Есть много способов организовать рассылку сообщений в приложении. Выбор во многом зависит от ситуации, но, опираясь на методы, описанные выше, вы без труда сможете заставить разные части вашего приложения общаться друг с другом. Это важное условие получения на выходе легко сопровождаемых и слабосвязанных архитектур.

Глава 6

Файлы

Во многих приложениях возникает необходимость записать данные в файл или прочитать их из файла. Например, приложение может изменить изображение и сохранить его на диск, загрузить видео на SD-карту (предварительно определив ее доступность) или использовать простую индексированную базу данных, состоящую из простых файлов JSON. Логика работы с файлами часто приходится реализовывать при разработке приложений, поэтому хорошее понимание основ важно для большинства разработчиков.

Задачи

В этой главе вы узнаете:

- 1) как определить характеристики файла, такие как размер или дата последнего изменения;
- 2) как читать данные из файлов и записывать их в файлы;
- 3) как скопировать данные из одного файла в другой.

ANDROID

В Android разработчик может определить местоположение внешней SD-карты, сжать группу файлов для выгрузки, организовать наблюдение за изменениями в пользовательских настройках, получить растровые изображения из внешних ресурсов, прочитать файл или вывести сообщение в файл журнала. Даже база данных SQLite (база данных, предоставляемая платформой AOSP) хранится в виде файла, что позволяет приманить к ней ту же логику, чтобы определить размер или создать копию для экспорта.

Поддержка операций с файлами в Java прошла долгий путь, и современные версии языка включают потоковые API, новый пакет `java.nio.file` (`nio` означает `new input/output` – новый ввод/вывод) и некоторые удобные вспомогательные классы, такие как `Files` и `Paths`, поддерживающие разные способы доступа к файловой системе. К сожалению, они недоступны большинству разработчиков для Android, и на момент написания этой книги только две последние версии ОС Android – около 21 % всех установок Android – имели поддержку пакета `java.nio.file`. В настоящее время потоковые API и вспомогательный класс `Files` недоступны ни в одной версии Android.

Но не надо отчаиваться! Проявив немного изобретательности, мы можем использовать существующие фреймворки и стандартную библиотеку для выполнения практически любых операций, какие только могут потребоваться. Чаще всего для подобных целей используется пакет `java.io` (как вы уже догадались... это `input and output` – ввод и вывод) и один из его классов – `java.io.File`. Экземпляр `File` является абстрактным представлением местоположения в локальной файловой системе. Обратите внимание, что экземпляр `File` может представлять и файл, и каталог, и имеет такие методы, как `isDirectory` или `isFile`, помогающие отличать их.

Чтобы получить ссылку на существующий файл, можно передать путь к нему в вызов конструктора `File`: `File file = new File("path/to/file.ext");`. Приложения для Android часто имеют ограниченный доступ к файловой системе – обычно приложению назначается специальный каталог на устройстве, для которого будут установлены разрешения на чтение и запись. Получить ссылку на этот каталог можно вызовом метода `getFilesDir()` любого экземпляра `Context`, который вернет уже готовый экземпляр `File`, представляющий каталог, созданный системой для вашего приложения. Итак, в приложении для Android можно создать новый файл, указав этот каталог в роли корневого каталога:

```
File file = new File(context.getFilesDir(), "path/to/file.ext");
```

Если файл не существует, его нужно создать: `file.createNewFile()`. Также важно гарантировать существование всех промежуточных каталогов в пути к файлу; сделать это можно вызовом метода `file.getParentFile().mkdirs()`, который создаст все необходимые подкаталоги в пути к файлу. То есть применительно к предыдущему примеру `file.mkdirs()` создаст папку с именем `path`, а затем внутри нее еще одну папку с именем `to`. Обратите внимание на множественное число имени `mkdirs`; вызов `file.mkdir()` создаст только один каталог, определяемый экземпляром `File`.

Определение характеристик файла, таких как размер или дата последнего изменения

Получив экземпляр `File`, представляющий файл, можно читать и записывать данные в него или определять его характеристики, такие как размер или дата изменения:

Java

```
File file = new File(context.getFilesDir(), "path/to/file.ext");
long sizeInBytes = file.length();
long lastModifiedTimestamp = file.lastModified();
```

Kotlin

```
val file = File(context.filesDir, "path/to/file.ext")
val sizeInBytes = file.length()
val lastModifiedTimestamp = file.lastModified()
```

Чтение и запись данных в файлы

Первое, что многим приходит на ум при мысли о записи данных в файл, – запись значения типа `String` в экземпляре `File`. Но на самом деле это не самая частая операция в Java и Android – вспомните о двоичных файлах с изображениями, аудио и видео, и сжатыми данными в формате ZIP и TAR. Конечно, существуют API для чтения и записи строк `String` в экземпляры `File`, такие как `FileReader` и `FileWriter`, предлагающие упрощенный способ чтения/записи строк, но мы сосредоточимся на базовом шаблоне чтения/записи данных вообще: потоках байтов. Преимущество работы с данными на уровне байтов состоит в том, что одну и ту же операцию можно использовать для чего угодно: записи текстовых файлов, потоковой передачи мультимедиа, загрузки изображений и т. д.

Но не пугайтесь! Все данные в наших программах уже представлены байтами на некотором уровне, и мы довольно легко можем получить эту информацию с помощью существующих API в стандартной библиотеке. Например, получить байты из `String` можно простым вызовом метода `getBytes()` соответствующего экземпляра.

Кроме того, возможно, вы удивитесь, узнав, что класс `java.io.File` не имеет специальных методов записи или чтения, и на самом деле для записи и чтения необходимо использовать еще один вспомогательный класс-посредник: поток (`InputStream`, `OutputStream` или их подклассы, в зависимости от ситуации). Под потоком в Java подразумевается просто часть данных, которые можно читать последовательно, от начала до конца. Преимущество такого подхода заключается в отсутствии необходимости хранить весь объем информации одновременно. В действительности объект потока может даже не иметь видимого конца! Например, в приложении может понадобиться воспроизвести видео из внешнего источника; используя поток, мы просто читаем байты по мере поступления и буферизируем или отображаем их в соответствии с нашей программой.

Но вернемся к нашему практическому примеру. Для начала попробуем записать данные в файл. Для этого нам понадобится экземпляр `FileOutputStream`. Получить экземпляр `FileOutputStream` можно вызовом конструктора с готовым экземпляром `File`: `OutputStream outputStream = new FileOutputStream(file);`. В классе `FileOutputStream` имеется несколько перегруженных версий метода `write`, но мы пока остановимся на одной из них: `write(byte[] bytes)`. Просто передайте этому методу массив байтов в единственном аргументе, и эти байты будут записаны в экземпляр `File`, на который ссылается `FileOutputStream`.

То есть вот как можно записать некоторый текст в файл:

Java

```
// outputStream - это действительный экземпляр FileOutputStream
String message = "Hello world!";
outputStream.write(message.getBytes());
```

Kotlin

```
// outputStream - это действительный экземпляр FileOutputStream
val message = "Hello world!"
outputStream.write(message.toByteArray())
```

Вот и все! Мы только что записали текст «Hello world!» в файл! Конечно, в Java и Android приходится предусматривать некоторые защитные меры, которые делают реализацию этой простой задачи немного более громоздкой. Например, конструктор `FileOutputStream` может сгенерировать исключение `FileNotFoundException`, а метод `write` – исключение `IOException`. Кроме того, мы должны закрыть поток, закончив работу с ним, что также может вызвать исключение `IOException`. Полный код, реализующий все эти меры предосторожности, выглядит довольно сложно, но ничто не мешает нам заключить его в метод с контролируемым исключением и тем самым немного уменьшить количество строк:

Java

```
public class Files {
    public static void writeToFile(File file, byte[] data)
        throws IOException {
        OutputStream outputStream = null;
        try {
            if (!file.exists()) {
                // сначала нужно создать все промежуточные каталоги
                file.getParentFile().mkdirs();
                // затем создать пустой файл
                file.createNewFile();
            }
            // создать поток вывода для записи данных
            outputStream = new FileOutputStream(file);
            // и записать в него данные
            outputStream.write(data);
        } finally {
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Kotlin

```
@Throws(IOException::class)
fun writeToFile(file: File, data: ByteArray) {
    var outputStream: OutputStream? = null
    try {
        if (!file.exists()) {
            // сначала нужно создать все промежуточные каталоги
            file.parentFile.mkdirs()
            // затем создать пустой файл
            file.createNewFile()
        }
        // создать поток вывода для записи данных
        outputStream = FileOutputStream(file)
        // и записать в него данные
        outputStream.write(data)
    }
}
```

```

    } finally {
        outputStream?.close()
    }
}

```

И затем использовать его в программе:

Java

```

File file = new File(context.getFilesDir(), "path/to/file.ext");
byte[] data = "hello world!".toBytes();
Files.writeToFile(file, data);

```

Kotlin

```

val file = File(context.getFilesDir(), "path/to/file.ext")
writeToFile(file, "hello world!".toByteArray())

```

Чтение данных из файлов реализуется аналогично, с той лишь разницей, как вы наверняка догадались, что вместо `FileOutputStream` используется `FileInputStream`.

Поскольку мы рассуждаем в терминах двоичных данных, нам нужно сделать еще кое-что, чтобы преобразовать содержимое файла в удобочитаемый текст. Если в приложении часто нужно читать и записывать текст в файлы, для этого лучше использовать более простые альтернативы, такие как `FileWriter` и `FileReader`, но, как уже говорилось выше, использование потоков байтов является универсальным решением, и преобразование байтов в строку выполняется тривиально просто.

Конструктор `FileInputStream`, подобно конструктору `FileOutputStream`, также принимает параметр `File`: `InputStream inputStream = new FileInputStream(file);`. После создания потока ввода из него можно извлекать отдельные байты, вызывая метод `read()` без параметров, или большие блоки, передавая в вызов буфер в виде массива байтов. Сейчас мы поговорим о простом методе `read()`, возвращающем один байт, но имейте в виду, что версия, принимающая буфер, обычно действует более эффективно, особенно при работе с большими файлами.

Метод `read` возвращает целое число, представляющее количество прочитанных байтов, при этом значение `-1` обозначает конец файла (и потока). Это число можно привести к типу `char`, чтобы создать строковое представление содержимого файла:

```

// inputStream -- это действительный экземпляр FileInputStream
StringBuilder builder = new StringBuilder();
int byte = inputStream.read();
while (byte != -1) {
    builder.append((char) byte);
    byte = inputStream.read();
}
String message = builder.toString();

```

И снова некоторые методы могут сгенерировать контролируемые исключения, и точно так же нужно закрыть поток, закончив работу с ним, поэтому будет полезно заключить всю эту низкоуровневую логику в единственный метод:

Java

```
public class Files {
    public static String readStringFromFile(File file)
        throws IOException {
        InputStream inputStream = null;
        try {
            // создать поток ввода для чтения из файла
            inputStream = new FileInputStream(file);
            // StringBuilder обеспечивает эффективный способ сборки строк из символов
            StringBuilder builder = new StringBuilder();
            // Для удобочитаемости я разместил эту операцию в отдельной строке,
            // но часто она выполняется внутри условного выражения цикла
            int b = inputStream.read();
            while (b != -1) {
                builder.append((char) b);
                b = inputStream.read();
            }
            // мы можем вставить оператор return сюда, потому что блок finally
            // выполняется всегда!
            return builder.toString();
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
        }
    }
}
```

Kotlin

```
@Throws(IOException::class)
fun readStringFromFile(file: File): String {
    var inputStream: InputStream? = null
    try {
        // создать поток ввода для чтения из файла
        inputStream = FileInputStream(file)
        // StringBuilder обеспечивает эффективный способ сборки строк из символов
        val builder = StringBuilder()
        // Для удобочитаемости я разместил эту операцию в отдельной строке,
        // но часто она выполняется внутри условного выражения цикла
        var b = inputStream.read()
        while (b != -1) {
            builder.append(b.toChar())
            b = inputStream.read()
        }
        // мы можем вставить оператор return сюда, потому что блок finally
        // выполняется всегда!
        return builder.toString()
    } finally {
        inputStream?.close()
    }
}
```

Помимо `FileInputStream`, есть и другие классы потоков ввода, поэтому иногда желательно абстрагировать метод преобразования потока в строку, например:

Java

```
public static String readStringFromFile(File file)
    throws IOException {
    InputStream stream = new FileInputStream(file);
    return getStringFromStream(stream);
}

public static String getStringFromStream(InputStream stream) throws IOException {
    try {
        // StringBuilder обеспечивает эффективный способ сборки строк из символов
        StringBuilder builder = new StringBuilder();
        // Для удобочитаемости я разместил эту операцию в отдельной строке,
        // но часто она выполняется внутри условного выражения цикла
        int b = stream.read();
        while (b != -1) {
            builder.append((char) b);
            b = stream.read();
        }
        // мы можем вставить оператор return сюда, потому что блок finally
        // выполняется всегда!
        return builder.toString();
    } finally {
        if (stream != null) {
            stream.close();
        }
    }
}
}
```

Kotlin

```
object Files {
    @Throws(IOException::class)
    fun readStringFromFile(file: File): String {
        val stream = FileInputStream(file)
        return getStringFromStream(stream)
    }

    @Throws(IOException::class)
    fun getStringFromStream(stream: InputStream): String {
        stream.use { s ->
            val builder = StringBuilder()
            var b = s.read()
            while (b != -1) {
                builder.append(b.toChar())
                b = s.read()
            }
            return builder.toString()
        }
    }
}
}
```

Копирование данных из одного файла в другой

Операции чтения и записи легко объединить, чтобы реализовать копирование любого файла, будь то простой текст или видеозапись концерта! Поскольку нас не интересует тип содержимого файла, нам не нужны «танцы» с преобразованием экземпляров `Character` или извлечением байтов из `String` – сохранение логической независимости обеспечивает удобочитаемость реализации этой операции:

Java

```
public class Files {
    public static void copy(File source, File destination)
        throws IOException {
        OutputStream outputStream = null;
        InputStream inputStream = null;
        try {
            inputStream = new FileInputStream(source);
            outputStream = new FileOutputStream(destination);
            int byte = inputStream.read();
            while (byte != -1) {
                outputStream.write(byte);
                byte = inputStream.read();
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Kotlin

```
@Throws(IOException::class)
fun copyFile(source: File, destination: File) {
    var outputStream: OutputStream? = null
    var inputStream: InputStream? = null
    try {
        inputStream = FileInputStream(source)
        outputStream = FileOutputStream(destination)
        var byteRead = inputStream.read()
        while (byteRead != -1) {
            outputStream.write(byteRead)
            byteRead = inputStream.read()
        }
    } finally {
        inputStream?.close()
        outputStream?.close()
    }
}
```

! Для наглядности в своих примерах мы использовали метод `InputStream.read`. Он возвращает единственный байт. Однако есть возможность значительно улучшить производительность, выполняя чтение блоками в массивы байтов, которые обычно называют «буферами». `InputStream` имеет перегруженные версии метода `read`, принимающие такие буферы, а `OutputStream` имеет аналогичные перегруженные версии метода `write`.

Вот и все! Теперь вы знаете, как читать и записывать данные в файлы в приложениях для Android. Обязательно ознакомьтесь с описанием класса `File` в документации разработчика!

i Библиотека Apache Commons для Java имеет хорошо известный и хорошо продуманный модуль `apache.commons.io`, реализующий поддержку файлов и ввода/вывода. Особое внимание обратите на вспомогательные классы `IOUtils` и `FileUtils`.

iOS

В iOS операции с файлами заложены в основу некоторых очень мощных технологий. В конце концов, необходимость работы с файлами возникает в любом достаточно сложном приложении. Из-за изолированного характера iOS перед началом использования файлов важно прежде познакомиться с некоторыми особенностями организации данных.

Определение характеристик файла, таких как размер или дата последнего изменения

На самом деле доступ к файлам разрешен в двух основных областях: в контейнере пакета приложения и контейнере данных приложения. Начнем с контейнера пакета приложения.

Пакеты приложений

В пакет приложения входит двоичный файл с кодом приложения и все ресурсы, собираемые и распространяемые вместе с приложением. Этот пакет имеет цифровую подпись для защиты от подделки. Благодаря этому невозможно изменить файлы внутри пакета приложения или выполнить запись в его каталог (подробнее об этом ниже).

Доступ к файлам в пакете приложения осуществляется с использованием класса `Bundle` из библиотеки Swift. Приложение может иметь несколько пакетов, поэтому выбор пакета осуществляется с помощью переменной в классе `main`. Чтобы получить доступ к файлу с именем `image.png`, например, нужно создать URL файла:

```
let file = Bundle.main.url(forResource: "image", withExtension: "png")
```

Если файл находится в подкаталоге с именем `sample-images`, создание URL выглядело бы следующим образом:

```
let file =
    Bundle.main.url(forResource: "image", withExtension: "png",
                    subdirectory: "sample-images")
```


Данные (и документы)

Используя статические и неизменяемые файлы из пакета приложения, вы быстро столкнетесь с ограничениями. Рано или поздно появится необходимость читать и записывать пользовательские документы и данные. В iOS каждое приложение получает набор из трех каталогов:

- *Documents*;
- *Library*;
- *tmp*.

Каждый из этих каталогов служит определенной цели, как подробно описано в следующих разделах.

Documents

Сгенерированные пользователем данные и файлы должны помещаться в этот каталог. Файлы из этого каталога автоматически сохраняются в iTunes и iCloud. Также есть возможность включить общий доступ к файлам, чтобы пользователи могли напрямую взаимодействовать с ними.

Library

В папке *Library* имеется несколько предопределенных каталогов, в которые обычно помещаются дополнительные файлы. Наиболее важные из них: *Application Support* и *Caches*. Данные, которые необходимо сохранить для использования в будущем, но не зависящие от пользователя, должны храниться в каталоге *Application Support*. Кешированные данные должны храниться в *Caches*.

tmp

Этот каталог предназначен для временных файлов. Вся ответственность за своевременную очистку этого каталога полностью возлагается на приложение. Однако иногда, когда приложение не используется, система тоже может очищать данный каталог.

Доступ к каталогам

Напрямую создать URL файла или каталога в контейнере данных приложения практически невозможно, из-за сложной и непрозрачной логики формирования путей к этим файлам. Это сделано намеренно, чтобы заставить разработчика использовать для файловых операций класс «мастер на все руки», который называется *FileManager*.

Например, чтобы получить доступ к файлу с именем *image.png* в каталоге *Documents* приложения, можно создать URL, как показано ниже:

```
let file = try? FileManager.default
    .url(for: .documentDirectory, in: .userDomainMask,
         appropriateFor: nil, create: false)
    .appendingPathComponent("image.png")
```

В симуляторе этот код сгенерирует прямой путь к файлу: `~/Library/Developer/CoreSimulator/Devices/CF5BCBA7-C7CA-4484-AB54-7BE938D67ECB/data/Containers/Data/Application/313B2DDD-ABDD-4D14-B6CD-85847F29EF2C/Documents/image.png`.

Обратите внимание на значение перечисления `.documentDirectory`, которое используется для выбора конкретного каталога в контейнере данных приложения. Работа по определению всех родительских каталогов выполняется автоматически классом `FileManager`. Кроме того, есть несколько других предопределенных ключей, предоставляемых этим классом. Например, путь к файлу с именем `data.json` в каталоге *Application Support* конструируется следующим образом:

```
let jsonFile = try? FileManager.default
    .url(for: .applicationSupportDirectory, in: .userDomainMask,
        appropriateFor: nil, create: false)
    .appendingPathComponent("data.json")
```

А так можно создать путь к файлу с именем `download.dat`, хранящемуся в каталоге *tmp*:

```
let tempFile =
    try? FileManager.default.tmpDirectory
        .appendingPathComponent("download.dat")
```

Атрибуты файлов

Теперь, получив представление о структуре файловой системы приложения, рассмотрим сами файловые операции. Примечательно, что в стандартной библиотеке `Swift Standard Library` они довольно тривиальны и требуют для выполнения всего нескольких строк кода.

Продолжим пример с файлом изображения `image.png`, включенным в основной пакет приложения. Размер этого файла можно получить непосредственно из объекта `URL`, созданного для доступа к файлу:

```
let url = Bundle.main.url(forResource: "image", withExtension: "png")
if let resourceValues = try? url.resourceValues(forKeys: [.fileSizeKey]) {
    print(resourceValues.fileSize)
}
```

Переменная `url` – это объект `URL`, который указывает на файл с именем `image` и расширением `png` (т. е. `image.png`), расположенный в пакете приложения. В классе `URL` есть метод, который синхронно извлекает «значения ресурсов» (которые в других операционных системах называются «атрибутами файлов») для указанных ключей. В этом примере мы задали ключ `.fileSizeKey`, который соответствует размеру файла на диске.

Аналогично можно получить другие атрибуты файла, достаточно лишь указать соответствующие ключи. Например, получить дату последнего изменения файла можно, обратившись к ключу `.contentModificationDateKey`. Кроме того, есть возможность получить оба этих атрибута сразу, как показано ниже:

```
let url = Bundle.main.url(forResource: "image", withExtension: "png")
if let resourceValues = try? url.resourceValues(forKeys: [.fileSizeKey,
    .contentModificationDateKey]) {
    print(resourceValues.fileSize)
    print(resourceValues.contentModificationDate)
}
```

Чтение и запись данных в файлы

Для чтения и записи данных в файлы существуют простые удобные способы, поддерживаемые для некоторых объектов в фреймворке Foundation и стандартной библиотеке Swift Standard Library, в том числе String и Data. Например, вот как можно прочитать содержимое текстового файла *file.txt* из каталога *Documents* приложения в строковый объект:

```
let file =
    try? FileManager.default.url(for: .documentDirectory, in: .userDomainMask,
                                appropriateFor: nil, create: false)
                                .appendingPathComponent("file.txt")
// Прочитать файл в строку
let contents = try? String(contentsOf: file, encoding: .utf8)
```

Запись в файл производится аналогично и требует вызова `write(to:atomically:encoding:)`, например:

```
let file =
    try? FileManager.default.url(for: .documentDirectory, in: .userDomainMask,
                                appropriateFor: nil, create: false)
                                .appendingPathComponent("file.txt")
// Прочитать файл в строку
var contents = try? String(contentsOf: file, encoding: .utf8)
...
// Записать строку обратно в тот же файл
try? contents.write(to: file, atomically: false, encoding: .utf8)
```

Поддержка операций с файлами для объекта Data почти не отличается. Это объекты, предназначенные для доступа к данным в форме байтов данных, по аналогии с массивами байтов в стиле языка C. Например, допустим, у вас есть URL файла изображения; вы можете прочитать изображение в память и записать изображение на диск, как показано здесь:

```
// Прочитать данные из файла в объект Data
var data = try? Data(contentsOf: imageUrl)
...
// Записать данные обратно в тот же файл
try? data?.write(to: imageUrl)
```

Все дороги ведут в FileManager

Рано или поздно, разрабатывая приложения для iOS, вам потребуются более сложные операции с файлами. Для этих случаев существует универсальный класс `FileManager`, предлагающий обобщенный поточно-ориентированный подход к реализации сложных операций с файлами.

Для чтения того же файла *file.txt* с использованием `FileManager` придется добавить дополнительную логику:

```
// Получить ссылку на файл в каталоге Documents
let file =
    try? FileManager.default.url(for: .documentDirectory, in: .userDomainMask,
```

```

        appropriateFor: nil, create: false)
        .appendingPathComponent("file.txt")

// Прочитать содержимое файла в объект Data
if let contents = FileManager.default.contents(atPath: file.path) {
    // Преобразовать исходные данные в строку String
    let contentsString = String(data: contents, encoding: .utf8)!
    print(contentsString)
}

```

Первое, на что следует обратить внимание, – это использование `file.path` для преобразования объекта URL в строковое представление абсолютного пути к файлу в файловой системе; это необходимо из-за неполной поддержки URL в `FileManager`. Затем оператор `if let` читает содержимое файла в объект данных. Это обеспечивает некоторую защиту от пустых значений, если `FileManager` не сможет найти или получить доступ к запрошенному файлу. Наконец, если предположить, что файл доступен, из объекта данных создается экземпляр строки с использованием указанной кодировки – в данном примере UTF-8.

Запись в файл выполняется аналогично. Вот как можно записать строку в файл с использованием `FileManager`:

```

let example = "I love tacos."

// Преобразовать строку в объект Data
let exampleData = example.data(using: .utf8)

// Создать файл, используя объект данных (или затереть существующий файл)
FileManager.default.createFile(
    atPath: sharedFile.path, contents: exampleData, attributes: nil) // вернет Bool

```

Успех операции с файлом определяется логическим значением `true` или `false`, возвращаемым вызовом `createFile(atPath:contents:attribute:)`. Этот способ отличается от применения более современного оператора `throws` в Swift, который использует `String` и `Data` напрямую.

Но зачем использовать устаревший способ на основе `FileManager` (он появился в фреймворке Foundation намного раньше, чем стандартная библиотека Swift), который к тому же более громоздкий, если сравнивать с методами экземпляров `Data` и `String`? Ответ заключается в назначении `FileManager`. Класс `FileManager` лучше подходит для реализации более сложных взаимодействий с файлами и каталогами, для исследования иерархии каталогов, проверки существования, удаления, перезаписи, обновления атрибутов файлов и многого другого. Но он не так хорош для простых операций чтения и записи данных.

Копирование данных из одного файла в другой

Чтобы скопировать файл только с использованием методов, предоставляемых классом `String`, потребовалось бы прочитать файл в экземпляр `String` в памяти и затем записать этот экземпляр `String` в другой файл вызовом `write(to:atomically:encoding:)`. Это не самый эффективный способ копирования файлов, и его не получится использовать, если размер файла превышает объем доступной памяти.

Вот как можно скопировать файл с помощью FileManager:

```
// Получить ссылку на исходный файл
let originalFile = try? FileManager.default
    .url(for: .documentDirectory, in: .userDomainMask,
        appropriateFor: nil, create: false)
    .appendingPathComponent("file.txt")

// Получить ссылку на новый файл, куда требуется скопировать данные
let copiedFile = try? FileManager.default
    .url(for: .documentDirectory, in: .userDomainMask,
        appropriateFor: nil, create: false)
    .appendingPathComponent("newFile.txt")

// Скопировать файл
try? FileManager.default.copyItem(at: originalFile, to: copiedfile)
```

Это более эффективный способ копирования, чем в первом примере с объектом String, потому что FileManager не требует чтения файла в память перед копированием. Кроме того, FileManager использует преимущества Apple File System (APFS), проприетарной файловой системы Apple, которая поддерживает невероятно эффективный процесс создания клонов объектов.

В FileManager кроется еще немало приятных сюрпризов, которые вам предстоит обнаружить. Но и того, о чем мы рассказали, вполне достаточно, чтобы начать использовать основные файловые операции в iOS. Обязательно загляните в документацию Apple, чтобы получить представление обо всех возможных операциях.

URL и строки

Во всех примерах кода в этой главе для передачи информации о пути к файлу использовались объекты URL вместо строк String. Это связано с тем, что объекты URL, как правило, лучше подходят для хранения информации о пути и делают это более эффективно. Точно так же они поддерживают более эффективные способы изменения представления пути к файлу, позволяя добавлять каталоги, изменять имена и многое другое.

Некоторые методы, предлагаемые классом FileManager и другими API из фреймворка Foundation, используют строки вместо объектов URL. Объекты URL позволяют получить путь в строковом представлении, для чего достаточно обратиться к переменной экземпляра path. Например:

```
let fileURL = Bundle.main.url(forResource: "file", withExtension: "txt")!
let filePath = fileURL.path
```

Резервное копирование в iCloud и iTunes

Еще один важный аспект, который мы не охватили, – это синхронизация файлов в iCloud. Запись в файлы в iCloud производится как обычно, подобно файлам в пользовательском каталоге *Documents*, но для ссылки на целевой файл используется другой контейнер. Все файлы в пользовательских каталогах *Documents* и *Application Support* автоматически сохраняются в iCloud и iTunes.

Можно и рекомендуется предотвращать резервное копирование определенных файлов в iTunes и iCloud. Например, большой видеофайл, полученный из

веб-службы, который можно в будущем загрузить снова, желательно исключить из процесса резервного копирования, установив атрибут `isExcludedFromBackupKey` непосредственно в URL, как показано ниже:

```
var values = URLResourceValues()  
values.isExcludedFromBackup = true  
try? fileUrl.setResourceValues(values)
```

ЧТО МЫ УЗНАЛИ

Как вы могли заметить, операции с файлами в Android и iOS имеют много общего, и в то же время они совершенно разные. В конце концов, мы имеем дело с битами и байтами, но способы доступа к этим данным определяются двумя совершенно разными архитектурными парадигмами.

Потоковый подход к чтению и записи данных в Android резко отличается от более процедурного и запутанного подхода в iOS. Кроме того, в iOS можно заметить артефакты, доставшиеся в наследство от UNIX, которые пронизывают всю файловую структуру.

Мы надеемся, что этот обзор двух подходов послужит вам отправной точкой для понимания операций с файловой системой в обеих платформах. В следующей главе мы поговорим о сохранении данных вне файлов и перейдем в область баз данных и графов.

Глава 7

Хранение данных

Механизмы хранения данных позволяют приложениям формировать то, что может выглядеть как память. Существует несколько способов организовать такое хранение в мобильных приложениях, но наиболее распространенным является применение реляционной базы данных. С самого начала Android и iOS поддерживали возможность подключения, чтения и записи в базы данных.

В результате этого сеанс пользователя в приложении больше не является чем-то эфемерным, а привязан к определенному моменту времени. Приложение может сохранять свои данные и информацию о состоянии, а также восстанавливать их. Несмотря на совершенно разные архитектуры и подходы, существует общий набор функциональных возможностей, поддерживаемых в Android и iOS, что позволяет обсуждать их одновременно.

Задачи

В этой главе вы узнаете:

- 1) как установить соединение с базой данных;
- 2) как создать в базе данных таблицу или хранимый объект;
- 3) как записать данные в таблицу или хранимый объект;
- 4) как прочитать данные из таблицы или хранимого объекта.

ANDROID

В Android в роли основной системы управления базами данных используется SQLite. Версия SQLite зависит от версии Android: на момент написания этой книги использовались версии от 3.4 в Android 1 до 3.19 в Android 27. За более точной информацией обращайтесь к документации разработчиков для Android (<https://oreil.ly/lQegA>).

SQLite – это система управления реляционными базами данных (СУРБД), очень похожая на MySQL и PostgreSQL. Конечно, есть некоторые отличия в поддерживаемых типах данных, функциях и реализациях некоторых конструкций, таких как ALTER TABLE, но в целом они очень похожи, и если у вас есть опыт работы с базами данных SQL, вы без труда освоите и SQLite.

Соединение с базой данных

В Android (не в традиционном программировании на Java) вы будете пользоваться в основном классом `SQLiteOpenHelper`. Но это абстрактный класс, поэтому нужно определить свой подкласс. Например:

Java

```
public class DBHelper extends SQLiteOpenHelper {
    public DBHelper(Context context) {
        super(context, "db", null, 1);
    }

    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) {
        // пока ничего не делает
    }

    @Override
    public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i1) {
        // пока ничего не делает
    }
}
```

Kotlin

```
class DBHelper(context: Context) : SQLiteOpenHelper(context, "db", null, 1) {
    override fun onCreate(sqLiteDatabase: SQLiteDatabase) {
        // пока ничего не делает
    }

    override fun onUpgrade(sqLiteDatabase: SQLiteDatabase, i: Int, i1: Int) {
        // пока ничего не делает
    }
}
```

Метод `onUpgrade` используется для добавления или миграции (переноса) данных при внесении изменений в схему базы данных. Это происходит при изменении номера версии вашей базы данных (четвертый аргумент конструктора) и вызывается только один раз для каждого изменения. На самом деле мы должны были бы сказать «при увеличении», а не «при изменении», потому что попытка использовать БД с версией ниже существующей на устройстве сразу же приведет к сбою.

Обратите внимание, что следование соглашениям об использовании общедоступных констант для имен таблиц и столбцов может облегчить чтение или запись данных в таблицы в других классах.

Чтобы получить экземпляр базы данных с помощью этого вспомогательного класса, нужно вызвать метод `getWritableDatabase` или `getReadableDatabase` в зависимости от требуемого режима доступа. В режиме доступа для записи возможно все то же, что в режиме для чтения, поэтому, планируя выполнять чтение и запись, используйте `getWritableDatabase`. Режим доступа для чтения может пригодиться, разве что когда важно гарантировать неизменность содержимого базы данных при выполнении операций.

Вот как можно получить экземпляр базы данных для записи:

Java

```
SQLiteDatabase database = new DbHelper(this).getWritableDatabase();
```

Kotlin

```
val database = DbHelper(this).getWritableDatabase()
```

С этого момента вы будете иметь доступ к экземпляру `SQLiteDatabase`, имеющему методы для выполнения основных операций, таких как чтение и вставка данных, а также более универсальные для выполнения произвольных SQL-запросов, такие как `execSQL`.

Создание таблицы или хранимого объекта

В своем подклассе у вас будет возможность переопределить методы `onCreate` и `onUpgrade`. Сейчас мы сосредоточимся только на первом. Метод `onCreate` вызывается лишь один раз, при создании экземпляра класса впервые (то есть при первом создании экземпляра, обычно почти сразу после первой установки приложения). Последующие запуски приложения не будут приводить к вызову `onCreate`.

Метод `onCreate` – удобное место для создания начальной схемы. Например, представим, что нам нужно создать базу данных с единственной (пока) таблицей `USAGE_EVENTS`, в которой будут храниться идентификаторы, названия и время событий. Для создания базы данных в методе `onCreate` нашего подкласса мы будем использовать стандартный SQL. Дополним пример, показанный выше, и добавим некоторые строковые константы, которые пригодятся нам при конструировании SQL-оператора `create table` и его выполнении в `onCreate`:

Java

```
private class DbHelper extends SQLiteOpenHelper {

    public static final String TABLE_NAME = "EVENTS";
    public static final String COLUMN_ID = "ID";
    public static final String COLUMN_NAME = "NAME";
    public static final String COLUMN_TIMESTAMP = "TIMESTAMP";
    public static final String CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " +
        TABLE_NAME + " (" +
        COLUMN_ID + " STRING PRIMARY KEY, " +
        COLUMN_NAME + " STRING, " +
        COLUMN_TIMESTAMP + " INTEGER);";

    public DbHelper(Context context) {
        super(context, "db", null, 1);
    }

    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase) {
        sqLiteDatabase.execSQL(CREATE_TABLE);
    }

    @Override
```

```
public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i1) {
    // пока ничего не делает
}
}
```

Kotlin

```
private class DbHelper(context: Context) : SQLiteOpenHelper(context, "db", null, 1) {
    override fun onCreate(database: SQLiteDatabase) {
        database.execSQL(CREATE_TABLE)
    }
    override fun onUpgrade(database: SQLiteDatabase, i: Int, i1: Int) {
        // пока ничего не делает
    }
    companion object {
        const val TABLE_NAME = "EVENTS"
        const val COLUMN_ID = "ID"
        const val COLUMN_NAME = "NAME"
        const val COLUMN_TIMESTAMP = "TIMESTAMP"
        const val CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " +
            TABLE_NAME + " (" +
            COLUMN_ID + " STRING PRIMARY KEY, " +
            COLUMN_NAME + " STRING, " +
            COLUMN_TIMESTAMP + " INTEGER);"
    }
}
```

Запись данных в таблицу или хранимый объект

Как мы уже видели, `SQLiteDatabase` способен выполнять произвольные команды SQL, но, кроме того, предоставляет методы для выполнения операций CRUD (Create, Read, Update, Delete – создать, прочитать, изменить, удалить), включая операцию вставки. Вдобавок к этому `SQLiteDatabase` поддерживает транзакции, которые можно использовать для дополнительного контроля над операциями записи в базу данных. Рассматривая следующий простой пример, вам может показаться, что использование транзакции ничего особенного не дает, но в действительности это не так – она позволяет отлавливать ошибки и реагировать на них; что еще более важно, если операция вставки завершится неудачей – или если мы добавим больше операций с базой данных в блок `try`, терпящий неудачу, – транзакция *не* будет установлена как успешная и автоматически откатит все операции, производившиеся в блоке `try`.

Несмотря на кажущуюся простоту, этот пример демонстрирует шаблон, который вы наверняка захотите использовать в своей практике применения транзакций, – просто добавьте больше операций в блок `try` и сделайте что-нибудь в блоке `catch`. (Напомню, что исключение `Exception` повлечет неявный откат транзакции – явного метода, выполняющего откат, не существует.) Этот шаблон подходит также для гораздо более сложных операций записи и изменения данных:

Java

```
ContentValues contentValues = new ContentValues();
contentValues.put(DbHelper.COLUMN_ID, UUID.randomUUID().toString());
contentValues.put(DbHelper.COLUMN_NAME, "Request Data");
contentValues.put(DbHelper.COLUMN_TIMESTAMP, System.currentTimeMillis());
SQLiteDatabase database = getDatabase();
database.beginTransaction();
try {
    database.insert(DbHelper.TABLE_NAME, null, contentValues);
    database.setTransactionSuccessful();
    Log.d("MyTag", "wrote data to database");
} catch (Exception e) {
    Log.d("MyTag", "there was a problem inserting data: " + e.getMessage());
} finally {
    Log.d("MyTag", "finally");
    database.endTransaction();
    database.close();
}
```

Kotlin

```
val contentValues = ContentValues()
contentValues.put(COLUMN_ID, UUID.randomUUID().toString())
contentValues.put(COLUMN_NAME, "Request Data")
contentValues.put(COLUMN_TIMESTAMP, System.currentTimeMillis())
writableDatabase.beginTransaction()
try {
    writableDatabase.insert(DbHelper.TABLE_NAME, null, contentValues)
    writableDatabase.setTransactionSuccessful()
    Log.d("MyTag", "wrote data to database")
} catch (e: Exception) {
    Log.d("MyTag", "there was a problem inserting data: " + e.message)
} finally {
    Log.d("MyTag", "finally")
    writableDatabase.endTransaction()
    writableDatabase.close()
}
```

Чтение данных из таблицы или хранимого объекта

Метод `SQLiteDatabase.query` предлагает наиболее прямой способ получить информацию из базы данных. Но сигнатура наиболее практичной перегруженной версии включает восемь параметров! Однако давайте притворимся, что нас это не пугает, и посмотрим, как пользоваться этим методом.

Ниже демонстрируется самая простая версия метода (принимает всего семь параметров!). Она позволяет получить из таблицы все записи со всеми столбцами в виде экземпляра `Cursor`, поддерживающего навигацию по набору данных. Фактически это запрос `"SELECT * FROM EVENTS"`:

Java

```
SQLiteDatabase database = new DbHelper(this).getWritableDatabase();
database.query(DbHelper.TABLE_NAME, null, null, null, null, null, null);
```

Kotlin

```
val database = DbHelper(this).getWritableDatabase()
database.query(DbHelper.TABLE_NAME, null, null, null, null, null, null);
Заглянем внутрь, используя более реалистичный запрос:
```

Java

```
SqliteDatabase database = new DbHelper(this).getWritableDatabase();
database.query(
    DbHelper.TABLE_NAME, // Строка с именем таблицы
    new String[] { DbHelper.COLUMN_ID }, // Массив строк с именами столбцов
    DbHelper.COLUMN_NAME + " = ?", // Строка для предложения WHERE
    new String[] { "Request Data" }, // Массив строк со значениями для WHERE
    null, // GROUP BY
    null, // HAVING
    null, // ORDER BY
    "1" // Строка для предложения LIMIT
);
```

Kotlin

```
val database = DbHelper(context).writableDatabase
database.query(
    DbHelper.TABLE_NAME, // Строка с именем таблицы
    arrayOf(DbHelper.COLUMN_ID), // Массив строк с именами столбцов
    DbHelper.COLUMN_NAME + " = ?", // Строка для предложения WHERE
    arrayOf("Request Data"), // Массив строк со значениями для WHERE
    null, // GROUP BY
    null, // HAVING
    null, // ORDER BY
    "1" // Строка для предложения LIMIT
)
```

Предыдущий код фактически выполняет запрос `SELECT ID FROM EVENTS WHERE NAME = \"Request Data\" LIMIT 1`. Использование символа-заполнителя – распространенный прием, помогающий предотвращать атаки типа «инъекция SQL».

Но где хранятся данные, полученные в случае успешного выполнения запроса? Он находится в наборе данных в памяти, доступ к которому можно получить с помощью `Cursor`; метод `query` возвращает экземпляр `Cursor`. Например:

Java

```
Cursor cursor = database.query(DbHelper.TABLE_NAME, null, null, null, null,
    null, null);
while (cursor.moveToNext()) {
    Log.d("MyTag", "id: " + cursor.getString(0));
    Log.d("MyTag", "name: " + cursor.getString(1));
    Log.d("MyTag", "time: " + cursor.getLong(2));
}
cursor.close();
```

Kotlin

```
val cursor = database.query(DbHelper.TABLE_NAME, null, null, null, null,
    null, null)
```

```
while (cursor.moveToNext()) {
    Log.d("MyTag", "id: " + cursor.getString(0))
    Log.d("MyTag", "name: " + cursor.getString(1))
    Log.d("MyTag", "time: " + cursor.getLong(2))
}
cursor.close()
```

Как видите, курсор позволяет получить доступ к столбцам по их индексам, однако в классе `Cursor` есть также метод `getColumnByIndex`, возвращающий имя столбца по его индексу, который можно использовать примерно так:

Java

```
cursor.getString(cursor.getColumnIndex(DbHelper.COLUMN_ID));
```

Kotlin

```
cursor.getString(cursor.getColumnIndex(DbHelper.COLUMN_ID))
```

Существует также масса других доступных API (и сторонних механизмов объектно-реляционного отображения [Object-Relational Mapping, ORM]), и возможности, которые предлагает поддержка SQL, ограничены только вашей фантазией. Если вы работаете над приложением, которое регулярно обращается к базе данных, подумайте об использовании метода `compileStatement`, который компилирует строку с запросом SQL и позволяет переопределять значения параметров при каждом вызове. Этот метод действует очень быстро и действительно может улучшить воспринимаемую производительность при работе с большими наборами данных.

На момент написания этой книги компания Google рекомендовала использовать библиотеку `Room`. Вот выдержка из документации для разработчиков Android (<https://oreil.ly/rtG9C>):

Все эти API (`SQLiteDatabase` и др.) обладают широкими возможностями, но они довольно низкоуровневые и требуют много времени и сил для использования:

- запросы SQL не проверяются во время компиляции. При изменении схемы данных вам придется изменить соответствующие SQL-запросы. Этот процесс может занять много времени и чреват ошибками;
- для преобразований между запросами SQL и объектами данных приходится писать много шаблонного кода.

По этим причинам мы настоятельно советуем использовать библиотеку `Room Persistence Library` в роли слоя абстракции для доступа к информации в базах данных `SQLite` из вашего приложения.

Почему SQLite? Почему не Room? Почему не Realm? Почему не <вставьте сюда свой механизм ORM>?

Мы (авторы этой книги) использовали `SQLite` в качестве примера по нескольким причинам: во-первых, так же как и в случае с другими шаблонами и наборами инструментов в экосистеме Android, эта система управления базами данных появилась раньше `Room`. Во-вторых, SQL хорошо известен разработчи-

кам, создающим приложения на любых языках, и с успехом может переносить запросы между платформами.

iOS

Основные отличия в механизмах хранения между iOS и Android связаны с используемой технологией. В Android, как вы видели, широко используется база данных SQLite и классы, напрямую взаимодействующие с базой данных посредством сетевых соединений и SQL. В iOS, напротив, технологии уровня баз данных абстрагированы в виде собственного объектного графа для хранения данных, который называется Core Data. В принципе, Core Data может использовать базу данных SQLite (и нередко использует) для хранения данных, но также может работать исключительно в оперативной памяти, играя роль временного хранилища, или использовать файлы XML. То есть база данных становится деталью реализации, которую разработчик может игнорировать.

Настройка соединения со слоем хранения данных

Даже притом что, как предполагается, разработчики могут забыть о различиях между графом объектов Core Data и простой базой данных SQLite, они все же имеют некоторые уникальные черты, которые мы используем в этой главе о хранении данных для сравнения iOS и Android. Первая уникальная черта, свойственная Core Data, – отсутствие необходимости устанавливать соединение с базой данных; обработка SQL-запросов и установка соединений выполняются самим механизмом Core Data. Однако для «раскрутки» стека Core Data требуется настроить соединение.

Настройка стека Core Data

Начать работу с Core Data в новом или существующем проекте совсем несложно, но прежде вам нужно понять, что такое Core Data. Начнем со слоя хранения. Предположим, что мы будем использовать стек Core Data в связке с базой данных SQLite. Это, пожалуй, самый распространенный вариант использования Core Data.

Над базой данных SQLite и слоем хранения находится объект `NSPersistentStoreCoordinator`. Этот объект поддерживает связь между базой данных, моделью управляемых объектов и контекстом управляемых объектов. Он использует модель управляемых объектов, определяющую связи между объектами данных, и преобразует эти объекты и отношения между ними в таблицы базы данных и запросы SQL. Он также оптимизирует количество доставляемых объектов и является настоящим «мозгом» Core Data. Его можно считать координатором трафика между SQLite и остальной частью приложения.

Модель управляемых объектов, или `NSManagedObjectModel`, определяется для каждого проекта в файле `.xcdatamodeld` и используется координатором хранилища для определения структуры данных. В этот файл разработчики добавляют новые объекты, отношения и свойства – именно так создается и определяется модель данных в проекте. Объекты, создаваемые на основе модели, называ-

ются «управляемыми объектами» и наследуют класс `NSManagedObject`, который является частью инфраструктуры Core Data. Большая часть работы с Core Data выполняется на этом уровне вместе со следующим классом, который мы обсудим: контекстом управляемого объекта.

Контекст управляемого объекта, или `NSManagedObjectContext`, можно рассматривать как своеобразный черновик, в котором можно изменять данные в экземплярах объектов модели до их сохранения в базе данных. Контекст управления управляемого объекта – это объект, который используется для описания контекста, помогающего инфраструктуре Core Data понять, как должен создаваться и изменяться ваш объект. Обычно в приложениях, использующих Core Data, есть два типа контекстов: контекст представления (действующий в основном потоке выполнения) и фоновый, или закрытый, контекст, который используется для сохранения объектов и фактически преобразует экземпляры `NSManagedObject` в записи в базе данных с помощью команд `INSERT` и `UPDATE`.

Прежде чем сохранять данные и/или извлекать их из хранилища, необходимо настроить стек Core Data. Обычно настройка выполняется в момент запуска приложения. Часто большинство настроек выполняются в методе `application(_:didFinishLaunchingWithOptions:)` делегата приложения.

Раньше процесс настройки был довольно громоздким, но теперь в Core Data есть удобный класс `NSPersistentContainer`, инициализирующий стек Core Data. Этот класс выполняет все необходимые настройки, описанные в файле с определением модели управляемых объектов, и имеет удобный метод для асинхронной загрузки хранилища. Вот как он используется:

```
let persistentContainer = NSPersistentContainer(name: "MyModel")
persistentContainer.loadPersistentStores { (description, error) in
    // Обработчик завершения загрузки
}
```

Этот код создает контейнер для целевой модели управляемых объектов с именем "MyModel" и сохраняет ссылку на него в переменной `persistentContainer`. Следующая строка загружает хранилище, в данном случае базу данных SQLite, с диска и по окончании вызывает обработчик завершения для проверки ошибок и выполнения любого необходимого кода. Если добавить этот код в код запуска в делегате приложения, приложение найдет файл модели управляемых объектов `MyModel.xcdatamodeld` в пакете приложения и использует его для создания или загрузки файла базы данных SQLite с именем `MyModel.sqlite` из каталога *Application Support*.



При желании есть возможность настроить местоположение файлов базы данных. Исторически для хранения баз данных SQLite использовался каталог *Documents*. Ваши требования могут отличаться, и для их удовлетворения вы можете добавить `NSPersistentStoreDescription` после создания `NSPersistentContainer` и сохранить его в `persistentStoreDescription`. Есть и другие варианты описания контейнера хранилища, но они выходят за рамки этой главы.

Внутри замыкания, обрабатывающего событие завершения загрузки, нужно проверить наличие ошибок, а затем обновить пользовательский интерфейс, который может ожидать загрузки хранилища. Вот более полный пример загрузки контейнера хранилища:

```

let persistentContainer = NSPersistentContainer(name: "MyModel")
persistentContainer.loadPersistentStores { (description, error) in
    guard let error = error else {
        // Сообщить об ошибке пользователю и/или попытаться исправить ее
        return
    }
    // Обновить пользовательский интерфейс
    // и/или начать использовать Core Data
}

```

Теперь, когда стек Core Data запущен и работает, посмотрим, как определять объекты.

Определение и создание таблицы или хранимого объекта

Объекты модели определяются в файле модели управляемых объектов, созданном в XCode. Чтобы добавить такой файл в существующий проект, найдите в меню пункт File > New > File (Файл > Создать > Файл) и в разделе Core Data выберите Data Model (Модель данных). В качестве имени файла используйте имя, которое будет передаваться контейнеру хранилища на этапе инициализации.

Этот файл можно редактировать внутри Xcode. Чтобы создать новый управляемый объект, щелкните на кнопке Add Entity (Добавить объект). В описании вы увидите поля Attributes (Атрибуты), Relationships (Отношения) и Fetched Properties (Извлекаемые свойства). Вы можете добавлять свойства в объекты, изменять имена объектов и связывать объекты с классами поддержки.

Отдельные свойства добавляются как атрибуты. Они соответствуют низкоуровневым типам данных в классах Swift, таким как String и Int, которые создаются (или предоставляются вручную) для каждого объекта в модели управляемых объектов. Важно помнить, что модель управляемых объектов только описывает объекты и никак не взаимодействует с базой данных. По сути, она просто отображает код Swift в представление, понятное координатору хранилища.

Чтобы добавить новый атрибут, щелкните на кнопке «+» в области Attributes (Атрибуты) редактора. Если, к примеру, вы добавили свойство с именем title и с типом String в объект с именем MyEntity, Xcode автоматически сгенерирует класс во время компиляции, скрытый от проекта Xcode, который будет выглядеть примерно так:

```

import Foundation
import CoreData

@objc(MyEntity)
public class MyEntity: NSManagedObject {
}

extension MyEntity {
    @nonobjc public class func fetchRequest() -> NSFetchRequest<MyEntity> {
        return NSFetchRequest<MyEntity>(entityName: "MyEntity")
    }
}

```



```
@NSManaged public var title: String?
}
```

Сам класс наследует `NSManagedObject` и имеет имя `MyEntity`, указанное в описании объекта. Однако это не обязательно: класс, реализующий объект модели, может иметь имя, отличающееся от имени объекта в описании. Также обратите внимание на спецификатор `@NSManaged` атрибута модели: он означает, что хранением этого свойства управляет Core Data. То есть это не обычное свойство, а хранимое. Фактически хранимые свойства поддерживают метод `setValue`, который вызывается при любой попытке присвоить значение свойству.

Такая гибкость позволяет добавлять в управляемые объекты свойства, которые не сохраняются в базе данных. По сути, это означает возможность определять вычисляемые свойства, играющие вспомогательную роль и не используемые для взаимодействия с базой данных. Например, можно определить управляемый объект с именем `Person` с атрибутами `firstName` и `lastName`, хранищимися в Core Data, и имеющий свойство `fullName`, которое просто объединяет атрибуты `firstName` и `lastName`, не требуя дублирования данных в базе данных:

```
public class Person: NSManagedObject {
    @NSManaged public var firstName: String
    @NSManaged public var lastName: String
    public var fullName: String {
        return "\(firstName) \(lastName)"
    }
}
```

Теперь перейдем к вопросу сохранения данных, используя в качестве примера наш управляемый объект `Person` (после создания соответствующего описания в модели управляемых объектов).

Запись хранимых данных в SQLite

Первое и очень важное, что следует сказать о записи данных в Core Data, – запись *нежелательно* выполнять в основном потоке выполнения. На самом деле в контейнере хранилища есть удобный метод, позволяющий легко и быстро делегировать запись данных фоновому потоку, как показано здесь:

```
persistentContainer.performBackgroundTask { (managedObjectContext) in
    // Некоторые операции...
}
```

Операциями в данном случае могут быть операции чтения или записи. Сейчас мы сосредоточимся на записи. Вернемся к нашему управляемому объекту `Person` и посмотрим, как создать новый экземпляр и записать его в хранилище.

При создании и изменении объектов, управляемых инфраструктурой Core Data, все операции сначала выполняются в контексте управляемого объекта. Мы можем изменять объекты в этом контексте как угодно и сколько угодно, и эти изменения никак не будут отражаться на базе данных, пока мы явно не сохраним изменения. То есть, чтобы создать новый экземпляр управляемого объекта, мы должны создать контекст. Вот как можно создать новый экземпляр

Person и присвоить ему некоторые свойства:

```
persistentContainer.performBackgroundTask { (managedObjectContext) in
    let person = Person(context: managedObjectContext)
    person.firstName = "Mike"
    person.lastName = "Dunn"
}
```

Здесь все просто. Мы создали новый экземпляр Person с именем person в контексте управляемого объекта, переданном в наше замыкание, а затем присвоили значения свойствам firstName и lastName.

Однако этот объект еще не был сохранен в базе данных. Фактически он будет существовать только до завершения замыкания, а затем уничтожится вместе с контекстом управляемого объекта. К счастью, сохранить данные тоже не сложно. Достаточно добавить одну строку в блок do и try и проверить ошибку, например:

```
persistentContainer.performBackgroundTask { (managedObjectContext) in
    let person = Person(context: managedObjectContext)
    person.firstName = "Mike"
    person.lastName = "Dunn"

    // Сохранить контекст
    do {
        try managedObjectContext.save()
    } catch {
        print("Error during save. \(error)")
    }
}
```

Заметили, что мы сохранили *контекст*, а не сам объект? Эта операция сохраняет все изменения, выполненные внутри контекста. То есть если бы мы создали миллион экземпляров Person, все они были бы сохранены одновременно. Если бы мы создали экземпляры объектов разных типов и изменяли свойства в некоторых других, все эти изменения и новые экземпляры также сохранились бы одновременно. Сохраняется контекст целиком, а не отдельные объекты. Это может привести к проблемам производительности. Поэтому для предотвращения подобных ситуаций рекомендуется проводить профилирование и тестирование с большими наборами данных.

Чтобы изменить объект, его нужно извлечь из Core Data в контексте управляемого объекта, изменить требуемые свойства и затем сохранить контекст. Давайте посмотрим, как извлечь хранимые данные из SQLite.

Чтение данных из SQLite

Этот раздел озаглавлен как «Чтение данных из SQLite», но на самом деле мы будем читать данные через контекст управляемого объекта. Если он *решил* обратиться к хранилищу SQLite, то обратится к координатору хранилища за помощью. Использование NSFetchRequest, как мы увидим ниже, приводит к отправке запроса в хранилище и получению результатов и может выполняться

ся очень медленно. Однако контексты управляемых объектов часто содержат много данных, кешируемых в памяти ради высокой производительности, и извлеченные данные могут использоваться и обрабатываться точно так же, как и обычные объекты.

Чтобы найти объект, нужно создать запрос на выборку и выполнить его в контексте управляемого объекта, например:

```
let managedObjectContext = persistentContainer.viewContext
let fetchRequest = NSFetchRequest<Person>(entityName: "Person")
do {
    let persons = try managedObjectContext.fetch(fetchRequest)
} catch {
    fatalError("Fetch could not be completed.")
}
```

Рассмотрим эту последовательность инструкций поближе.

Сначала, обратившись к свойству `viewContext` контейнера хранилища, мы получаем контекст управляемого объекта в режиме только для чтения. Этот режим позволяет просматривать объекты, но не сохранять изменения. Затем создаем запрос на выборку объекта типа `Person`, который в модели управляемых объектов имеет имя `Person`. После этого мы передаем запрос в вызов метода `fetch(_ :)` контекста. Этот вызов может сгенерировать исключение, поэтому он заключен в блоки `do` и `catch`. Если в процессе выборки возникнет ошибка, мы генерируем фатальную ошибку с сообщением «Fetch could not be completed» (Выборка не может быть выполнена).

После успешной выборки мы получим коллекцию `persons`, по которой сможем выполнить итерации и прочитать все экземпляры `Person`, сохраненные прежде.

Core Data обладает очень широкими возможностями. Это мощный инструмент, достойный находиться в арсенале каждого разработчика приложений для iOS. С его помощью можно создать высокопроизводительный граф объектов и упростить сохранение данных. К числу самых больших недостатков Core Data можно отнести некоторую избыточность кода и большое количество настроек. Однако усилия, вложенные на начальном этапе проекта, могут избавить вас от необходимости писать сложные и хрупкие операторы SQL и дадут взамен дополнительный контроль ошибок в именах и значениях свойств во время компиляции.



Для извлечения результатов из табличных представлений можно использовать объект `NSFetchedResultsController`, который получает результаты от `NSFetchRequest` и автоматически обновляет таблицу при каждом изменении данных. Это упрощает взаимодействие с Core Data из таких представлений, как `UITableView` и `UICollectionView`.

Что мы узнали

Android и iOS имеют много общего, как вы видели и еще не раз увидите в этой книге. Однако, как показала эта глава, иногда они не могут расходиться довольно далеко друг от друга.

- Низкоуровневый способ взаимодействий с SQLite в Android предлагает архитектурную простоту, которой нет в Core Data, что позволяет разработчикам использовать уже имеющиеся знания, приобретенные при работе с серверными языками.
- Core Data – это не база данных. Это сложный граф объектов, *чрезвычайно* мощный и отделенный от SQLite с целью замаскировать сложности работы с базами данных напрямую.
- Оба подхода позволяют сохранять данные и требуют тщательного планирования и поддержки в будущих версиях приложений.

Следующим логическим шагом после разговоров о хранении объектов является обсуждение темы конкурентного (параллельного) выполнения, которой посвящена глава 8. Приступим прямо сейчас!

Глава 8

Конкурентное (многопоточное) выполнение

Под конкурентным выполнением (иногда его называют параллельным, или многопоточным, выполнением) подразумевается одновременное выполнение нескольких задач, делящих вычислительные ресурсы и очень быстро чередующихся друг с другом, пока какая-либо не завершит свою работу и не будет исключена из этого процесса. Акт передачи вычислительных ресурсов между различными задачами называют «переключением контекста». В информатике этот термин имеет более общее и сильно отличающееся определение (переключение ресурсов вашего мозга с одной мысли на другую).

Задачи

В этой главе вы узнаете:

- 1) как запустить задачу в фоновом потоке выполнения;
- 2) как передать результаты выполнения задачи из фонового потока в главный.

ANDROID

В Java вычислительный контекст представляет экземпляр класса `Thread`, который создается в точности, как вы могли бы ожидать:

Java

```
Thread thread = new Thread(...);
```

Kotlin

```
val thread = Thread(...)
```

В конструктор потока выполнения `Thread` передается экземпляр `Runnable`. После запуска потока (вызовом метода `start`) он вызывает метод `run` получен-

ного экземпляра `Runnable`. И когда этот метод завершится (вернув управление или возбуждив исключение), поток выполнения остановится.

Приложение для Android всегда имеет по меньшей мере один экземпляр `Thread`, в котором по умолчанию выполняется вся работа, включая, в частности, обслуживание пользовательского интерфейса. Это важно – на самом деле этот поток иногда называют «поток пользовательского интерфейса», но чаще «основным», или «главным», потоком. Со времени появления первой версии Android и до момента написания этой книги основной поток рисовал пиксели на экране 60 раз в секунду – примерно раз в 16 миллисекунд – и назывался «кадром». Это чрезвычайно ресурсоемкий процесс, поэтому очень важно отдать этому потоку как можно больше вычислительных ресурсов для выполнения данной работы. По этой причине тяжелые, с вычислительной точки зрения, задачи должны выполняться «вне» основного потока (то есть в любом другом потоке, который *не* отвечает за пользовательский интерфейс, *не* является основным потоком программы и иногда называется «фоновым потоком»). Любые операции ввода/вывода, такие как доступ к файловой системе, транзакции в базе данных или продолжительные операции, например сетевые запросы, обязательно должны выполняться не из основного потока. Фактически вы увидите ошибки или предупреждения при попытке выполнить подобные операции из основного потока. Кроме того, когда основной поток выполняет слишком много работы, пользовательский интерфейс будет обновляться «рывками». Например, `RecyclerView` может прокручиваться скачками или экран будет зависать на некоторое время. Android Studio достаточно интеллектуальна, чтобы предупредить нас о подозрительных действиях; и вы часто будете видеть такие сообщения:

```
I/Choreographer(1234): Skipped 20 frames! The application may be doing too much
work on its main thread.
```

В данном случае сообщается, что система *двадцать раз* пыталась перерисовать пользовательский интерфейс, но не смогла сделать это, потому что была лишена необходимых ресурсов.

Запуск задачи в фоновом потоке

Так как же выполнить работу вне основного потока? Делается это довольно просто. Как мы уже говорили, нужно создать новый экземпляр `Thread`, выполнить некоторую работу в методе `run` экземпляра `Runnable`, переданного конструктору, и запустить поток вызовом `start`:

Java

```
new Thread(() -> Log.d("MyTag", "This log is from a background thread")).start();
```

Kotlin

```
Thread({Log.d("MyTag", "This log is from a background thread")}).start()
```

Вот и все! Существует также ряд вспомогательных классов, упрощающих управление потоками, в том числе `ThreadPoolExecutor` и `Executors` (последний содержит ряд статических функций, которые обычно возвращают готовые к ис-

пользованию экземпляры `ThreadPoolExecutor`). Использование пула потоков позволяет контролировать объем работы, выполняемой в фоновом режиме, а также ставить задачи в очередь и удаляет их по мере выполнения. Загляните в документацию с описанием `ThreadPoolExecutor` (<https://oreil.ly/9pV5G>) и `Executors` (<https://oreil.ly/WKIRY>).

Возможно, вы слышали, что конкурентное выполнение – это один из самых сложных аспектов в информатике, и мы не будем притворяться, что в нем нет ничего чрезвычайно сложного, но большинство из этих сложностей спрятаны от нас за фасадом системы или легко обходятся стороной. Как-то Майк (один из авторов книги) сказал, что «многопоточность реализуется просто; сложнее реализовать синхронизацию». То есть, как было показано выше, мы легко можем запустить некоторую задачу в фоновом потоке, но за этим утверждением следует 3500 звездочек (извините за гиперболу). Например, в Android нельзя получить доступ ни к какому экземпляру `View`, кроме как из основного потока. Это действительно создает большие сложности! В методе `Runnable.run` нельзя даже *ссылаться* на экземпляр `View` без риска получить `RuntimeException`. Кроме того, порядок завершения фоновых потоков никак не гарантируется; и здесь мы начинаем вдаваться в сложности конкурентного выполнения – байт-код, который генерирует компилятор Java, не всегда выглядит так же, как исходный код на Java.

Вот классический пример «небезопасного» потока выполнения:

Java

```
public class MainActivity extends Activity {
    private static final int MAX = 1000;
    private int mCounter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main); // имеется представление с кнопкой
        findViewById(R.id.button).setOnClickListener(view -> {
            while (mCounter < MAX) {
                new Thread(() -> {
                    Log.d("MyTag", String.valueOf(mCounter++));
                }).start();
            }
        });
    }
}
```

Kotlin

```
class MainActivity : Activity() {
    private const val MAX = 1000
    private var counter: Int = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main) // имеется представление с кнопкой
```

```

findViewById(R.id.button).setOnClickListener {
    while (counter < MAX) {
        Thread { Log.d("MyTag", counter++.toString()) }.start()
    }
}
}
}
}
}

```

В зависимости от особенностей устройства и окружения может потребоваться запустить больше или меньше параллельных задач, чтобы увидеть эффект, но, поэкспериментировав с этой простой задачей, вы рано или поздно увидите, что окончательное значение в журнале *не равно* 9999. Но не будем тратить ваше и наше время на объяснение нюансов безопасности многопоточного выполнения и сразу скажем, что байт-код оператора инкремента выглядит примерно так:

- 1) получить значение `mCounter` из памяти;
- 2) прибавить 1 к значению `mCounter`;
- 3) записать результат обратно в память.

Поскольку система быстро-быстро переключает параллельные потоки, чтобы сымитировать параллельное выполнение операций (это называется «переключением контекста»), поток № 391 может прочитать значение `mCounter` в тот момент, когда поток № 390 уже выполнил шаг 1, но еще не выполнил шаг 2. Эта конкретная ситуация называется состоянием гонки. Данная проблема широко известна как взаимовлияние потоков и является одним из примеров недостаточной безопасности при многопоточном выполнении. Это далеко неполное исследование рисков конкурентного программирования – некоторые реализации Java позволяют потокам создавать копии переменных, чтобы избежать дорогостоящих вызовов IPC (Interprocess Communication – межпроцессные взаимодействия), поэтому, когда несколько потоков изменяют один и тот же экземпляр объекта, его состояние не может быть гарантировано.

Для борьбы с этим эффектом существуют специальные механизмы, такие как ключевые слова `synchronized` и `volatile` и классы `Atomic`. И поскольку структуры данных особенно уязвимы (представьте, что один поток выполняет обход содержимого структуры данных, а другой добавляет или удаляет элементы к ней), некоторые структуры имеют потокобезопасные (и менее производительные) версии или вспомогательные методы, помогающие уменьшить риски.

Более глубокое обсуждение проблем конкурентного выполнения выходит за рамки этой главы. Могут потребоваться годы, чтобы освоить многопоточное программирование даже в одном технологическом стеке. Некоторые разработчики никогда по-настоящему не понимают, что происходит за кулисами, и это нормально! Потрясающий разработчик пользовательского интерфейса может использовать совершенно иные приемы конкурентного программирования, чем программист, работающий с большими данными. Один простой трюк в Android заключается в том, чтобы запустить вычисления в фоновом потоке, а затем вызвать `Activity.runOnUiThread`, `View.post` или `Handler.post`, дабы передать результаты в основной поток, чтобы гарантировать последовательное выполнение и невосприимчивость к действиям других процессов.

Передача результатов из фонового потока в главный

Как уже упоминалось, чтобы обновить пользовательский интерфейс по окончании вычислений в фоновом потоке (например, после получения информации из локальной базы данных, от удаленного сервера, из файловой системы и т. д.), вы должны вернуться в основной поток и только потом пытаться ссылаться на экземпляры View. Более того, чтобы избежать проблем синхронизации, описанных выше, не только представления, но и любые другие общие объекты полезно обновлять в одном (основном) потоке.

Существует три основных способа отправки сообщений из фонового потока в основной:

- 1) метод `View.post` принимает экземпляр `Runnable` и вызывает метод `run` этого экземпляра в основном потоке;
- 2) `Activity.runOnUiThread` работает почти так же, как `View.post`, но сначала проверяет, выполняется ли вызов в основном потоке – в этом случае метод `run` вызывается немедленно, а не отправляется в конец очереди сообщений (см. пример ниже);
- 3) экземпляр `Handler` передает сообщение экземпляру `Looper`, с которым он связан, то есть экземпляр `Handler` создан вызовом `new Handler(Looper.getMainLooper())`, он будет отправлять сообщения в основной поток. Метод `Handler.post` точно так же принимает экземпляр `Runnable` и вызывает его метод `run`. Обратите внимание, что обе предыдущие операции фактически выполняются тут же, если только `Activity.runOnUiThread` не вызывается непосредственно из потока пользовательского интерфейса.

Вот простой пример:

Java

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main); // имеется представление с кнопкой
        Button button = findViewById(R.id.button);
        button.setOnClickListener(view -> {
            button.setText("Sleeping");
            new Thread(() -> {
                try {
                    Thread.sleep(2000);
                    button.post(() -> button.setText("Awake"));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        });
    }
}
```

Kotlin

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

```

super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main) // имеется представление с кнопкой
button.setOnClickListener { view ->
    button.text = "Sleeping"
    Thread {
        try {
            Thread.sleep(2000)
            button.post { button.text = "Awake" }
        } catch (e: InterruptedException) {
            // ничего не делать
        }
    }.start()
}
}
}
}

```

Завершение потока выполнения

Следующая тема, которую мы обсудим, тоже полна противоречий. Обычно поток выполнения завершается, когда закончит выполнять свою работу, – когда завершается метод `run`. Однако часто желательно преждевременно остановить фоновый поток. Например, начав загружать большой видеоклип с размером в несколько сотен мегабайтов, пользователь, осознав через пару минут, что загрузка продлится слишком долго или не хватит места в памяти устройства, может решить прекратить загрузку.

Если вы запустили этот процесс в собственном потоке выполнения и использовали стороннюю библиотеку с атомарным методом `download`, то не сможете остановить его:

Java

```

// допустим, что downloader -- это уже настроенный экземпляр стороннего
// класса Downloader
Downloader downloader = new Downloader(url);
new Thread(downloader::download).start();

```

Kotlin

```

// допустим, что downloader -- это уже настроенный экземпляр стороннего
// класса Downloader
val downloader = Downloader(url);
Thread(Runnable { downloader.download() }).start()

```

В классе `Thread` имеется устаревший метод `stop`, но мы не рекомендуем пользоваться им, потому что в будущем он может быть удален из стандартной библиотеки.

Изучающих `Thread` API часто привлекает метод `join`, но нередко они понимают его неправильно. В документации к Java 8 метод `join` описывается так:

Эта реализация в цикле вызывает метод `this.wait`, опирающийся на условие `this.isAlive`. Когда поток завершает работу, вызывается метод `this.notifyAll`. Рекомендуется, чтобы приложения не использовали методы `wait`, `notify` или `notifyAll` экземпляров `Thread`.

Может *показаться*, что поток завершает выполнение, особенно это относится к «основному» потоку (потоку пользовательского интерфейса), но в действительности он все еще находится в памяти и продолжает удерживать ссылки на объекты, пока не завершится поток, вызвавший метод. То есть если в Android, где поток пользовательского интерфейса служит контекстом по умолчанию, запустить «фоновый» поток и «присоединиться» к нему вызовом метода `join`, он будет продолжать занимать память, пока не завершится основной поток (когда завершится само приложение или система решит отобрать ресурсы у простаивающего приложения, чтобы передать их другим программам). Поэтому вы можете даже не заметить фоновый поток, но знайте, что он *продолжает* существовать и удерживать ссылки (вызывая утечки памяти) до завершения самой программы.

В классе `Thread` есть еще один, суть которого тоже часто понимается неправильно, – метод `interrupt`. Этот метод делает совсем не то, что многие думают, – он просто устанавливает логический флаг в потоке. До вызова `interrupt` обращения к `isInterrupted` будут возвращать `false`, а после вызова `interrupt` – `true`. Сам по себе метод `interrupt` ничего не делает, чтобы остановить выполнение потока.

Этот API был создан почти как соглашение; авторы библиотек должны периодически проверять флаг `isInterrupted`, и если он имеет значение `true`, генерировать исключение или немедленно завершать выполнение потока, *но это поведение никак не гарантируется*. Мы можем надеяться, что авторы библиотеки `Downloader` реализовали это соглашение, но не можем быть в этом уверены, а это значит, что подобные задачи мы должны решать самостоятельно. И снова используем в качестве примера загрузку файла: вы можете проверять `isInterrupted` в каждой итерации цикла `InputStream.read/OutputStream.write`. Например, обратимся к простому методу загрузки в классе `Networking`, описанному в следующей главе, и изменим его, добавив возможность отмены:

Java

```
public class Networking {
    public static void downloadBinaryData(Context context) throws Exception {
        File file = new File(context.getFilesDir(), "downloaded-image.png");
        FileOutputStream fileOutputStream = new FileOutputStream(file);
        URL url = new URL("http://moagrius.com/assets/images/hero-trips.png");
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        int data;
        while ((data = connection.getInputStream().read()) != -1) {
            // сгенерировать исключение при прерывании операции загрузки
            if (Thread.currentThread().isInterrupted()) {
                throw new InterruptedException("Download cancelled!");
            }
            fileOutputStream.write(data);
        }
        fileOutputStream.flush();
    }
}
```

Kotlin

```

fun downloadBinaryData(context: Context) {
    val file = File(context.filesDir, "downloaded-image.png")
    val fileOutputStream = FileOutputStream(file)
    val url = URL("http://moagrius.com/assets/images/hero-trips.png")
    val connection = url.openConnection() as HttpURLConnection
    var data = connection.inputStream.read()
    while (data != -1) {
        // сгенерировать исключение при прерывании операции загрузки
        if (Thread.currentThread().isInterrupted) {
            throw InterruptedException("Download cancelled!")
        }
        fileOutputStream.write(data)
        data = connection.inputStream.read()
    }
    fileOutputStream.flush()
}

```

В блоке `try/catch`, где вызывается `downloadBinaryData`, вы, вероятно, захотите удалить частично загруженный файл и обновить пользовательский интерфейс, чтобы скрыть компонент, отображающий прогресс загрузки, или показать сообщение пользователю. Возможно, вы также захотите вызвать метод `cancel`, который, в свою очередь, вызовет `Thread.interrupt()` фонового потока. Вот как это могло бы выглядеть в коде (с учетом особенностей Android):

Java

```

public class MainActivity extends Activity {

    private Thread mBackgroundThread;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main); // имеется представление с кнопкой
        Button button = findViewById(R.id.button);
        button.setOnClickListener(this::clickHandler);
    }

    public void clickHandler(View view) {
        if (mBackgroundThread != null) {
            cancel();
        } else {
            downloadInBackground();
        }
    }

    public void downloadInBackground() {
        mBackgroundThread = new Thread(() -> {
            try {
                downloadBinaryData();
            } catch (Exception e) {
                Log.d("MyTag", "cancelled!");
            } finally {

```

```

        mBackgroundThread = null;
    }
});
mBackgroundThread.start();
}

public void cancel() {
    mBackgroundThread.interrupt();
}

private boolean assetDownloadIsNotCancelled() throws InterruptedException {
    if (mBackgroundThread.isInterrupted()) {
        throw new InterruptedException("Download cancelled!");
    }
    return true;
}

public void downloadBinaryData() throws Exception {
    File file = new File(getFilesDir(), "downloaded-image.png");
    if (file.exists()) {
        file.delete();
    }
    FileOutputStream fileOutputStream = new FileOutputStream(file);
    assetDownloadIsNotCancelled();
    URL url = new URL("http://moagrius.com/assets/images/hero-trips.png");
    assetDownloadIsNotCancelled();
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    assetDownloadIsNotCancelled();
    InputStream inputStream = connection.getInputStream();
    byte[] buffer = new byte[1024];
    try {
        while (assetDownloadIsNotCancelled()) {
            int data = inputStream.read(buffer);
            if (data == -1) {
                break; // EOF
            }
            fileOutputStream.write(buffer, 0, data);
            fileOutputStream.flush();
        }
    } finally {
        fileOutputStream.close();
        inputStream.close();
    }
}
}
}
}

```

Kotlin

```

class MainActivity : Activity() {
    private var backgroundThread: Thread? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main) // имеется представление с кнопкой
    }
}

```

```
    val button = findViewById<Button>(R.id.browse_button)
    button.setOnClickListener(::clickHandler)
}

fun clickHandler(view: View) {
    if (backgroundThread != null) {
        cancel()
    } else {
        downloadInBackground()
    }
}

fun downloadInBackground() {
    backgroundThread = Thread {
        try {
            downloadBinaryData()
        } catch (e: Exception) {
            Log.d("MyTag", "cancelled!")
        } finally {
            backgroundThread = null
        }
    }
    backgroundThread?.start()
}

fun cancel() {
    backgroundThread?.interrupt()
}

@Throws(InterruptedOperationException::class)
private fun assetDownloadIsNotCancelled(): Boolean {
    if (backgroundThread?.isInterrupted) {
        throw InterruptedException("Download cancelled!")
    }
    return true
}

@Throws(Exception::class)
fun downloadBinaryData() {
    val file = File(filesDir, "downloaded-image.png")
    if (file.exists()) {
        file.delete()
    }
    val fileOutputStream = FileOutputStream(file)
    assetDownloadIsNotCancelled()
    val url = URL("http://moagrius.com/assets/images/hero-trips.png")
    assetDownloadIsNotCancelled()
    val connection = url.openConnection() as HttpURLConnection
    assetDownloadIsNotCancelled()
    val inputStream = connection.inputStream
    val buffer = ByteArray(1024)
    try {
        while (assetDownloadIsNotCancelled()) {
            val data = inputStream.read(buffer)
            if (data == -1) {
```

```
        break // EOF
    }
    fileOutputStream.write(buffer, 0, data)
    fileOutputStream.flush()
}
} finally {
    fileOutputStream.close()
    inputStream.close()
}
}
```

Библиотека RxJava (<https://oreil.ly/HyJnl>) предлагает надежный API для параллельного выполнения операций со слоями абстракции, позволяющими отделить разрабатываемый код от опасностей низкоуровневого управления потоками. Она может оказаться хорошим инструментом для создания общей базы кода и отлично подходит для решения многих задач. Тем не менее, как это характерно для большинства абстракций, классы из стандартной библиотеки, такие как Thread, могут обеспечить более точное управление выполнением.

Класс AsyncTask в Android позволяет запускать фоновые задания и посылать результаты в основной поток. Но имейте в виду, что этот класс имеет свои недостатки. С одной стороны, в самых ранних версиях этот класс использовал несколько потоков из пула ThreadPoolExecutor, позднее он был изменен и стал использовать один поток, но затем снова был изменен. Кроме того, рабочий метод AsyncTask не имеет возможности выхода, если вдруг потребуется отменить задачу. В последнее время сообщество пришло к общему мнению, что AsyncTask не представляет достаточной ценности, чтобы оправдать недостатки, – авторы книги считают, что почти во всех случаях более простым, надежным и управляемым является подход на основе Thread и post, описанный выше.

Кроме того, многие из тем, затронутых в этой главе, очень подробно освещены в книге Андерса Ёранссона (Anders Göransson) «Efficient Android Threading» (O'Reilly)¹. И конечно же, по многим вопросам вам поможет книга Джошуа Блоха (Joshua Bloch) «Efficient Java» (O'Reilly)².

iOS

Подходы к организации параллельных вычислений в iOS основываются на применении старых классических приемов, использовавшихся еще в языке C, и использовании модернизированного и интеллектуального механизма Grand Central Dispatch. В iOS доступны три основных варианта выполнения заданий в фоновом режиме: DispatchQueue, Operation и Thread.

Иногда можно использовать оба подхода сразу, но для целей этой главы мы сосредоточимся на Grand Central Dispatch и DispatchQueue; они применяются

¹ Ёранссон Андерс. Эффективное использование потоков в операционной системе Android. ДМК Пресс, 2017. ISBN 978-5-97060-182-2. – Прим. перев.

² Джошуа Блох. Java. Эффективное программирование. Лори, 2014. ISBN 978-5-85582-347-9. – Прим. перев.

чаще всего, даже несмотря на наличие более высокоуровневых инструментов. Это связано с простотой и легкостью их создания.

Запуск задачи в фоновом потоке

Итак, как в iOS запустить задачу в фоновом потоке? Вот простой пример:

```
DispatchQueue.global().async {
    print("Do something")
}
```

Первая строка обращается к глобальной очереди фоновых заданий, которая является частью механизма Grand Central Dispatch, или «GCD». Мы получаем ссылку на глобальную очередь и вызываем ее метод `async`, передавая свое замыкание. Вторая строка – это тело замыкания, которое просто выводит строку текста.

В GCD поддерживаются два варианта выполнения операций: синхронный и асинхронный. В синхронном режиме операции выполняются с помощью метода `sync()` – они гарантированно будут выполнены немедленно. В асинхронном режиме (с помощью метода `async()`) операции выполняются в какой-то момент в будущем, который невозможно предсказать точно.

Честно говоря, предыдущего кода достаточно для большинства случаев, но иногда желательно иметь возможность более полного контроля над приоритетом конкретного фонового задания. Это можно организовать, как показано ниже:

```
DispatchQueue.global(qos: .userInitiated).async {
    print("Do something")
}
```

Здесь в вызов `global()` передается новый аргумент `qos`. Это перечисление, определяющее набор приоритетов выполнения в iOS. Доступны следующие варианты:

`userInteractive`

Наивысший приоритет, используемый для интерактивных операций. Если с этим приоритетом запустить медленную операцию, это приведет к притормаживанию пользовательского интерфейса.

`userInitiated`

Немного более низкий приоритет и чаще используемый для фоновых заданий, чем `userInteractive`. Обычно применяется для операций, запущенных пользователем, результаты которых должны быть получены как можно быстрее.

`default`

Приоритет по умолчанию. Используется для заданий с негарантированным качеством обслуживания (Quality of Service, QoS). Должен указываться разработчиками напрямую.

`utility`

Еще более низкий приоритет. Предназначен для выполнения вспомогательных операций, таких как загрузка книги или видео, в приложениях переднего плана.

background

Самый низкий приоритет. Используется для выполнения заданий, скрытых от пользователя.

unspecified

Предназначен для системных нужд и определения уровня качества обслуживания (QoS). Вы не должны использовать его.

i Эти уровни качества обслуживания также соответствуют уровням производительности и энергоэффективности. Задания на уровне `userInteractive` выполняются быстрее, но с меньшей энергоэффективностью. Задания на уровне `background` выполняются медленнее, но с меньшими затратами электроэнергии.

В предыдущем примере мы передали значение `userInitiated`, указав, что операция должна выполняться вне основного потока в глобальной очереди с довольно высоким приоритетом.

Правила качества обслуживания – это один из немногих способов управления глобальной очередью заданий. Кроме него, есть еще один способ:

```
let queue = DispatchQueue(
    label: "com.oreilly.nativeappdevelopment", qos: .background, attributes: [.concurrent])
queue.async {
    print("Do something")
}
```

В этом примере мы создали свой экземпляр `DispatchQueue`. Каждая очередь должна иметь уникальную метку. Apple рекомендует использовать метки в стиле обратного DNS, но вообще достаточно, чтобы метка была уникальной. В нашем примере мы передали в параметре `qos` значение `background`, а это означает, что задания в этой очереди будут выполняться с самым низким приоритетом.

Кроме того, мы передали значение `.concurrent` в параметре `attributes`. Это означает, что все операции – в данном случае замыкание – в очереди будут выполняться одновременно. В глобальной очереди этот параметр по умолчанию имеет значение `.concurrent`. Для пользовательских очередей, напротив, это значение *не* является значением по умолчанию, то есть задания в них по умолчанию выполняются последовательно.

Мы сохраняем ссылку на эту новую очередь в переменной `queue`. Следующая инструкция обращается к этому объекту очереди и точно так же, как в предыдущих примерах, вызывает его метод `async(_:)`.

Передача результатов из фонового потока в главный

Теперь вы знаете, как запустить операцию в фоновом потоке. Но иногда желательно передать какие-то данные в основной поток, чтобы выполнить некоторую важную работу. В iOS в этом часто возникает необходимость, когда требуется обновить пользовательский интерфейс, потому что все манипуляции с пользовательским интерфейсом выполняются только в основном потоке.

Реализуется это довольно просто:

```
DispatchQueue.main.async {
    // Обновить пользовательский интерфейс
}
```

Обратите внимание на свойство `main` класса `DispatchQueue`. Оно представляет общую очередь, в которой действуют все объекты пользовательского интерфейса. Любые обновления пользовательского интерфейса должны выполняться в таком вызове. Например, после выполнения сетевого вызова с использованием `URLSessionDataTask` может потребоваться отобразить некоторую информацию в пользовательском интерфейсе. Эту задачу решает следующий код:

```
let url = URL(string: "https://www.example.com")!
let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
    // Обработать ошибки и пр.

    guard let data = data, let string = String(data: data, encoding: .utf8) else {
        print("No data returned.")
        return
    }

    DispatchQueue.main.async {
        // Обновить пользовательский интерфейс
    }
}
task.resume()
```

Не углубляясь в особенности работы `URLSession` в iOS (подробнее об этом рассказывается в главе 9), обратите внимание на вызов `DispatchQueue.main.async(_:)` в теле обработчика успешного завершения операции. Его можно использовать для изменения текста метки, других компонентов пользовательского интерфейса или просто для перехода в основной поток с целью синхронизации данных.

i Отмена операции GCD – сложная и трудоемкая задача. Если вам может потребоваться отменять операции, используйте объекты `Operation` с `OperationQueue`. Однако обсуждение этой темы выходит за рамки данной главы, но всю необходимую информацию вы сможете найти в документации Apple для разработчиков с описанием `Operation` и `OperationQueue`.

Что мы узнали

Организация конкурентного выполнения операций, если речь не идет о паре простых инструкций, может превратиться в весьма сложную задачу. И в конкурентном программировании есть целые классы ошибок, свойственные только ему.

- В Android имеется несколько методов запуска параллельных операций, но на практике чаще применяется класс `Thread` и ряд стандартных подходов к управлению выполнением операций.

- GCD предлагает удобную библиотеку планирования, с помощью которой приложения для iOS (и macOS) могут выполнять фоновые задания.
- В обеих системах, Android и iOS, важно гарантировать, что основной поток не будет блокироваться продолжительными операциями, иначе пользовательский интерфейс будет работать с задержками или выглядеть зависшим, что может вызвать у пользователя отрицательные эмоции.

Мы также увидели, в каких случаях фоновые потоки чрезвычайно полезны. Примером могут служить сетевые запросы. Теперь, зная, как запускать операции в фоновом режиме, займемся исследованием приемов выполнения сетевых операций в Android и iOS.

Глава 9

Сетевые взаимодействия

Безусловно, можно написать практически полностью автономное приложение, которое отправляет HTTP-запросы только для обновления себя с использованием встроенной, автоматизированной инфраструктуры обновления Play, но такое трудно себе представить в современном мире. Независимо от наличия или отсутствия необходимости аутентифицировать пользователей, публиковать параметры использования для анализа, читать контент из сети, загружать или выгружать файлы, получать push-уведомления или просто синхронизироваться с сервером времени, вам наверняка пригодятся знания и умения, позволяющие создавать HTTP-запросы и получать HTTP-ответы.

К счастью, спецификация HTTP определяет очень простой набор правил, понять и освоить которые сможет любой из вас. В простейшем случае можно считать, что HTTP-транзакция состоит из двух частей: запроса и ответа. И тот, и другой можно представить себе в виде простых (часто легко читаемых человеком) текстовых фрагментов. Первая строка в запросе и ответе описывает некоторые основные атрибуты (например, URL-адрес запроса и статус ответа). Следующие строки содержат заголовки в традиционном синтаксисе пар ключ/значение: имя_заголовка: значение_заголовка. Далее следует пустая строка и тело запроса или ответа. Вот и все! За дополнительной информацией обращайтесь к статье «HTTP Made Really Easy» (<https://oreil.ly/Qh4ZZ>).

От высокоуровневых протоколов аутентификации до потоковой передачи зашифрованных видеоданных и загрузки изображений на телефон в фоновом режиме – все эти операции используют те же базовые принципы, что описаны выше.

Задачи

В этой главе вы узнаете:

- 1) как загрузить текстовый файл с удаленного сервера и вывести его;
- 2) как создать запрос HTTP POST;
- 3) как загрузить двоичный файл.

ANDROID

Благодаря фантастически открытой природе Android появилось множество потрясающих сторонних библиотек, помогающих решать по-настоящему сложные задачи, к которым относятся и сетевые взаимодействия. Наиболее популярной из них, пожалуй, является библиотека OkHttp, разработанная известным разработчиком Android по имени Джейк Уортон (Jake Wharton), который написал массу великолепного программного обеспечения с открытым исходным кодом, работая в компании Square. В настоящее время он продолжает вносить свой вклад, работая в Google. Взглянув на список в конце этого раздела, посвященного Android, вы увидите, что подавляющее большинство ссылок введет в учетную запись Square на GitHub.

Не забывайте, что всякий раз, имея дело с чем-то, кроме оперативной памяти (ОЗУ), соответствующие операции следует выполнять в фоновом потоке. Чтение локального файла, отправка запроса к базе данных и даже сохранение настроек должны производиться вне основного потока, обслуживающего пользовательский интерфейс. *Это особенно важно при выполнении сетевых взаимодействий.* Представьте, что имеется небольшой файл для его открытия, чтения, закрытия и преобразования содержимого в значение String и требуется всего несколько миллисекунд, но даже эти несколько миллисекунд могут вызвать «рывки» в работе RecyclerView или NavigationDrawer. А теперь представьте, что вместо обращения к локальному файлу вы посылаете запрос в сеть. Уверен, что всем вам приходилось сталкиваться с запросами, которые обрабатываются по 10 или 20 секунд и даже дольше! Только подумайте, что получится, если такой длительный запрос заблокирует поток пользовательского интерфейса в ожидании ответа. *При выполнении сетевых операций вся работа, связанная с ними, почти во всех случаях должна выполняться в фоновом потоке.*

Загрузка текстового файла с удаленного сервера и его вывод

А теперь перейдем к практике. На самом деле послать сетевой запрос из Java не так уж и сложно – вот шесть строк (если считать только инструкции), которые загрузят и выведут на экран любой доступный удаленный файл:

Java

```
public class Networking {
    public static void printRemoteFile() throws IOException {
        URL url = new URL("http://moagrius.com");
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        int data;
        while ((data = connection.getInputStream().read()) != -1) {
            System.out.print((char) data);
        }
    }
}
```

Kotlin

```
@Throws(IOException::class)
fun printRemoteFile() {
    val url = URL("http://moagrius.com")
    val connection = url.openConnection() as HttpURLConnection
    var data: Int = connection.inputStream.read();
    while (data != -1) {
        print(data.toChar())
        data = connection.inputStream.read()
    }
}
```

Рассмотрим этот код строку за строкой.

Прежде всего нужно создать экземпляр URL. Он представляет *ресурс*, а не *местоположение* этого ресурса, несмотря на то что аббревиатура «URL» расшифровывается как *universal resource locator* – универсальный указатель на ресурс. Местоположение ресурса определяется строкой, которая передается в конструктор URL.

Преобразовав String в URL, мы получаем из стандартной библиотеки массу возможностей, например вызов метода `openConnection` возвращает экземпляр соединения `URLConnection`. С этим соединением много чего можно сделать, и самое простое – взять входной поток `InputStream` и прочитать его, как показано в следующих нескольких строках кода! Этот код напоминает тот, что мы использовали в главе 6 для работы с потоковыми данными (загляните туда, чтобы освежить воспоминания).

Вот и все! Просто, не правда ли? А теперь добавим еще несколько мазков в картину.

Создание запроса HTTP POST

В этом разделе мы используем общедоступную бесплатную службу *jsonplaceholder.typicode.com*, которой будем посылать данные в запросах POST. Это поможет нам убедиться, что наш код действительно отправляет данные. В частности, мы будем использовать конечную точку создания ресурса (<https://oreil.ly/2iZxz>).

Изменим предыдущий пример, добавив отправку пустого запроса POST:

Java

```
public class Networking {
    public static void saveRemoteData() throws IOException {
        URL url = new URL("https://jsonplaceholder.typicode.com/posts");
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("POST");
        int data;
        while ((data = connection.getInputStream().read()) != -1) {
            System.out.print((char) data);
        }
    }
}
```

Kotlin

```
@Throws(IOException::class)
fun saveRemoteData() {
    val url = URL("https://jsonplaceholder.typicode.com/posts")
    val connection = url.openConnection() as HttpURLConnection
    connection.requestMethod = "POST"
    var data: Int = connection.inputStream.read()
    while (data != -1) {
        print(data.toChar())
        data = connection.inputStream.read()
    }
}
```

Приглядевшись, можно заметить только два небольших изменения:

- 1) мы привели значение, возвращаемое вызовом `openConnection`, к типу `HttpURLConnection` вместо `URLConnection`, потому что в адресе используется схема HTTPS;
- 2) установили метод запроса POST вместо используемого по умолчанию GET.

Этот код исправно работает, и в результате вы должны увидеть идентификатор вновь созданного вами объекта, но, кроме этого, ничего больше не происходит. На практике часто желательно отправить некоторые данные в запросе. Для этой цели можно использовать параметры, заголовки или тело запроса. Взаимодействуя с данной службой, мы будем посылать код JSON в теле запроса.

Объект соединения имеет не только встроенный поток ввода `InputStream`, но также поток вывода `OutputStream`. Однако для работы с ним требуется приложить немного дополнительных усилий – нужно вызвать метод `connection.setDoOutput(true)`, чтобы получить возможность вывода в этот поток.

Снова изменим пример:

Java

```
public class Networking {
    public static void saveRemoteData() throws IOException {
        URL url = new URL("https://jsonplaceholder.typicode.com/posts");
        HttpURLConnection connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("POST");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
        String json = "{title:'foo', body:'bar', userId:1}";
        connection.getOutputStream().write(json.getBytes());
        int data;
        while ((data = connection.getInputStream().read()) != -1) {
            System.out.print((char) data);
        }
    }
}
```

Kotlin

```
@Throws(IOException::class)
fun saveRemoteData() {
```

```

val url = URL("https://jsonplaceholder.typicode.com/posts")
val connection = url.openConnection() as HttpURLConnection
connection.requestMethod = "POST"
connection.doOutput = true
connection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded")
val json = "{\"title:'foo', body:'bar', userId:1}"
connection.outputStream.write(json.toByteArray())
var data: Int = connection.inputStream.read();
while (data != -1) {
    print(data.toChar())
    data = connection.inputStream.read()
}
}
}

```

Теперь мы отправляем некоторые данные, которые сервер может обработать по своему усмотрению. Как видите, для получения новых возможностей каждый раз приходится прилагать чуть больше усилий и уделять чуть больше внимания, но если начать с самых основ (первый пример), можно получить хорошее представление о том, как все это работает.

Работая с сетевыми библиотеками, вы быстро заметите, что по большей части приходится выполнять одни и те же действия, даже если за кулисами библиотеки используют совершенно разные операции. Разрабатывая первую версию загрузчика двоичных файлов для своего текущего работодателя, мы использовали разные библиотеки. Позднее мы решили выполнять все сетевые запросы (для получения изображений, обращений к REST API, загрузки файлов и т. д.) с помощью единственного HTTP-клиента, поэтому я заварил кофе и решил посвятить день переделке нашего загрузчика для работы с OkHttp. Я не помню, сколько времени это заняло или сколько строк изменилось, но скажу точно, что к обеду все, включая тесты, было готово (читай: это оказалось простой задачей).

Загрузка двоичного файла

Теперь вспомним, о чем рассказывалось в главе 6, и реализуем загрузку двоичного файла. Не волнуйтесь, эта задача мало отличается от той, что решили выше.

Вот URL изображения, которое я использую на своем личном сайте: <http://moagrius.com/assets/images/hero-trips.png>. Это всего лишь изображение, но это двоичные данные – ту же логику вы сможете применить для загрузки видео или исполняемого файла и даже для загрузки произвольных двоичных данных.

Давайте подумаем, что нам нужно сделать. Мы знаем, что при чтении из `InputStream` будем получать байты, составляющие изображение, но что с ними делать? Вспомните, что мы делали в главе 6, рассказывающей о файлах: нам понадобится `FileOutputStream`, чтобы сохранить эти байты на нашем локальном устройстве. Фактически это единственное отличие! Попробуем воплотить его:

Java

```

public class Networking {
    public static void downloadBinaryData(Context context) throws IOException {

```



```

File file = new File(context.getFilesDir(), "downloaded-image.png");
FileOutputStream fileOutputStream = new FileOutputStream(file);
URL url = new URL("http://moagrius.com/assets/images/hero-trips.png");
URLConnection connection = (URLConnection) url.openConnection();
int data;
while ((data = connection.getInputStream().read()) != -1) {
    fileOutputStream.write(data);
}
fileOutputStream.flush();
}
}

```

Kotlin

```

@Throws(IOException::class)
fun downloadBinaryData(context: Context) {
    val file = File(context.filesDir, "downloaded-image.png")
    val fileOutputStream = FileOutputStream(file)
    val url = URL("http://moagrius.com/assets/images/hero-trips.png")
    val connection = url.openConnection() as HttpURLConnection
    var data: Int = connection.inputStream.read()
    while (data != -1) {
        fileOutputStream.write(data)
        data = connection.inputStream.read()
    }
    fileOutputStream.flush()
}

```

После вызова этого метода у вас должна появиться точная копия файла с моего сайта. Если у вас это получилось – поздравляю!

i Круг сетевых взаимодействий намного шире, чем было показано в этой главе, но мы подскажем вам, куда еще можно обратиться за дополнительной информацией:

- OkHttp (https://oreil.ly/_Lk9z) и Volley (<https://oreil.ly/1fSlz>) – обе эти библиотеки предлагают дополнительные сетевые API;
- Picasso (<https://oreil.ly/0VyGt>), Glide (<https://oreil.ly/D3HE>) и Volley (<https://oreil.ly/2gaY5>) предлагают очень простые API для загрузки и отображения удаленных изображений;
- Gson (<https://oreil.ly/RRhOS>) и Jackson (<https://oreil.ly/e95wK>) – превосходные парсеры JSON, которые можно использовать для чтения данных из REST API;
- Retrofit (<https://oreil.ly/lT-5Z>) использует библиотеку OkHttp для отправки сетевых запросов и библиотеку Gson для парсинга данных в формате JSON и предлагает очень простые инструменты для чтения удаленных данных.

iOS

Поддержка сетевых взаимодействий изначально была одной из сильных сторон macOS и iOS. Система предлагает хорошо продуманный и исчерпывающий набор объектов. Фактически большинство сторонних сетевых библиотек для iOS основываются на классах, предлагаемых операционной системой. Среди этих библиотек есть несколько очень хороших вариантов, как, например, Ala-

moFire, но большинство разработчиков предпочитают использовать встроенные инструменты. Поэтому в этой главе мы все свое внимание сосредоточим на инструментах, имеющихся непосредственно в iOS.

Вперед!

Загрузка текстового файла с удаленного сервера и его вывод

Чтение текстовых данных с веб-сервера, где бы он не находился, – это самое простое сетевое взаимодействие, какое только можно представить. Вот как в iOS можно запросить информацию с сервера:

```
let url = URL(string: "https://www.oreilly.com")!
let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
    guard let data = data else { return }
    let string = String(data: data, encoding: .utf8)!
    print(string)
}
task.resume()
```

Рассмотрим поближе, что здесь происходит.

Сначала создается экземпляр URL, указывающий на определенный файл, веб-сайт или API – в данном случае на домашнюю страницу O'Reilly Media. Затем с помощью общего объекта URLSession мы генерируем задание URLSessionDataTask для указанного URL и передаем обработчик успешного выполнения в последнем замыкании. Внутри обработчика мы преобразуем полученные данные (в виде объекта Data) в экземпляр String, а затем выводим их – это самый обычный HTML-код, который предназначен для отображения в веб-браузере. В заключение мы вызываем resume() созданного задания, чтобы запустить весь процесс.



Возможно, вы заметили, что в примере мы используем URL со схемой HTTPS. Начиная с iOS 9 все HTTP-запросы должны использовать схему HTTPS, если явно не разрешено иное в файле *Info.plist* приложения в ключе `NSAppTransportSecurity`.

В этом примере появилось несколько новых классов. Первый из них – URLSession. Этот класс представляет верхний уровень сетевых API в iOS. Основная функция каждого экземпляра URLSession в iOS заключается в координации различных сетевых задач; задачи связываются непосредственно с объектом URLSession во время их создания. Сами объекты сеансов можно настраивать, передавая им объект URLSessionConfiguration в инициализаторе. При необходимости можно создать несколько своих сеансов или использовать общий сеанс, доступный по метке названному свойству URLSession.shared, как показано выше.



Экземпляры URLSession можно настраивать разными способами. Для удобства URLSessionConfiguration предлагает ряд стандартных конфигураций. К ним относятся default, ephemeral и background(with:). За дополнительной информацией обращайтесь к документации для разработчиков Apple (https://oreil.ly/_ldvZ).

Экземпляр `URLSession` также является инициатором заданий. Сетевое задание можно представить как операцию; оно реализует отправку сетевого запроса и получение ответа. Подобно операциям, они часто выполняются асинхронно.

Вот три основных типа заданий:

- `URLSessionDataTask` – используется для получения данных (возвращаются в форме объекта `Data`) с сервера;
- `URLSessionDownloadTask` – в основном используется для загрузки файлов с сервера и может приостанавливаться и возобновляться, а также позволяет отображать прогресс загрузки (об этом мы поговорим далее этой главе);
- `URLSessionUploadTask` – применяется для отправки данных на сервере (как будет показано в следующем разделе).

Важно знать и понимать, когда использовать тот или иной тип заданий. Задания `URLSessionDataTask` редко используются для загрузки больших файлов. Задания `URLSessionDownloadTask` было бы неразумно (и довольно громоздко) использовать для получения простого ответа в формате JSON. Задания специализированы, чтобы сделать нашу работу как можно проще и логичнее.

Теперь вернемся к предыдущей нашей реализации. Такой упрощенный способ загрузки файлов редко используется на практике. Вот гораздо более полный пример, проверяющий ошибки, допустимые коды состояния ответа сервера и пустые данные:

```
let url = URL(string: "https://www.oreilly.com")!
let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
    if let error = error {
        print(error.localizedDescription)
        return
    }

    guard let response = response as? HTTPURLResponse, response.statusCode < 300 else {
        print("A server error occurred.")
        return
    }

    guard let data = data, let string = String(data: data, encoding: .utf8) else {
        print("No data returned.")
        return
    }

    print(string)
}
task.resume()
```

Получение данных с сервера – безусловно, полезная возможность, и большинство сетевых вызовов – это запросы на получение данных. Но для работы с REST API нужно уметь не только получать, но и посылать данные.

Давайте посмотрим, как это делается в iOS.

Создание запроса HTTP POST

Прежде чем посылать произвольные данные, попробуем отправить на сервер фрагмент текста. Вот как это можно сделать:

```

let data = "text to send".data(using: .utf8)
let url = URL(string: "https://www.oreilly.com")!

var urlRequest = URLRequest(url: url)
urlRequest.httpMethod = "POST"
urlRequest.httpBody = data

let task =
    URLSession.shared.dataTask(with: urlRequest) { (data, response, error) in
        ...
    }
task.resume()

```

Это простой пример, но обычно требуется отправлять на сервер структурированные текстовые строки, часто в формате JSON или в виде списка пар ключ/значение. Для отправки таких строк удобно использовать объект `URLComponents`, с помощью которого можно создавать списки пар ключ/значение, как показано ниже:

```

var components = URLComponents()
components.queryItems = [
    URLQueryItem(name: "name", value: "O'Reilly"),
    URLQueryItem(name: "isAwesome", value: "true")
]

let url = URL(string: "https://www.oreilly.com")!

var urlRequest = URLRequest(url: url)
urlRequest.httpMethod = "POST"
urlRequest.httpBody = components.query?.data(using: .utf8)

let task =
    URLSession.shared.dataTask(with: urlRequest) { (data, response, error) in
        ...
    }
task.resume()

```

Этот код сначала создает экземпляр `URLComponents`. Затем добавляет в его свойство `queryItems` элементы – объекты `URLQueryItem` с парами ключ/значение. После этого объект преобразуется в строку и добавляется в свойство `httpBody` объекта `URLRequest`.

Также часто бывает нужно отправить произвольные данные, не организованные в пары ключ/значение. Вот пример выгрузки файла на сервер:

```

let data = "file data".data(using: .utf8)!

let url = URL(string: "http://www.example.com")!
var request = URLRequest(url: url)
request.httpMethod = "POST"

let task = URLSession.shared.uploadTask(with: request, from: data)
task.resume()

```

Здесь сначала создается строка данных, представляющая наш фиктивный файл, `file data`, содержимое которого преобразуется в объект `Data`, чтобы подготовить его к последующей отправке. Затем определяется URL с адресом `http://`

`www.example.com` (это фиктивный адрес и в данном примере играет роль заполнителя) и передается в новый объект `URLRequest`, который мы уже использовали в примере отправки запроса `POST`. `URLRequest` – это абстракция нашего запроса. Этот объект позволяет произвести некоторые настройки перед отправкой запроса на сервер. В данном примере мы настраиваем используемый метод HTTP: `POST`. Значение свойства `httpMethod` отображается непосредственно в стандартный глагол `REST`, по умолчанию используется `GET`.

Наконец, в этом примере, так же как в примерах в предыдущем разделе, создается задание, однако в этом случае создается задание `URLSessionUploadTask`, для чего вызывается метод `uploadTask`. Чтобы запустить задание отправки объекта `data` на сервер и выполнить запрос, вызывается метод `resume()`.

Это простой пример. Вероятно, сервер вернет какие-то данные с информацией об обработке запроса, включая информацию об успехе или неудаче. Чтобы получить эту информацию, можно использовать перегруженную версию метода `uploadTask`, как показано ниже:

```
...
let task =
    URLSession.shared.uploadTask(with: request, from: data) { (data, response, error) in
        // Обработать в еггор ошибки, возникшие на стороне клиента
        // Обработать в response ошибки, возникшие на стороне сервера
        // Обработать успешное выполнение запроса,
        // используя полученный объект data
    }
}
```

Большинство современных API принимают структурированные данные, например, в формате `JSON` и возвращают ответ в формате `JSON`. Наш упрощенный пример отправки строки с данными легко можно распространить на такие случаи без внесения существенных изменений, кроме создания структуры для данных. Ниже приводится законченный пример, демонстрирующий отправку данных в формате `JSON` и обработку ответа:

```
// Определение объекта для представления данных
struct Book: Codable {
    let title: String
    let isbn: String
}

// Заполнить структуру данными
let book = Book(title: "Native Application Development", isbn: "this ISBN")

// Преобразовать структуру в объект Data для передачи в запрос
let data = try! JSONEncoder().encode(book)

// Создать запрос
let url = URL(string: "http://www.example.com")!
var request = URLRequest(url: url)
request.httpMethod = "POST"
request.setValue("application/json", forHTTPHeaderField: "Content-Type")

// Создать и запустить задание
```

```

let task =
    URLSession.shared.uploadTask(with: request, from: data) { (data, response, error) in
        // Обработать ошибки, возникшие на стороне клиента
        if let error = error {
            print(error.localizedDescription)
            return
        }

        // Обработать ошибки, возникшие на стороне сервера
        guard let response =
            response as? HTTPURLResponse, response.statusCode < 300 else {
                print("A server error occured.")
                return
            }

        // Проверить наличие возвращаемых данных
        guard let data = data else {
            print("No data returned.")
            return
        }

        // Декодировать полученный фрагмент JSON в экземпляр Book
        do {
            let book = try JSONDecoder().decode(Book.self, from: data)
            print("The book's title is \(book.title) and the ISBN is \(book.isbn)")
        } catch {
            print("Could not decode response to JSON")
        }
    }
task.resume()

```

Пример выглядит довольно объемным, но если присмотреться внимательно, вы найдете лишь несколько строк с существенными изменениями. Во-первых, в начале примера определяется структура для представления данных. Затем с помощью `JSONEncoder` эта структура преобразуется в объект `Data`. `JSONEncoder` входит в состав стандартной библиотеки Swift и предлагает простой метод для сериализации и десериализации объектов JSON.

Существует целая глава, посвященная «транспортным данным», таким как JSON. Вы можете проверить это для получения дополнительной информации о функциональности и доступных методах.



Далее в книге вы найдете отдельную главу, посвященную «транспортировке данных». В ней приводится более подробная информация об имеющихся методах и возможностях.

Следующее существенное изменение можно заметить в строке

...

```
request.setValue("application/json", forHTTPHeaderField: "Content-Type")
```

Эта команда добавляет в запрос HTTP-заголовок `Content-Type` со значением `application/json`. Аналогично можно добавить другие HTTP-заголовки, но для большинства API достаточно явно указать, что данные посылаются в формате JSON.

Остальной код в вышеприведенном примере выглядит знакомым по предыдущим примерам, за исключением обработчиков. Здесь выполняются некоторые проверки на наличие ошибок на стороне клиента и/или сервера, а также данных, возвращаемых сервером. Затем используется новый объект `JSONDecoder`, чтобы преобразовать полученный объект `data` (который на самом деле просто хранит данные в формате JSON) непосредственно в экземпляр `Book`, который, в свою очередь, используется для вывода свойств `title` и `isbn`.

Большинство сетевых операций, реализованных в `URLSession`, имеют определенные сходства. Однако, чтобы загрузить файл с сервера, придется написать дополнительную логику, учитывающую несколько иной набор требований. Давайте посмотрим, что это за логика, рассмотрев в следующем разделе загрузку двоичных файлов.

Загрузка двоичного файла

Загрузка файлов в коде очень похожа на запрос данных с сервера. Следующий пример показывает, как послать серверу запрос на загрузку файла:

```
let url = URL(string: "https://www.example.com/file.zip")!
let task =
    URLSession.shared.downloadTask(with: url) { (fileUrl, response, error) in
        // Обработать ошибки, возникшие на стороне клиента
        if let error = error {
            print(error.localizedDescription)
            return
        }

        // Обработать ошибки, возникшие на стороне сервера
        guard let response =
            response as? HTTPURLResponse, response.statusCode < 300 else {
                print("A server error occurred.")
                return
            }

        // Проверить наличие загруженного файла
        guard let tempFileUrl = fileUrl else { return }
        print(tempFileUrl.path)
    }
task.resume()
```

В этом примере мы запрашиваем файл с определенным URL. Одно из основных отличий состоит в том, что вместо `dataTask(with:)` для создания запроса мы используем `downloadTask(with:)`. Обработчик результата имеет немного другую сигнатуру – вместо экземпляра `Data` в него передается экземпляр `URL`, которому мы дали имя `fileUrl` в нашем примере.

Задания загрузки отличаются от заданий получения данных тем, что последние получают данные в памяти, а задания загрузки получают ссылку на временный файл по завершении загрузки. Загрузка коротких файлов может выполняться мгновенно, но для загрузки больших файлов может понадобиться значительное время.

Наш обработчик завершения загрузки будет вызван сразу, как только закончится загрузка файла, и получит ссылку на временную папку, куда был записан этот файл. Эти файлы недолговечны и будут удалены системой при первом удобном случае.

❗ Не забывайте переместить файл в другое место, например в папку *Documents*. Предыдущий пример просто выводит путь к файлу. За информацией о том, как переместить файл, обращайтесь к главе 6.

При выполнении загрузки больших файлов желательно предоставлять пользователям некоторый интерфейс, сообщающий, какая часть файла загружена и сколько еще осталось загрузить. Наш упрощенный пример не использует этот механизм. Однако в подсистеме `NSURLSession` есть способ реализовать это: протокол `NSURLSessionDownloadDelegate`.

NSURLSessionDownloadDelegate

`NSURLSessionDownloadDelegate` содержит несколько необязательных методов для реализации в объектах, предназначенных для обработки событий, возникающих в процессе загрузки. Вот, например, как можно реализовать делегата, добавить его в `NSURLSession` и создать задание загрузки, обновляющее прогресс:

```
class NetworkClient: NSObject {
    // ...
}
extension NetworkClient: URLSessionDownloadDelegate {
    func urlSession(_ session: URLSession,
                   downloadTask: URLSessionDownloadTask,
                   didFinishDownloadingTo location: URL) {

        // Проверить наличие ошибок на стороне сервера
        guard let response =
            downloadTask.response as? HTTPURLResponse, response.statusCode < 300 else {
            return }

        // Вывести местоположение временного файла
        print(location.path)
    }

    func urlSession(_ session: URLSession, task: URLSessionTask,
                   didFinishWithError error: Error?) {
        if let error = error {
            print(error.localizedDescription)
        }
    }

    func urlSession(_ session: URLSession,
                   downloadTask: URLSessionDownloadTask,
                   didWriteData bytesWritten: Int64,
                   totalBytesWritten: Int64,
                   totalBytesExpectedToWrite: Int64) {
        let percent = (totalBytesWritten/totalBytesExpectedToWrite) * 100
        print(percent)
    }
}
```



```

}
let url = URL(string: "https://www.example.com/file.zip")!
let client = NetworkClient()
let urlSession = URLSession(configuration: .default, delegate: client, delegateQueue: nil)
let task = urlSession.downloadTask(with: url)
task.resume()

```

Рассмотрим этот пример подробнее.

Сначала определяется новый класс `NetworkClient`, наследующий `NSObject` – базовый класс для всех объектов в языке Objective-C. Для этого класса определяется расширение, в котором реализуется протокол `URLSessionDownloadDelegate`. Первый метод, `urlSession(_:downloadTask:didFinishDownloadingTo:)`, является обязательным: когда задание загрузки завершится, оно вызовет этот метод, чтобы сообщить, где находится временный загруженный файл. Также в теле этого метода можно обработать ошибки, возникшие на стороне сервера, проверив код состояния в свойстве `response` объекта `downloadTask`.

Следующий метод, `urlSession(_:task:didCompleteWithError:)`, является необязательным, но в промышленных приложениях лучше не пренебрегать им. Здесь можно обработать ошибки, возникшие на стороне клиента, и проверить, не возникли ли проблемы, препятствующие успешной загрузке файла. В этом примере мы просто выводим описание ошибки в консоль.

Наконец то, ради чего мы написали этот пример, – передача прогресса загрузки в приложение. Последний метод в расширении, `urlSession(_:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:)`, вызывается периодически с интервалом, определяемым сетевыми подсистемами в iOS, и получает значения `totalBytesWritten` и `totalBytesExpectedToWrite`. Эти значения можно использовать для отображения на экране чего угодно, от текстовых меток до компонентов `UIProgressView` с полосой прогресса. В нашем примере мы выводим значение в консоль, но вы можете воспользоваться своим воображением, чтобы представить, что вообще возможно!

В конце примера создается экземпляр нашего класса, который затем передается в инициализатор `URLSession` в параметре `delegate`. Обратите особое внимание на две строки, где это происходит:

```

...
let urlSession =
    URLSession(configuration: .default, delegate: client, delegateQueue: nil)
let task = urlSession.downloadTask(with: url)
task.resume()

```

Сначала создается экземпляр `URLSession`, а затем, в следующей строке, он используется для создания экземпляра `downloadTask`. Мы не использовали экземпляр `URLSession.shared`, как в предыдущих примерах, потому что нам понадобилось определить делегата, а в `URLSession` это можно сделать только с помощью инициализатора.

Приостановка и возобновление

Обновления прогресса в процессе длительной загрузки – не единственное, что может понадобиться вашим пользователям. Возможно, в какой-то момент за-

грузка может потерпеть неудачу. Или, может быть, пользователь решит приостановить загрузку, чтобы переключиться на сеть Wi-Fi, и затем возобновить ее. В любом случае, приостановка и возобновление `NSURLSession` возможны, но выглядит эта процедура немного странно.

Давайте сначала рассмотрим пример, а затем обсудим его:

```
class PauseableClient: NSObject {
    let url = URL(string: "https://www.example.com/file.zip")!
    var resumeData: Data?

    func startDownload() -> URLSessionDownloadTask? {
        let task = URLSession.shared.downloadTask(with: url)
        task.resume()
        return task
    }

    func pauseDownload(for task: URLSessionDownloadTask?) {
        guard let task = task else { return }
        task.cancel { (resumeData) in
            self.resumeData = resumeData
        }
    }

    func resumeDownload() -> URLSessionDownloadTask? {
        guard let resumeData = resumeData else {
            print("Download can't be resumed!")
            return nil
        }

        let task = URLSession.shared.downloadTask(withResumeData: resumeData)
        task.resume()
        return task
    }
}

let client = PauseableClient()
var task = client.startDownload()
client.pauseDownload(for: task)
task = client.resumeDownload()
```

В этом примере определяется класс `PauseableClient` с тремя методами: `startDownload()`, `pauseDownload()` и `resumeDownload()`. Метод `startDownload()` запускает задание загрузки предположительно большого файла. Он возвращает объект задания `task`, который будет использоваться позже. Далее мы вызываем `startDownload()` и тут же вызываем метод `pauseDownload(for:)`, которому передаем задание.

Внутри `pauseDownload()` мы вызываем метод `cancel(byProducingResumeData:)` полученного задания и передаем ему замыкание в единственном аргументе. Сразу после отмены задания вызывается наше замыкание, и ему передается токен – экземпляр `Data`, – который затем используется для возобновления загрузки. Сохранение этого токена позволяет в будущем возобновить загрузку.

i Данные для возобновления можно также сохранить в методе делегата `urlSession(_:task:didCompleteWithError:)`, который вызывается в случае появления ошибки на стороне клиента, и проверить `userInfo[NSURLSessionDownloadTaskResumeData]` на наличие ошибки в параметре `error`. Также следует отметить, что существуют определенные ограничения, касающиеся генерирования данных для возобновления. За дополнительной информацией о них обращайтесь к документации разработчика Apple с описанием `NSURLSessionTask` (<https://oreil.ly/GITHH>).

На самом деле задание, созданное для загрузки файла, на этом этапе исчезнет. Вызов `cancel` в действительности не приостанавливает загрузку, а *отменяет* ее. Тем не менее этот запрос можно вернуть к жизни без повторной загрузки уже загруженных данных.

Это делается в нашем примере с помощью метода `resumeDownload()`, который проверяет данные для возобновления и затем вызывает `NSURLSession`, чтобы создать новый экземпляр `NSURLSessionDownloadTask` вызовом метода `downloadTask(withResumeData:)`. Данные для возобновления – это тот же экземпляр `resumeData`, который был сохранен в `pauseDownload(for:)` при отмене предыдущего задания.

Наконец, после создания нового задания вызывается метод `resume()`, чтобы продолжить загрузку с той же точки, на которой она была прервана раньше.

Делегаты

В подсистеме `NSURLSession` есть также другие делегаты и методы делегатов, чем было показано в примерах в этой главе. Для обработки событий уровня задания объекты могут также реализовать некоторые или все методы протоколов `NSURLSessionTaskDelegate`, `NSURLSessionDataDelegate` или `NSURLSessionDownloadDelegate`. К ним относятся такие события, как начало и конец отдельных запросов, а также периодические обновления данных или заданий загрузки, подобные тем, что были показаны выше.

Фоновые потоки и обновление пользовательского интерфейса

`NSURLSessionTask` действует асинхронно, в своих собственных фоновых потоках, не блокируя текущий поток, и для этого не требуется делать ничего особенного. Однако вызываемые замыкания не возвращаются в основной поток, поэтому любые обновления пользовательского интерфейса нужно отправлять явно или использовать механизмы синхронизации с основным потоком.

Безопасность передаваемых данных

Для обращения к небезопасным URL, например начинающимся с `http://`, необходимо определить специальные настройки для ключа `NSAppTransportSecurity` в файле `Info.plist` приложения. Мы советуем перечислить в `NSExceptionDomains` доверенные домены, которые могут быть небезопасными. Можно полностью отключить механизм поддержки безопасности App Transport Security, установив в его ключе `NSAllowsArbitraryLoads` значение `true`, но мы не советуем поступать так без крайней необходимости.

Что мы узнали

В этой главе мы узнали, как организовать сетевые взаимодействия в Android и iOS, в том числе:

- как выполнить простой запрос на обеих платформах. В обеих системах используется похожий процесс, но в Android, в отличие от iOS, используются потоки данных;
- как отправить данные на сервер, чтобы разработчик мог полностью контролировать запрос и ответ;
- обсудили особенности загрузки двоичных файлов и что необходимо сделать, чтобы сохранить их в файловой системе;
- механизмы управления безопасностью, встроенные в iOS, по умолчанию требуют использовать схему HTTPS. Система Android более свободна, открыта и накладывает меньше ограничений в отношении настроек сетевой безопасности по умолчанию.

Сети иногда оказываются недоступны, и в них могут возникать сбои. В следующей главе мы покажем, как дать пользователям обратную связь при появлении проблем или когда вам просто нужно держать их в курсе каких-либо событий. Поехали!

Глава 10

Обратная связь с пользователем

Для любой программы важна возможность предоставлять пользователю оперативную информацию в короткоживущем интерфейсе. Это особенно верно для мобильных приложений, где мы быстро исчерпали бы доступные ресурсы, предусматривая появление каждого предупреждения, приглашения или уведомления в пользовательском интерфейсе. Обе системы предлагают разные средства для достижения этой цели.

Задачи

В этой главе вы узнаете:

- 1) какие системные инструменты можно использовать для обратной связи с пользователем в разных обстоятельствах;
- 2) как обновлять строку состояния.

ANDROID

В Android есть несколько API для немедленного отображения информации о событии, ошибке или изменении состояния. Некоторые просто отображают текст, а другие допускают различные уровни настройки и интерактивности. Далее мы рассмотрим три основных класса обратной связи и покажем, как их использовать с учетом конкретных потребностей.

Отображение обратной связи с использованием системных инструментов

Вы можете предоставлять информацию своим пользователям любым удобным для вас способом, используя стандартные методы компоновки и рисования, однако Android предлагает еще три API, специально созданных для отображения обратной связи:

- класс `Toast`;

- класс `Snackbar`;
- класс `Dialog`.

Класс `Toast` появился раньше `Snackbar`, и хотя основные приемы их использования очень похожи, первый выглядит немного проще. С другой стороны, при правильном использовании класс `Snackbar` может обеспечить более привлекательный пользовательский интерфейс и обладает гибкостью, недоступной для сообщений `Toast`. Диалоги – это модальные интерфейсы, поддерживающие широкий спектр элементов, включая элементы для отображения *и получения* информации от ваших пользователей.

Итак, приступим.

Toast

Внешний вид сообщений `Toast` зависит от реализации. Некоторые детали могут быть по-разному реализованы в разных версиях ОС Android, также различия могут иметь место в зависимости от производителя устройства.

Вообще говоря, сообщение `Toast` – это короткое и кратковременное сообщение, отображаемое поверх существующего контента. Вот цитата из документации для разработчиков Android (<https://oreil.ly/zXgqS>):

Тост – это всплывающее уведомление с информацией о той или иной операции. Он занимает минимум места, не отвлекая пользователя от текущего действия и не нарушая работоспособности приложения. По истечении тайм-аута тост автоматически исчезает.

Вот как можно вывести сообщение `Toast`:

Java

```
Toast.makeText(context, "Hi there!", Toast.LENGTH_SHORT).show();
```

Kotlin

```
Toast.makeText(context, "Hi there!", Toast.LENGTH_SHORT).show()
```

Очень просто! Для вывода сообщения необходимо передать экземпляр `Context`, текст сообщения и константу, определяющую продолжительность показа (`LENGTH_SHORT` или `LENGTH_LONG`).

Класс `Toast` имеет также ряд методов для настройки отображения сообщения на экране (его «тяжесть») и даже для создания всплывающих окон `Toast` с нестандартными деревьями представлений, но в большинстве случаев используются только методы `makeText` и `show`.

Snackbar

Как уже упоминалось, основные приемы использования `Toast` и `Snackbar` очень похожи – оба класса выводят короткое сообщение поверх существующего пользовательского интерфейса. Фактически в самом простом случае вызов `Snackbar` почти идентичен вызову `Toast`, показанному выше:

Java

```
Snackbar.make(someView, "Hello there!", Snackbar.LENGTH_SHORT).show();
```

Kotlin

```
Snackbar.make(someView, "Hello there!", Snackbar.LENGTH_SHORT).show()
```

В первом параметре методу `Snackbar.make` вместо экземпляра `Context` передается экземпляр `View`, и имя самого метода немного отличается (`Toast.makeText` и `Snackbar.make`), но в остальном они идентичны.

Основное отличие заключено в восприятии пользователем:

сообщения `Snackbar` обычно появляются снизу вверх, а сообщения `Toast` плавно проявляются в центре экрана.

В сообщении `Snackbar` можно включить простую кнопку, справа от текста, единственным вызовом метода: `setAction`. То же возможно при использовании класса `Toast`, но для этого придется написать больше кода.

Также обратите внимание, что в документации для разработчиков Android (<https://oreil.ly/XPy7s>) настоятельно рекомендуется отдавать предпочтение `Snackbar`:

Класс `Snackbar` является дальнейшим развитием `Toast`. В настоящее время `Toast` еще поддерживается, но для отображения коротких и кратковременных сообщений предпочтительнее использовать `Snackbar`.

Однако главное отличие `Snackbar` от `Toast` можно заметить, присоединив `Snackbar` к `CoordinatorLayout`, играющему роль корневого представления экземпляра `View`, передаваемого в метод `make` в первом параметре. В этом случае `Snackbar` обретает дополнительные особенности, такие как возможность закрыть сообщение, смахнув его с экрана, или учесть размещение других компонентов, управляемых `CoordinatorLayout`. Например, `FloatingActionButton` будет сдвигаться вверх и в сторону при вставке `Snackbar`. За дополнительной информацией обращайтесь к документации для разработчиков Android (<https://oreil.ly/n8Mbx>).

Dialog

Класс `Dialog` и его подклассы – более мощные и гибкие инструменты, чем `Toast` и `Snackbar`, но они также требуют больше внимания, обслуживания и настройки.

Вот цитата из документации (<https://oreil.ly/xPFao>):

Диалог – это небольшое окно, которое предлагает пользователю принять решение или ввести дополнительную информацию. Диалог не заполняет весь экран и, как правило, используется в особых случаях, когда требуется вмешательство пользователя перед продолжением.

Так же как при использовании ранее обсуждавшихся классов, диалогу можно придать сколь угодно сложный интерфейс; существуют также средства обратной связи для выбора одного или нескольких вариантов, значительно упрощающие получение ответа пользователя.

Вот как можно создать и показать диалог с заголовком, сообщением и кнопками **Принять** и **Отменить**, чтобы пользователь мог принять или отменить предложенную операцию:

Java

```
new AlertDialog.Builder(context).setMessage(R.string.my_message)
    .setTitle(R.string.my_title)
    .setPositiveButton(R.string.my_accept_string,
        (d, id) -> Log.d("MyTag", "Accepted!"))
    .setNegativeButton(R.string.my_cancel_string,
        (d, id) -> Log.d("MyTag", "Cancelled!"))
    .create().show();
```

Kotlin

```
AlertDialog.Builder(context).setMessage(R.string.my_message)
    .setTitle(R.string.my_title)
    .setPositiveButton(R.string.my_accept_string,
        { d, id -> Log.d("MyTag", "Accepted!") })
    .setNegativeButton(R.string.my_cancel_string,
        { d, id -> Log.d("MyTag", "Cancelled!") })
    .create().show()
```

За дополнительной информацией о диалогах обращайтесь к документации для разработчиков (<https://oreil.ly/-VXCz>).

Изменение строки состояния

В Android строка состояния (status bar) – это самая верхняя область экрана, где находится информация о системе, такая как время, уровень заряда батареи, состояние сети и т. д. Системная информация обычно отображается в правой половине в этой верхней панели, а слева обычно выводится информация уровня приложения, чаще в форме маленьких значков. Эти значки известны как значки «уведомлений» и обычно соответствуют экземплярам `Notification`, представляющим пользовательский интерфейс, который существует за пределами приложения, и, как правило, состоят из коротких сообщений, но также могут включать в себя интерактивные элементы. Например, завершив загрузку, приложение может создать уведомление `Notification` с сообщением для пользователя и значком в строке состояния, чтобы пользователь мог открыть это сообщение позже. К сообщению также можно приложить некоторые действия, такие как открытие в интерфейсе приложения загруженного элемента или списка загрузок. Класс `Notification` обладает полным набором функций, но, пожалуй, не настолько интуитивным, как некоторые из инструментов, обсуждавшихся выше.

Также обратите внимание, что `Notification` несколько раз кардинально менялся в процессе развития AOSP. Основные изменения были внесены в ОС Android версии 5, а также в версии 8. В этой главе рассматриваются только самые основы, а за информацией об остальных возможностях, как обычно, обращайтесь к документации для разработчиков (<https://oreil.ly/wGqyn>).

Следующий пример иллюстрирует использование `Notification` посредством библиотеки `compat` (не забудьте добавить зависимость от `com.android.support:supportcompat:xx.xx.xx`):

Java

```
private static final int CHANNEL_ID = 1;
private static final int NOTIFICATION_ID = 1;

...

NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID)
    .setSmallIcon(R.drawable.my_notification_icon)
    .setContentTitle("Title")
    .setContentText("Message body.")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT);

NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, builder.build());
```

Kotlin

```
const val CHANNEL_ID = 1
const val NOTIFICATION_ID = 1

...

val builder = NotificationCompat.Builder(this, CHANNEL_ID)
    .setSmallIcon(R.drawable.my_notification_icon)
    .setContentTitle("Title")
    .setContentText("Message body.")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
val notificationManager = NotificationManagerCompat.from(this)
notificationManager.notify(NOTIFICATION_ID, builder.build())
```

Чтобы запустить Activity, когда пользователь коснется уведомления, используйте класс `PendingIntent`:

Java

```
Intent intent = new Intent(this, DownloadsActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(context, 0, intent, 0);
```

Kotlin

```
val intent = Intent(this, DownloadsActivity::class)
val pendingIntent = PendingIntent.getActivity(context, 0, intent, 0)
```

Добавьте его в построитель перед вызовом `build`:

Java

```
builder.setContentIntent(pendingIntent);
```

Kotlin

```
builder.contentIntent = pendingIntent
```

За дополнительной информацией обращайтесь к документации (<https://oreil.ly/rcuYz>).



Эти встроенные механизмы обратной связи с пользователями могут способствовать повышению удобства использования приложений. Например, встроенная диалоговая система с кнопками **Принять** и **Отменить** может показаться пользова-

телям более знакомой, и они с большей вероятностью подпишутся на вашу ссылку (или дадут вашему приложению хороший рейтинг), если вы используете их. Однако если эти инструменты не соответствуют вашим требованиям, есть возможность создать (или изменить) свои компоненты пользовательского интерфейса, лучше подходящие для вашего случая.

iOS

Прочитавшие раздел об Android этой главы быстро заметят, что в данной сфере iOS и Android сильно отличаются друг от друга, и одно из наиболее заметных отличий – доступные способы отображения обратной связи. Давайте посмотрим, что доступно в iOS и какие варианты пользуются наибольшей популярностью.

Отображение обратной связи с использованием системных инструментов

В iOS есть два способа отображения обратной связи: предупреждения и листы запроса пользовательских действий (action sheets). Оба реализованы на основе класса `UIAlertController` и отличаются только передачей аргумента `.alert` или `.actionSheet`.

- ❑ В iOS нет встроенной поддержки вывода уведомлений в стиле `Snackbar`. Однако аналогичный эффект можно получить с помощью сторонней библиотеки, но мы не будем обсуждать эту возможность, так как это выходит далеко за рамки данной книги.

Apple в своей документации не дает четких указаний, когда лучше использовать листы действий, а когда предупреждения, однако за многие годы был выработан стандарт де-факто – использовать предупреждения, когда неожиданно возникло событие, требующее немедленной обратной связи, а листы действий – в ответ на действия пользователя, когда пользователю понятен контекст выполняемого действия и требуется, чтобы он выбрал один из вариантов.

Вот пример отображения предупреждения:

```
let viewController = UIViewController(nibName: ..., bundle: ...)

let alert = UIAlertController(
    title: "Title", message: "This is the message", preferredStyle: .alert)
alert.addAction(
    UIAlertAction(title: "OK", style: .default, handler: { (action) in
        print("OK button pressed!")
    })
)
viewController.present(alert, animated: true, completion: nil)
```

Рассмотрим этот код подробнее.

Сначала создается контроллер представлений, который потом будет задействован для отображения предупреждений. Начиная с версии iOS 8 представ-

ления предупреждений фактически являются экземплярами `UIViewController` и могут отображаться другими контроллерами представлений, как любые другие представления. Далее вызовом инициализатора `UIAlertController` создается экземпляр предупреждения. Ему передаются параметры `title`, `message` и `preferredStyle`. В последнем параметре передается аргумент `.alert`, указывающий, что создается предупреждение; если передать аргумент `.actionSheet`, будет создан лист действий.

В следующей строке вызывается метод `addAction(_:)`, чтобы добавить кнопку в предупреждение. Здесь мы непосредственно создаем экземпляр `UIAlertAction` для передачи в этот метод, определив заголовок, стиль (в данном случае `.default`) и обработчик кнопки. Внутри обработчика мы выводим сообщение **OK button pressed!** (Кнопка ОК нажата!) в консоль.

Наконец, мы используем контроллер представления, созданный в начале примера, чтобы отобразить предупреждение поверх имеющегося интерфейса.

Самое главное во всем этом – вызов метода `addAction(_:)`. Именно так добавляются кнопки в предупреждения и листы действий.

Теперь вы легко догадаетесь, как создать предупреждение с несколькими кнопками. Вот пример с тремя кнопками:

```
let viewController = UIViewController(nibName: nil, bundle: nil)

let alert = UIAlertController(title: "Terms & Conditions",
                             message: "Hit Agree to agree to terms and conditions.",
                             preferredStyle: .alert)

let agreeAction = UIAlertAction(title: "Agree", style: .default) { (action) in
    print("Terms and conditions accepted.")
}

let viewAction =
    UIAlertAction(title: "View Terms & Conditions", style: .default) { (action) in
        print("Blah blah blah.")
    }

let cancelAction = UIAlertAction(title: "Cancel", style: .cancel) { (action) in
    print("I disagree with terms and conditions.")
}

// Добавить все действия
alert.addAction(agreeAction)
alert.addAction(viewAction)
alert.addAction(cancelAction)

viewController.present(alert, animated: true, completion: nil)
```

Здесь мы используем почти идентичную логику создания и отображения предупреждения, с той лишь разницей, что действия создаются не в вызове `addAction(_:)`, а отдельно, после чего добавляются непосредственно в предупреждение.

Еще одно небольшое отличие этого примера – использование стиля `.cancel` для создания кнопки `cancelAction`. В iOS имеется ограниченный набор стилей кнопок. Стиль `.default` отображает простую кнопку, выделенную жирным шрифтом в двухкнопочной конфигурации. Стиль `.cancel` отображает кнопку

в соответствующей позиции и с соответствующей подписью. Последний стиль, `.destructive`, отображает кнопку, обычно с текстом красного цвета, чтобы показать, что она выполняет потенциально разрушительное действие.

Вот пример создания кнопки со стилем `.destructive`:

```
let deleteAction =
    UIAlertAction(title: "Yes, Delete", style: .destructive) { (action) in
        print("All your photos were deleted. What were you thinking?")
    }
}
```

Завершая обсуждение предупреждений и листов действий, мы должны предупредить вас: используйте эти инструменты только тогда, когда они действительно необходимы. Они разрушают поток восприятия пользователя и при чрезмерном использовании начинают игнорироваться. С большой властью приходит большая ответственность. Коснитесь кнопки **Да**, если вы согласны.

Изменение строки состояния

Класс `Notification` в Android позволяет разработчикам настраивать элементы в строке состояния и тем самым обеспечивать обратную связь. В iOS есть только один способ использования строки состояния для представления обратной связи – индикатор активности в строке состояния сети.

Несмотря на минималистский внешний вид, этот индикатор может быть очень полезен для пользователей. Он может помочь пользователю определить причину замедления работы приложения и подсказать, что данные все еще загружаются. Такие мелочи могут придать приложению ощущение завершенности.

Вот как это делается:

```
// Вывести индикатор
UIApplication.shared.isNetworkActivityIndicatorVisible = true

// Скрыть индикатор
UIApplication.shared.isNetworkActivityIndicatorVisible = false
```

Обратите внимание, как легко показать и скрыть индикатор. Многие сторонние сетевые библиотеки предлагают свои способы сделать то же самое, но почти все они в конечном итоге обращаются к системным методам, как показано ниже:

```
class NetworkClient {
    func startDownload() {
        ...
        UIApplication.shared.isNetworkActivityIndicatorVisible = true
    }

    func downloadCompleted() {
        ...
        UIApplication.shared.isNetworkActivityIndicatorVisible = false
    }
}
```

Этот способ не обладает той же широтой возможностей, как аналогичный ему в Android, тем не менее он остается невероятно полезным для пользователя.

Текстовые поля в предупреждениях

Предупреждения позволяют не только выводить текст, но и получать информацию от пользователя. Существует удобный метод для добавления текстового поля:

```
alert.addTextField { (textField) in
    textField.placeholder = "Enter your comment"
}
```

А вот как можно получить значение из этого поля в `UIAlertAction`:

```
let action = UIAlertAction(title: "Save", style: .default) { (action) in
    guard let textField = alert.textFields?[0] else { return }
    print(textField.text ?? "")
}
```

Тактильная обратная связь

Поддержка тактильной обратной связи позволяет сообщать пользователям о событиях не визуальным способом. Есть три класса для представления разных типов событий:

- `UINotificationFeedbackGenerator`;
- `UIImpactFeedbackGenerator`;
- `UISelectionFeedbackGenerator`.

Ниже приводится короткий пример тактильной обратной связи, сообщающей о неудачной загрузке:

```
let hapticFeedbackGenerator = UINotificationFeedbackGenerator()
hapticFeedbackGenerator.notificationOccurred(.error)
```

Этот код сгенерирует сигнал, воспринимаемый пользователем как нечто негативное и срочное. Используйте этот метод с осторожностью и помните, что не все классы устройств, например iPad, поддерживают средства тактильной обратной связи.

Что мы узнали

Это одна из областей разработки мобильных приложений, в которой Android и iOS имеют наибольшие отличия между технологическими стеками:

- Android поддерживает несколько отображений обратной связи, например `Toast`, `Snackbar` и `Dialog`. Также в Android имеется возможность использовать строку состояния для вывода предупреждений пользователю;
- в iOS круг средств обратной связи более ограниченный. Тем не менее доступные варианты просты в использовании и соответствуют стандартным соглашениям в `UIKit`;
- в iOS нельзя изменить содержимое строки состояния. Это одно из основных отличий между Android и iOS: степень контроля разработчика

над устройством. Компания Apple накладывает гораздо более жесткие ограничения. Разработчики могут только сообщить о наличии сетевой активности.

В этой главе мы узнали, как в наших приложениях сообщать нашим пользователям о важных событиях. В следующей главе мы поговорим о способах сохранения информации, немного менее громоздких, чем полноценный слой поддержки баз данных.

Глава 11

Предпочтения пользователя

Возможность персонализации приложения – это отличный способ помочь пользователю адаптировать приложение под свои нужды и предпочтения. Обе системы, Android и iOS, предлагают набор структур и шаблонов для решения этой задачи. Конечно, существуют сложные и громоздкие технологии, которые не только можно, но и часто необходимо использовать для реализации более сложных сценариев. Однако в большинстве случаев вполне можно ограничиться простыми и удобными средствами чтения и записи пользовательских настроек.

Задачи

В этой главе вы узнаете:

- 1) как сохранять предпочтения пользователя;
- 2) как читать предпочтения пользователя;
- 3) как работать с предпочтениями в многопользовательских приложениях.

ANDROID

В Android для хранения пользовательских настроек можно использовать файловую систему или базу данных, но вообще в Android имеется готовый механизм `SharedPreferences`. Он часто рекомендуется для хранения данных, но не является строго обязательным, и если вы обнаружите, что другой способ для вас удобнее, – смело используйте его.

Вот выдержка из документации для разработчиков Android (<https://oreil.ly/Bw8Eq>):

Если вам не требуется хранить много данных или какая-то определенная организация хранилища, используйте `SharedPreferences`. Механизм `SharedPreferences` позволяет читать и записывать пары ключ/значение простых типов данных: логические значения, числа с плавающей точкой, целые числа и строки.

SharedPreferences не является безопасным по умолчанию – значения хранятся в файле XML в каталоге приложения. Механизм KeyStore, предлагаемый системой, обеспечивает некоторую безопасность, но в старых операционных системах есть проблемы, которые могут усложнить его использование. Существуют сторонние библиотеки, которые (как утверждают их разработчики) предоставляют API, аналогичный SharedPreferences, но обеспечивают некоторый уровень безопасности.

Однако независимо от имеющихся ограничений механизм SharedPreferences прост и удобен в использовании и поддерживает выполнение в фоновых потоках, так что попробуйте его!

Сохранение предпочтений пользователя

Чтобы сохранить пару ключ/значение, нужен экземпляр SharedPreferences – в Android имеется уже готовый экземпляр, который можно получить вызовом метода `getSharedPreferences(String fileName, Context.MODE_PRIVATE)`; любого экземпляра `Context`. Экземпляры `Activity` имеют метод `getPreferences`, возвращающий файл с настройками по умолчанию, что позволяет опустить первый параметр (`fileName`).

После этого вам понадобится экземпляр `Editor`, который можно получить вызовом метода `edit` экземпляра `SharedPreferences`:

Java

```
SharedPreferences preferences = myContext.getSharedPreferences("prefs",
    Context.MODE_PRIVATE);
SharedPreferences.Editor editor = preferences.edit();
```

Kotlin

```
val preferences = myContext.getSharedPreferences("prefs",
    Context.MODE_PRIVATE)
val editor = preferences.edit()
```

Теперь можно сохранить пару ключ/значение, вызвав метод, такой как `putBoolean(String key, boolean value)` или `putString(String key, String value)`. После этого можно вызвать метод `commit` экземпляра `Editor`, который сохранит изменения синхронно, или метод `apply`, который сохранит изменения асинхронно:

Java

```
SharedPreferences preferences = myContext.getSharedPreferences("prefs",
    Context.MODE_PRIVATE);
SharedPreferences.Editor editor = preferences.edit();
editor.putBoolean("night mode", false);
editor.apply();
```

Kotlin

```
val preferences = myContext.getSharedPreferences("prefs", Context.MODE_PRIVATE)
val editor = preferences.edit()
editor.putBoolean("night mode", false)
editor.apply()
```


Как отмечалось выше, механизм `SharedPreferences` предназначен для сохранения простых значений и может принимать только базовые типы данных, такие как `boolean`, `int`, `long`, `float` и `String`, а также `Set<String>`.

Чтение предпочтений пользователя

Чтение предпочтений пользователя из `SharedPreferences` выполняется даже проще, чем запись, – для этого не нужен экземпляр `Editor` и не приходится беспокоиться о многопоточности, потому что копия сохраняется в памяти (что обеспечивает быстрый доступ к ней).

Вот как можно прочитать значение типа `boolean`, сохраненное в предыдущем примере:

Java

```
SharedPreferences preferences = myContext.getSharedPreferences("prefs",
                                                                Context.MODE_PRIVATE);
boolean isNightMode = preferences.getBoolean("night mode", false);
```

Kotlin

```
val preferences = myContext.getSharedPreferences("prefs", Context.MODE_PRIVATE)
val isNightMode = preferences.getBoolean("night mode", false)
```

Очень просто!

Работа с предпочтениями в многопользовательских приложениях

Когда одним приложением может пользоваться несколько человек, использование их предпочтений несколько осложняется. Технически `SharedPreferences` предназначен для управления настройками приложения в целом – в одном файле, *доступном всем*, кто использует приложение. Однако в последнее время получила распространение практика использования разных файлов `SharedPreferences` для разных пользователей – от вас требуется только определить уникальные имена файлов. Вам решать, как это реализовать, но некоторые, к примеру, зашифровывают идентификатор пользователя:

Java

```
String userId = // некоторый секретный идентификатор пользователя...
String sha = // зашифровать идентификатор для его защиты
SharedPreferences preferences = myContext.getSharedPreferences(sha,
                                                                Context.MODE_PRIVATE);
editor.putBoolean("night-mode", false);
editor.commit();

// прочитать обратно
boolean isNightMode = preferences.getBoolean("night-mode", false);
Log.d("MyTag", "night mode=" + isNightMode);
```

Kotlin

```

val userId = // некоторый секретный идентификатор пользователя...
val sha = // зашифровать идентификатор для его защиты
val preferences = myContext.getSharedPreferences(sha, Context.MODE_PRIVATE)
val editor = preferences.edit()
editor.putBoolean("night-mode", false)
editor.commit()

val isNightMode = preferences.getBoolean("night-mode", false)
Log.d("MyTag", "night mode=" + isNightMode)

```

iOS

В iOS поддерживается несколько способов хранения пользовательских данных: пользовательские настройки по умолчанию, файловая система, Core Data и Keychain. Часто приложениям приходится сохранять фрагменты информации, уникальные для конкретного пользователя, но не обязательно конфиденциальные. Лучшее место для таких данных – пользовательские настройки по умолчанию. Это место отлично подходит для хранения, например, языковых настроек, стиля пользовательского интерфейса или выбора единиц измерения, которые пользователь хочет применить для отображения данных.

К счастью, в iOS (и macOS) имеется проверенный способ хранения таких данных на уровне приложения: `UserDefaults`. Давайте рассмотрим его!

Сохранение предпочтений пользователя

Прежде чем обсуждать возможность чтения данных, нужно научиться сохранять их. Сохранение предпочтений довольно легко реализовать с использованием `UserDefaults`. Вот простой пример, показывающий, как сохранить строку:

```

let defaults = UserDefaults.standard
defaults.set("some string value", forKey: "someKey")

```

Сначала нужно получить общий экземпляр `UserDefaults`, а затем сохранить пару, состоящую из ключа "someKey" и строки "some string value". Этот ключ мы используем позднее для поиска связанной с ним строки, как будет показано далее в этой главе.

Что происходит за кулисами

В папке *app* каждого приложения iOS имеется папка `Library`, в которой находится папка `Preferences`. За кулисами iOS создает или обновляет файл со списком свойств, когда приложение записывает значение в `UserDefaults.standard`. Сохранение отдельных типов производится в соответствии с протоколом `NSCoding`, что позволяет сериализовать и десериализовать их. При желании можно добавить поддержку данного протокола в свои классы, но об этом – чуть позже!



Файл со списком свойств управляется механизмом `UserDefaults`, и его следует рассматривать как деталь реализации базовой подсистемы, которая может измениться в будущих версиях iOS.

Типы данных

UserDefaults может хранить данные разных типов, включая логические значения, числа, строки, адреса URL, словари, массивы и пользовательские объекты. Вот более сложный пример, демонстрирующий имеющиеся возможности:

```
let defaults = UserDefaults.standard

// Логическое значение
defaults.set(true, forKey: "nightMode")

// Число
defaults.set(2.0, forKey: "playbackSpeed")

// Строка
defaults.set("en-US", forKey: "locale")

// URL
let url = URL(string: "https://www.example.com/api")
defaults.set(url, forKey: "apiURL")
```

Библиотека Swift предлагает ряд удобных методов для передачи значений определенных типов данных в UserDefaults. Каждое значение объединяется со строковым ключом. Использование стандартных объектов Swift обеспечивает достаточно широкие возможности. Но как быть в случаях, когда хотелось бы сохранить пользовательский класс?

К счастью, это возможно, нужно лишь реализовать поддержку NSCoding.

Поддержка NSCoding

Чтобы объекты можно было сохранять с помощью UserDefaults, они должны поддерживать протокол NSCoding. Это предполагает реализацию двух методов: `init(coder:)` и `encode(with:)`. Эти методы используются для кодирования и декодирования объектов. Рассмотрим простой пример.

```
@objc (SomeObject)
class SomeObject: NSObject, NSCoding {
    let someProperty: String

    init(someProperty: String) {
        self.someProperty = someProperty
    }

    // Поддержка протокола NSCoding
    required convenience init?(coder aDecoder: NSCoder) {
        guard let someProperty = aDecoder.decodeObject(forKey:
            "someProperty") as? String else {
            return nil
        }
        self.init(someProperty: someProperty)
    }

    func encode(with aCoder: NSCoder) {
        aCoder.encode(someProperty, forKey: "someProperty")
    }
}
```

```
let someObject = SomeObject(someProperty: "some value")

let defaults = UserDefaults.standard
defaults.set(someObject, forKey: "myObject")
```

Здесь сначала объявляется явное `@objc`-имя для класса. Это обусловлено важностью имени класса для операции разархивирования объекта. Также объявляется, что класс поддерживает протокол `NSCoding`. Потом определяется строковое свойство с именем `someProperty`, которое получает значение в инициализаторе класса. Далее в методе `encode(with:)` мы используем экземпляр `NSCoder`, переданный в метод, для кодирования значения `someProperty` с ключом `"someProperty"`. Этот метод вызывается при каждом вызове метода `set(value:forKey:)` в `UserDefaults`.

Позже, когда понадобится выполнить декодирование, `UserDefaults` создаст объект вызовом инициализатора `init?(coder:)`. Внутри этого метода мы пытаемся установить каждое свойство, предварительно декодируя значение данного ключа, а затем передавая его в инициализатор объекта по умолчанию. Поскольку мы сами решаем, какие свойства кодировать и декодировать вручную, мы можем исключить из этого процесса некоторые свойства и инициализировать их на основе других данных.

! Этот пример сохранения «строкового типа» легко сломать, если неверно ввести имя ключа или оно изменится с выпуском новой версии приложения. Эту проблему можно решить с помощью перечисления и некоторых встроенных функций `NSKeyedUnarchiver`, таких как `NSKeyedUnarchiver.setClass(SomeObject.self, forClassName: "SomeObject")`.

Использование *Codable* вместо *NSCoding*

При желании можно опустить поддержку `NSCoding` в пользовательских объектах, которые требуется сохранять в `UserDefaults`, и для кодирования/декодирования в формат JSON использовать поддержку `Codable`. Одним из преимуществ применения `Codable` вместо `NSCoding` является отказ от использования всей среды выполнения Objective C. Например:

```
struct SomeObject: Codable {
    let someProperty: String
}

let someObject = SomeObject(someProperty: "some value")

// Сохранить объект
let defaults = UserDefaults.standard
if let json = try? JSONEncoder().encode(someObject) {
    defaults.set(json, forKey: "myObject")
}
```

Для кодирования экземпляра типа с поддержкой `Codable` можно использовать `JSONEncoder` и затем сохранить экземпляр `Data` непосредственно с заданным ключом (`myObject` в этом примере).

Декодирование объекта из формата JSON тоже выполняется просто, как показано в следующем примере:

```
// Прочитать значение
let json = defaults.value(forKey: "myObject") as! Data
let someObject = try? JSONDecoder().decode(SomeObject.self, from: json)
```

Здесь сначала извлекается объект из `UserDefaults` в виде `Data` (с принудительным развертыванием, чтобы упростить пример). Затем он декодируется с помощью `JSONDecoder` и явно приводится к типу `SomeObject`.

Удаление ключей

Иногда может получиться так, что вы определите некоторый ключ, а в одной из последующих версий приложения решите, что он больше не нужен. Чтобы удалить ключ из `UserDefaults`, достаточно сделать следующий вызов:

```
let defaults = UserDefaults.standard
defaults.removeObject(forKey: "someKey")
```

Теперь, научившись сохранять (и удалять) данные, посмотрим, как читать хранимые предпочтения, чтобы затем можно было использовать их в приложении!

Чтение предпочтений пользователя

Чтение данных из `UserDefaults` выполняется так:

```
let defaults = UserDefaults.standard
let someValue = defaults.value(forKey: "someKey")
```

Этот код создаст объект с именем `someValue`, содержащий наши данные. К сожалению, `UserDefaults` не хранит информации о типе данных, поэтому по умолчанию используется тип `Any?`. Однако, кроме метода `value`, есть еще несколько методов, позволяющих получить данные, приведенные к одному из общих типов. Следующий фрагмент иллюстрирует применение некоторых из этих методов:

```
let defaults = UserDefaults.standard

// Логическое значение
let nightMode = defaults.bool(forKey: "nightMode") // true

// Число
let playbackSpeed = defaults.double(forKey: "playbackSpeed") // 2.0

// Строка
let locale = defaults.string(forKey: "locale") // "en-US"

// URL
let apiURL = defaults.url(forKey: "apiURL") // https://www.example.com/api
```

В этом примере показаны методы: `bool(forKey:)`, возвращающий тип `Boolean`; `double(forKey:)`, возвращающий значение `Double`; `string(forKey:)`, возвращающий строку; `url(forKey:)`, возвращающий экземпляр URL. Есть еще несколько методов, возвращающих значения других типов, таких как `Int` и `Float`. За более полной информацией о типах, которые могут декодироваться механизмом

UserDefaults, обращайтесь к документации Apple для разработчиков (https://oreil.ly/uIDX_).

Но обратите внимание, что в UserDefaults явно отсутствует поддержка нашего типа – класса `SomeObject`, который мы объявили выше в этой главе! Чтобы прочитать сохраненный экземпляр `SomeObject`, нужно использовать метод `object(forKey:)`, например:

```
// SomeObject с поддержкой NSCoder
let someObject = defaults.object(forKey: "someObject") as? SomeObject
```

Обратите внимание, что возвращаемый объект явно приводится к типу `SomeObject`. Это приведение можно опустить, если программа не использует особенностей конкретного типа и ей достаточно типа `Any?`.

Безопасность UserDefaults

Сразу отметим, что UserDefaults *не* является безопасным хранилищем. Данные с настройками по умолчанию хранятся в простом XML-файле. Чтобы обезопасить хранимые данные, лучше использовать встроенный механизм Keychain.

К сожалению, Keychain имеет сложный программный интерфейс и при его использовании легко допустить ошибку. Для устранения этого недостатка было создано несколько сторонних библиотек, значительно упрощающих работу с Keychain. Кроме того, в Apple недавно была создана библиотека `GenericKeychain` (в настоящее время более не поддерживаемая), целью которой было показать, как создать обертку для Keychain. Исходный код этой библиотеки можно найти на портале Apple для разработчиков (<https://oreil.ly/v9LXr>).

Сторонних библиотек-оберток для Keychain слишком много, чтобы рекомендовать какую-то конкретную. Поэтому мы советуем заглянуть в GitHub, выбрать одну из самых популярных библиотек и добавить ее в свой проект.

До сих пор мы говорили только о том, как хранить данные одного пользователя. Теперь поговорим о ситуации, когда приложением пользуется несколько человек, и о том, как управлять их предпочтениями в UserDefaults.

Работа с предпочтениями в многопользовательских приложениях

К сожалению, поддержка нескольких пользователей не имеет стандартной реализации в iOS, несмотря на наличие некоторых похожих особенностей в macOS. Однако эту проблему можно решить, сохраняя настройки разных пользователей в разных файлах. Вот пример, как это реализовать:

```
let defaults = UserDefaults.standard

// Получить текущее содержимое UserDefaults в виде словаря
let dictionary = defaults.dictionaryRepresentation()

// Сохранить словарь на диск
let oldData = try! NSKeyedArchiver.archivedData(
    withRootObject: dictionary, requiringSecureCoding: true)
```

```

try! oldData.write(to: URL(fileURLWithPath: "user1.plist"))

// Удалить все данные
dictionary.keys.forEach { key in
    defaults.removeObject(forKey: key)
}

// Получить предпочтения другого пользователя
let newData = try! Data(contentsOf: URL(fileURLWithPath: "user2.plist"))
if let newDictionary =
    try? NSKeyedUnarchiver.unarchiveTopLevelObjectWithData(newData) as? [String: Any] {
    // добавить новые данные в UserDefaults
    newDictionary.forEach { (keyValue) in
        let (key, value) = keyValue
        defaults.set(value, forKey: key)
    }
}

```

Рассмотрим подробнее, что здесь происходит.

Сначала мы получаем данные из UserDefaults в виде словаря. Этот прием часто используется для резервного копирования данных пользователя. Назовем этого пользователя «User 1». Затем с помощью NSKeyedArchiver преобразуем этот словарь в экземпляр Data, который потом можно записать в папку *Library/Preferences* приложения, но мы оставляем это вам в качестве самостоятельного упражнения. В нашем примере данные сохраняются в пути к файлу *user1.plist*.

Далее код выполняет итерации по ключам внутри словаря и вызывает `removeObject(forKey:)`, чтобы удалить данные из UserDefaults.

Наконец, наступает момент загрузки данных другого пользователя; назовем его «User 2». Предпочтения второго пользователя хранятся в файле *user2.plist*, поэтому мы извлекаем содержимое файла в форме Data, передаем их в NSKeyedUnarchiver, чтобы получить объект словаря `[String: Any]`, того же типа, из которого он был изначально записан. Чтобы добавить данные в UserDefaults, мы перебираем ключи в новом словаре и вручную устанавливаем значения в UserDefaults.

Это не самый простой процесс, но он работает. Вероятно, в будущих версиях iOS появится поддержка нескольких учетных записей пользователей, а пока можно использовать это решение. Возможно, что для записи большого списка пользовательских настроек в UserDefaults может потребоваться некоторое время. К счастью, этот механизм поддерживает работу в многопоточном окружении и всю эту работу можно перенести в фоновый поток.

Что мы узнали

Вот несколько важных уроков, которые мы вынесли из этой главы:

- лучший способ начать сохранять предпочтения пользователя – использовать готовый механизм, предоставляемый операционной системой. На самом деле этот подход обладает очень широкими возможностями, которые наверняка будут востребованы в приложении;

- обе системы, Android и iOS, предлагают схожие подходы к хранению и чтению пользовательских предпочтений. В Android используется механизм `SharedPreferences`, а в iOS – `UserDefaults`. Оба предоставляют хранилище пар ключ/значение, в котором данные сохраняются между сеансами.

В этой главе мы говорили о сохранении данных на устройстве в виде пользовательских предпочтений. Однако существуют другие стандартные форматы, такие как XML и JSON, которые поддерживаются обеими платформами. Давайте познакомимся поближе с дополнительными средствами сериализации данных в следующей главе.

Глава 12

Сериализация и транспорты

Под сериализацией объекта понимается преобразование абстрактной «модели» в переносимую сущность, обычно строковое представление модели, например в формате XML или JSON, либо в байты.

Под десериализацией данных понимается обратное преобразование переносимой сущности в объект, понятный для программы, например в объект типа `Author` со свойством `name`.

Задачи

В этой главе вы узнаете:

- как сериализовать и десериализовать экземпляры объектов.

ANDROID

В Android для внутренних нужд используется формат XML, но в реальном мире основным форматом представления сериализованных объектов по-прежнему является JSON. Следует особо отметить, что многие крупные организации с большим количеством высококвалифицированных разработчиков используют формат `protobuf`, или `Protocol Buffers`, однако обсуждение этой замечательной и высокоэффективной технологии выходит за рамки нашего исследования инфраструктуры и стандартных API.

Сериализация и десериализация экземпляров объектов

В Java и Android десериализация начинается с определения модели данных, например:

Java

```
public class Author {
    private String mName;

    public String getName() {
```

```

    return mName;
}

public void setName(String name) {
    mName = name;
}
}

```

Kotlin

```

class Author {
    var name:String? = null
}

```

Экземпляр объекта `Author`, метод `getName` которого возвращает "Mike", в формате JSON выглядит так:

```
{ name : "Mike" }
```

Данные в формате JSON можно передавать в сетевых запросах, записывать на диск или пересылать другим программам с использованием совершенно разных технологий. Формат JSON имеет устоявшееся определение, поэтому можно быть уверенным, что данные в формате JSON, сформированные нашим приложением в Android, будут выглядеть точно так же, как в приложениях для iOS или даже для Windows или Unix.

На самом деле в Android поддерживается три формата сериализации данных:

- JSON;
- XML;
- стандартный механизм сериализации в Java.

Здесь они перечислены в порядке убывания популярности. Все они поддерживаются стандартной библиотекой Java и операционной системой Android, но мы также рассмотрим сторонние библиотеки, такие как Gson (<https://oreil.ly/RRhOS>). Библиотека Gson была разработана в Google, но кроме нее имеется еще несколько очень популярных альтернатив. Если Gson или `org.json` по каким-то причинам не устроит вас или если вы не очень широко используете формат JSON в своем приложении, вы без труда найдете в интернете более легкие альтернативы.

org.json

Для подробного знакомства с возможностями этой библиотеки мы советуем обратиться к документации для разработчиков (<https://oreil.ly/5Zw0T>).

Сериализация и десериализация с использованием пакета `org.json` реализуются довольно просто. Возьмем в качестве примера класс `Author`, показанный выше. Чтобы сериализовать экземпляр этого класса, нужно создать экземпляр `JSONObject`, скопировать в него свойства из класса `Author` и вызвать метод `toString`:

Java

```

Author author = new Author();
author.setName("Mike");

```

```
// ...
JSONObject jsonObject = new JSONObject();
jsonObject.put("name", author.getName());
Log.d("MyTag", jsonObject.toString());
```

Kotlin

```
val author = Author()
author.name = "Mike"
val jsonObject = JSONObject()
jsonObject.put("name", author.name)
Log.d("MyTag", jsonObject.toString())
```

Этот код выведет {"name": "Mike"}.

Десериализация производится еще проще:

Java

```
String json = "{name:'Mike'}";
JSONObject jsonObject = new JSONObject(json);
Log.d("MyTag", jsonObject.get("name"));
```

Kotlin

```
val json = "{name:'Mike'}"
val jsonObject = JSONObject(json)
Log.d("MyTag", jsonObject.get("name") as String?)
```

Этот код выведет Mike.

Аналогичные средства есть для массивов (списков), правда, при этом придется добавить довольно большое количество преобразований с контролируемыми исключениями, обрабатывающих большие объекты. В Gson подобные преобразования реализуются проще.

Сериализация с использованием Gson реализуется почти так же, кроме отсутствия необходимости использовать промежуточную обертку:

```
Author author = new Author();
author.setName("Mike");
Log.d("MyTag", new Gson().toJson(author));
```

Этот код выведет {"name": "Mike"}.

Не заметили ничего странного? Имя name не совпадает с именем name. Gson по умолчанию использует имена свойств вместо методов доступа.

Этот способ записи – с использованием префикса `m` для переменных-членов и префикса `s` для статических переменных – известен как венгерская нотация. Вы уже видели, что сам Android использует исключительно венгерскую нотацию, поэтому многие разработчики приложений для Android тоже используют этот стиль. Это легко исправить, как будет показано ниже, но имейте в виду, что использование венгерской нотации создает проблемы при применении Gson (и другими подобными библиотеками). Обратите также внимание, что проблема с венгерской (или любой другой) нотацией НЕ является проблемой в Kotlin. Если класс Author определить, как показано в примере на Kotlin, вы получите вполне ожидаемый результат: {"name": "Mike"}.

Но вернемся к примеру на Java. Эта проблема весьма распространена, но она имеет простое решение – достаточно просто снабдить свойства аннотациями `@SerializedName`, в которых указать альтернативные имена. Например, рассмотрим следующее определение класса `Author`:

```
public class Author {
    @SerializedName("name")
    private String mName;

    public String getName() {
        return mName;
    }

    public void setName(String name) {
        mName = name;
    }
}
```

Теперь, попытавшись сериализовать экземпляр этого класса, вы получите `{"name": "Mike"}`. Обратите внимание, что этот прием действует в обоих направлениях: десериализация аннотированных свойств тоже выполняется правильно. Предположим, что мы десериализуем экземпляр класса `Author`, как показано ниже:

```
String json = "{name:'Mike'}";
Author author = new Gson().fromJson(json, Author.class);
Log.d("MyTag", author.getName());
```

Этот код выведет `Mike`, как и ожидалось.

Самое большое преимущество `Gson` в том, что эта библиотека благополучно обрабатывает подобные рекурсивные стратегии, чем не могут похвастаться классы из `org.json`.

org.xmlpull

Другой распространенный формат транспортировки данных – XML. Если вам приходилось разрабатывать веб-приложения, вы наверняка видели и обрабатывали данные в формате XML – если вы видите данные, насыщенные угловыми скобками, это наверняка XML или родственный ему формат.

Стандартные библиотеки Java, доступные в Android, не предлагают никаких встроенных API для работы с XML, однако в Android есть сторонний пакет `org.xmlpull`. Этот пакет содержит объекты, известные как «пассивные парсеры XML» («XML pull parsers»), с помощью которых можно читать и просматривать данные в формате XML. «Пассивный парсер» извлекает элементы из потока по запросу. Существуют также «активные парсеры» («push parser»), которые анализируют элементы по мере их появления в потоке, а не по запросу. Эта семантическая тонкость не имеет большого значения для нас в данном контексте.

Мы постараемся предоставить вашему вниманию максимально упрощенный пример парсинга XML, но даже этот небольшой пример, как вы увидите, является слишком многословным и не достаточно удобочитаемым. Если ваше приложение принимает данные в формате XML и вам нужно их анализиро-

вать, поэкспериментируйте с классами из `org.xml` или поищите более удачную стороннюю альтернативу. Вот сам пример:

Java

```
public class XmlParser {
    public static void parseNodeText() throws Exception {
        // простой фрагмент XML; обратите внимание на узел root (обязателен)
        String xml = "<root><child>A</child><child>B</child></root>";

        // преобразуем этот фрагмент в поток байтов для парсера
        InputStream inputStream = new ByteArrayInputStream(xml.getBytes());
        try {
            // получить экземпляр парсера по умолчанию
            XmlPullParser parser = Xml.newPullParser();
            // в этом простом примере мы не используем пространства имен
            parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
            // определить входной поток и пространство имен (null)
            parser.setInput(inputStream, null);
            // перейти к следующему (первому) тегу
            parser.nextTag();
            // убедиться, что это открывающий тег root
            parser.require(XmlPullParser.START_TAG, null, "root");
            // выполнить обход вложенных элементов,
            // останавливаясь на каждом открывающем теге
            while (parser.next() != XmlPullParser.END_TAG) {
                if (parser.getEventType() != XmlPullParser.START_TAG) {
                    continue;
                }
                // получить имя узла
                String name = parser.getName();
                // если это узел "child", нужно извлечь его содержимое,
                // сделаем это
                if (name.equals("child")) {
                    // сначала нужно выполнить переход от открывающего тега
                    // к содержимому узла
                    parser.next();
                    // записать в журнал
                    Log.d("MyTag", "text=" + parser.getText());
                    // затем извлечь закрывающий тег
                    // и повторить итерацию
                    parser.next();
                }
            }
        } finally {
            inputStream.close();
        }
    }
}
```

Kotlin

```
@Throws(Exception::class)
fun parseNodeText() {
```

```

// простой фрагмент XML; обратите внимание на узел root (обязателен)
val xml = "<root><child>A</child><child>B</child></root>"
// преобразуем этот фрагмент в поток байтов для парсера
val inputStream = ByteArrayInputStream(xml.toByteArray())
try {
    // получить экземпляр парсера по умолчанию
    val parser = Xml.newPullParser()
    // в этом простом примере мы не используем пространства имен
    parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false)
    // определить входной поток и пространство имен (null)
    parser.setInput(inputStream, null)
    // перейти к следующему (первому) тегу
    parser.nextTag()
    // убедиться, что это открывающий тег root
    parser.require(XmlPullParser.START_TAG, null, "root")
    // выполнить обход вложенных элементов,
    // останавливаясь на каждом открывающем теге
    while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.eventType != XmlPullParser.START_TAG) {
            continue
        }
        // получить имя узла
        val name = parser.name
        // если это узел "child", нужно извлечь его содержимое,
        // сделаем это
        if (name == "child") {
            // сначала нужно выполнить переход от открывающего тега
            // к содержимому узла
            parser.next()
            // записать в журнал
            Log.d("MyTag", "text=" + parser.text)
            // затем извлечь закрывающий тег
            // и повторить итерацию
            parser.next()
        }
    }
} finally {
    inputStream.close()
}
}

```

Более подробную информацию о пакете `org.xml` ищите в документации для разработчиков Android (<https://oreil.ly/OEm5A>).

Стандартный механизм сериализации в Java

При беглом знакомстве стандартный механизм сериализации в Java может показаться наиболее простым и понятным. Но в действительности он имеет много минусов – мы кратко рассмотрим их, но это не значит, что данный механизм не имеет практической ценности.

Основная предпосылка проста: преобразовать экземпляр объекта в байты. Эти байты можно сохранить на диск или отправить в сеть и при необходи-

мости собрать из них исходный объект. Однако для этого необходимо, чтобы программа, пытающаяся получить объект, сериализованный таким способом, имела точное определение Java-объекта – виртуальная машина Java (JVM) должна иметь класс, соответствующий полному имени класса объекта и определяющий свойства и методы с именами, совпадающими с именами свойств и методов в исходном классе.

Рассмотрим простой пример: представьте, что вы унаследовали код приложения от стороннего подрядчика. Они использовали класс с именем `Chapter`, имеющий множество свойств и методов, полезных для объекта, представляющего главу в книге. Однако имя пакета (а значит, и полное имя класса – так называемое «каноническое» имя) включало доменное имя подрядчика, и когда вы стали владельцем исходного кода приложения, у вас не было пакета с соответствующим именем. Обычно в этом нет ничего страшного – вы можете создать свой класс `Chapter`, имеющий или не имеющий методы и свойства с именами, аналогичными именам методов и свойств в старом классе `Chapter`. Однако подрядчик нарушил общепринятые соглашения, сериализовав экземпляры `Chapter` в байты и записав их в BLOB-поля в локальной базе данных SQLite. Это означает, что вы *должны* иметь пакет с именем, точно совпадающим с предыдущим каноническим именем, включая домен подрядчика. Когда вы решили выяснить, как работает экземпляр `Chapter` в вашем собственном приложении, то обнаружили, что этот промежуточный класс нужен, только чтобы десериализовать информацию из базы данных, преобразовать в обновленную версию класса `Chapter` и сохранить его свойства в обновленной базе данных. Это не только приводит к путанице, потому что эту историю приходится рассказывать каждому новому разработчику, но также каждый раз, когда вы открываете Android Studio, вы видите этот неиспользуемый пакет подрядчика в верхней части исходного кода, изобилующего определениями классов, которые существуют, только чтобы дать пользователям предыдущей версии приложения перенести свой контент в новое приложение.

А теперь, памятуя о недостатках, посмотрим, как можно использовать стандартную сериализацию Java на практике.

Первое, что нужно знать, – для использования стандартного механизма сериализации в Java сериализуемый класс должен реализовать интерфейс `Serializable`. Этот интерфейс не определяет методов, которые нужно реализовать; ваш класс просто должен объявить, что он поддерживает этот интерфейс.

Также вы должны определить в классе свойство `long serialVersionUID` для хранения идентификатора сериализации, например `private static final long serialVersionUID = 12345467890`; . Здесь можно использовать любое допустимое значение типа `Long`, лишь бы оно было уникальным среди ваших сериализуемых классов. Большинство сред разработки для Java или Android могут автоматически генерировать эти свойства, выбирая правильное имя и значение.

После этого остается только записать объект в файл, используя классы `InputStream` и `OutputStream`, с которыми мы познакомимся в главе 6. Давайте посмотрим, как сериализовать простой класс, используя все тот же класс `Author`, который мы определили выше:

Java

```
// допустим, "myObject" -- допустимый экземпляр
try {
    File file = new File(context.getFilesDir(), "data.obj");
    FileOutputStream fileOutputStream = new FileOutputStream(file);
    ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
    objectOutputStream.writeObject(myObject);
    objectOutputStream.close();
} catch (IOException e) {
    // ошибка
}
```

Kotlin

```
// допустим, "myObject" -- допустимый экземпляр
try {
    val file = File(context.getFilesDir(), "data.obj")
    val fileOutputStream = FileOutputStream(file)
    val objectOutputStream = ObjectOutputStream(fileOutputStream)
    objectOutputStream.writeObject(myObject)
    objectOutputStream.close()
    fileOutputStream.close()
} catch (i: IOException) {
    // ошибка
}
```

Теперь у вас есть файл с именем data.obj, который содержит байты, представляющие экземпляр объекта. Обратите внимание, что это полная копия и ее можно десериализовать обратно в состояние, имевшее место до сериализации, при условии что файл класса находится в CLASSPATH в момент десериализации. Обратное преобразование также выглядит довольно знакомым:

Java

```
try {
    FileInputStream fileInputStream = new FileInputStream("data.obj");
    ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
    Author author = (Author) objectInputStream.readObject();
    objectInputStream.close();
    fileInputStream.close();
} catch (IOException | ClassNotFoundException e) {
    // ошибка
}
```

Kotlin

```
try {
    val fileInputStream = FileInputStream("data.obj")
    val objectInputStream = ObjectInputStream(fileInputStream)
    val author = objectInputStream.readObject() as Author
    objectInputStream.close()
    fileInputStream.close()
} catch (e: IOException) {
    // ошибка
}
```



```

} catch (e: ClassNotFoundException) {
    // ошибка
}

```

Вот и все! Как уже отмечалось, большинство разработчиков предпочитают формат JSON при организации сетевых взаимодействий, но, как было показано выше, в вашем распоряжении имеется много других вариантов. Желающим мы советуем также познакомиться с `protobuffs` – новым способом передачи данных от Google.

iOS

Существует много разных способов преобразования данных в представления, пригодные для транспортировки, и обратно. Обычно это делается для передачи данных в сетевых запросах, и с этой целью наиболее часто используются форматы JSON и XML, причем в последние годы формат JSON пользуется большей популярностью, чем XML. Процесс преобразования между этими двумя форматами и экземплярами объектов в Swift сильно отличается. Существует также дополнительный формат, используемый почти исключительно компанией Apple, – список свойств. Это особая разновидность XML, которую приложения для iOS (и macOS) могут использовать для чтения и записи данных.

Современные версии Swift предлагают намного более удобную и простую реализацию преобразования данных, чем в прошлом, но все же она имеет некоторые сложности. Проще говоря, нам есть что рассказать вам.

Сериализация и десериализация экземпляров объектов

В доисторические времена, когда Objective-C только появился, преобразование объектов в формат JSON отличалось излишней сложностью или требовало использования сторонних библиотек. Даже сразу после появления Swift эта операция была сопряжена с большими трудностями. К счастью, в Apple обратили внимание на потребности разработчиков и уже в версии Swift 3 предложили новый современный подход к сериализации и десериализации JSON.

JSON

Допустим, для начала, что у нас имеется такой объект:

```

struct Author {
    let name: String
}

```

В формате JSON он будет выглядеть, как показано ниже:

```
{ "name": "Mike" }
```

Подобные данные JSON часто требуется посылать и получать при взаимодействии с сервером. Также нередко нужно сохранять данные в этом формате (или любом другом из описываемых в этой книге) в локальном хранилище

и восстанавливать их позже. Преобразование экземпляра описанной выше структуры `Author` в формат JSON будет выглядеть примерно так:

```
struct Author: Codable {
    let name: String
}

let author = Author(name: "Mike")
let rawData = try? JSONEncoder().encode(author)
```

Как видите, здесь мы добавили в объявление нашего объекта протокол `Codable`. Этот протокол является составной частью двух других протоколов: `Encodable` и `Decodable`. Они предлагают набор функций, которые компилятор Swift способен интерпретировать через некоторый синтаксический сахар и ожидаемые значения.

Объекты, поддерживающие протокол `Codable`, должны реализовать методы `encode(to:)` и `init(from:)`. В их отсутствие компилятор Swift попытается применить специальную логику и сгенерировать свои версии этих методов. Для этого он просмотрит свойства объекта и внесет изменения в специальный вложенный тип перечисления `CodingKeys`. Используя это перечисление, компилятор сможет сгенерировать правильные реализации `encode(to:)` и `init(from:)`.

Вы можете назвать это волшебством, но на самом деле компилятор просто принимает прагматичные решения относительно кода.

В нашем примере это проявляется в том, что `JSONEncoder` получает возможность преобразовать объект `author` в представление, пригодное для сохранения в локальном хранилище на устройстве или для отправки на сервер.

Протокол `Codable` также помогает в обратном преобразовании (или «десериализации») объекта из формата JSON. Вот как это выглядит:

```
let rawJson = String("{\"name\":\"Mike\"}").data(using: .utf8)!
let author = try? JSONDecoder().decode(Author.self, from: rawJson)
```

Сначала этот код создает объект `Data` с именем `rawJson`, содержащий строку в формате JSON, которая могла быть получена от сервера. Затем эта строка передается в метод `decode(_:from:)` экземпляра `JSONDecoder`. Этот метод также принимает тип объекта, который требуется получить; в данном случае мы передаем `Author.self` – структуру `Author`, определенную в предыдущем примере. `JSONDecoder` декодирует строку в объект с именем `author`.

i Если экземпляру `JSONDecoder` передать тип объекта, который он не сможет создать из данных в формате JSON, это приведет к ошибке. Мы добавили к вызову метода `try?`, чего достаточно для такого простого примера, но в действующем приложении такие ситуации желательно обрабатывать и выводить сообщения на экран или в журнал.

Протокол `Codable` обладает намного более широкими возможностями, чем было показано, но их обсуждение выходит за рамки этой главы.

А теперь поговорим о другом популярном транспортном формате: XML.

XML

XML – более старый формат, чем JSON. Существует множество разных стандартов XML, в том числе некоторые очень специфические, такие как простой про-

токол доступа к объектам (Simple Object Access Protocol, SOAP), но мы, чтобы не усложнять, будем использовать для демонстрации слегка измененную версию формата JSON, примененного выше. Начнем с такого фрагмента XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<author type="human">
  <name>Mike</name>
</author>
```

Этот фрагмент описывает автора с именем Mike, который оказался человеком (type="human") и, по случайному совпадению, автором этой книги.

Если вы ожидаете такой же простоты, какую предлагает протокол Codable, реализованный в JSONEncoder и JSONDecoder, то у меня есть плохие новости: XMLParser в iOS гораздо более многословен.

Возьмем этот фрагмент XML и создадим из него объект, выполнив парсинг:

```
class SomeObject: NSObject {
  func parseSomeXML() {
    let xml = "<?xml version=\\"1.0\\" encoding=\\"UTF-8\\"?>
      <author type=\\"human\\"><name>Mike</name></author>"
    let rawData = xml.data(using: .utf8)!

    let parser = XMLParser(data: rawData)
    parser.delegate = self;
    parser.parse()
  }
}
```

У нас есть класс SomeObject, наследующий NSObject, который имеет созданный нами метод parseSomeXML. Этот метод определяет строковую переменную с именем xml, содержащую наш фрагмент XML. В следующей строке мы преобразуем его в объект Data в кодировке UTF-8. Затем создаем экземпляр XMLParser с объектом rawData. Назначаем себя в качестве делегата для обработки событий парсинга. И наконец, вызываем parse(), запуская парсинг.

Если попытаться запустить этот код прямо сейчас, он завершится с ошибкой, потому что в настоящее время SomeObject не реализует протокол XMLParserDelegate, который должен быть реализован для правильной обработки событий парсинга. Поэтому давайте подробно рассмотрим каждый метод протокола, чтобы понять суть происходящего.

Парсинг XML происходит синхронно. Документ сканируется, и из него извлекаются элементы один за другим. В нашем делегате мы будем использовать четыре метода:

- 1) parser(_:didStartElement:namespaceURI:qualifiedName:attributes:)
- 2) parser(_:foundCharacters:)
- 3) parser(_:didEndElement:namespaceURI:qualifiedName:)
- 4) parserDidEndDocument(_:)

Вот первый метод из этого списка:

```
func parser(_ parser: XMLParser, didStartElement elementName: String,
           namespaceURI: String?, qualifiedName qName: String?,
           attributes attributeDict: [String : String] = [:]) {
```

```

    if elementName == "author" {
        author = Author()
        if let type = attributeDict["type"] {
            author?.type = type
        }
    }
}

```

Этот метод вызывается каждый раз, парсер обнаруживает в документе новый элемент XML. В нашем предыдущем примере XML единственными допустимыми элементами являются `author` и `name`. После обнаружения нового элемента `author` мы создаем новый экземпляр `Author` в свойстве `author` класса `SomeObject`. Оно играет роль временного хранилища для экземпляра `Author`, который еще предстоит заполнить данными, которые найдем, продолжая парсинг этого элемента.

Первый фрагмент данных – атрибут `type`. Если вы помните, наши авторы характеризуются типом, и в данном конкретном XML-элементе это тип `human`. Мы присваиваем это значение свойству `type` во временном экземпляре `Author`.

Продолжив парсинг документа, мы обнаружим элемент `name`. Этот элемент не требует никакой специальной обработки, мы должны лишь извлечь данные, находящиеся между тегами `<name>` и `</name>`. Эту операцию реализует следующий метод в нашем списке: `parser(_:foundCharacters:)`:

```

func parser(_ parser: XMLParser, foundCharacters string: String) {
    characters += string
}

```

В теле этого метода мы сохраняем все найденные символы в свойстве `characters` нашего экземпляра `SomeObject`. Это свойство действует как временный буфер для хранения символов, обнаруживаемых на каждом следующем шаге, пока парсер не встретит закрывающий тег:

```

func parser(_ parser: XMLParser, didEndElement elementName: String,
            namespaceURI: String?, qualifiedName qName: String?) {
    if elementName == "name" {
        author?.name = characters
    }
    characters = ""
}

```

Данный метод делегата вызывается парсером всякий раз, когда обнаруживается закрывающий тег. В нашем случае это `</name>`. В подобном случае мы переписываем содержимое временного буфера `characters` в свойство `name` нашего объекта `author`.

В заключение, добравшись до конца документа, мы берем получившийся объект и выводим его вызовами `print()`:

```

func parserDidEndDocument(_ parser: XMLParser) {
    print(author.name)
    print(author.type)
}

```

- i** Если бы в нашем XML-документе было больше элементов, мы продолжили бы создавать новые экземпляры авторов и сохраняли бы их перед началом парсинга нового элемента и создания временного объекта.

Вот полный код реализации парсинга, описанной выше:

```
struct Author {
    var name: String?
    var type: String?
}

class SomeObject: NSObject {
    var author: Author?
    var characters: String = ""

    func parseSomeXML() {
        let xml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>
            <author type=\"human\"><name>Mike</name></author>"
        let rawData = xml.data(using: .utf8)!

        let parser = XMLParser(data: rawData)
        parser.delegate = self;
        parser.parse()
    }
}

extension SomeObject: XMLParserDelegate {
    func parser(_ parser: XMLParser, didStartElement elementName: String,
        namespaceURI: String?, qualifiedName qName: String?,
        attributes attributeDict: [String : String] = [:]) {
        if elementName == "author" {
            author = Author()
            if let type = attributeDict["type"] {
                author?.type = type
            }
        }
    }

    func parser(_ parser: XMLParser, foundCharacters string: String) {
        characters += string
    }

    func parser(_ parser: XMLParser, didEndElement elementName: String,
        namespaceURI: String?, qualifiedName qName: String?) {
        if elementName == "name" {
            author?.name = characters
        }
        characters = ""
    }

    func parserDidEndDocument(_ parser: XMLParser) {
        print(author?.name)
        print(author?.type)
    }
}
```

Мы благополучно справились с относительно простым фрагментом XML. А есть ли менее многословные варианты XML?

Списки свойств

Списки свойств, или plist-файлы, – это формат на основе XML, который традиционно использовался для сериализации и десериализации данных в приложениях. Давайте посмотрим, как читать plist-файлы из файловой системы:

```
// Наш файл .plist в файловой системе
let plistURL = URL(...)!
guard let plistData = Data(contentsOf: plistURL) else { return }
let object = PropertyListDecoder().decode(Author.self, from: plistData)
```

Это выглядит очень похоже на пример декодирования JSON. Обратите внимание на использование `PropertyListDecoder`, напоминающего своей функциональностью `JSONDecoder`, но обрабатывающего данные в формате plist, а не JSON.

Запись данных в plist-файл тоже напоминает пример с форматом JSON:

```
let author = Author(name: "Mike")
let data = try? PropertyListEncoder().encode(author)
data?.write(to: ...) // сохранить файл plist на устройстве
```

Этот код запишет данные в формате XML в файл plist. Однако существуют дополнительные варианты экспорта файлов plist. Файлы этого типа можно экспортировать в форматы: XML, двоичное представление и Open Step.

Вот пример экспортирования файла plist в двоичный формат:

```
let author = Author(name: "Mike")
let encoder = PropertyListEncoder()
encoder.outputFormat = .binary // выбрать формат выходных данных
let data = try? encoder.encode(author)
data?.write(to: ...) // сохранить файл plist на устройстве
```

Дополнительные замечания для iOS

Волшебство протокола `Codable`, как оказывается, обусловлено некоторыми действиями компилятора, на которые, как оказывается, можно влиять. Например, ниже показано, как можно использовать другое имя для свойства в JSON, отличное от объявленного в структуре `Author`:

```
struct Author: Codable {
    let name: String

    private enum CodingKeys: String, CodingKey {
        case name = "something"
    }
}

let author = Author(name: "Mike")
let rawData = try? JSONEncoder().encode(author)
```

В этом примере свойство `name` в выходном фрагменте JSON изменяется на `something`. Если запустить этот код, он выведет следующий фрагмент JSON:

```
{ "something" : "Mike" }
```

Кроме того, есть возможность определить свои реализации методов `encode(to:)` и `init(from:)` для протоколов `Encodable` и `Decodable`, выполняющие сериализацию и десериализацию.

Что мы узнали

В Android и iOS используются удивительно похожие подходы к сериализации и десериализации данных в такие форматы, как XML и JSON. Существуют также аналогичные способы сериализации объектов Java, Kotlin или Swift в собственный формат платформы. Несмотря на различия в языке и структурах, обе платформы упрощают передачу данных.

В следующей главе мы затронем приемы расширения базовых объектов инфраструктуры (и других объектов), позволяющие добавлять новые возможности. Давайте посмотрим!

Глава 13

Расширения

Иногда функциональности, предлагаемой системами Android и iOS или сторонними библиотеками, оказывается недостаточно. Конечно, можно создать свои подклассы, но это не всегда возможно.

К счастью, обе платформы имеют средства, позволяющие расширять возможности существующих объектов.

Задачи

В этой главе вы узнаете, как добавлять новые функциональные возможности в существующие API.

ANDROID

Повсюду в этой книге в качестве языка по умолчанию для разработки Android-приложений мы использовали Java. И действительно, подавляющее большинство программ для Android пишется на языке Java. Но теперь мы должны особо отметить, что в Android Framework только Kotlin поддерживает возможность расширения существующих API. Что это значит? Под термином «расширение» в данном случае мы понимаем «изменение существующих или добавление новых функциональных возможностей в любой класс». Однако на языке Kotlin можно писать расширения для классов на Java и успешно пользоваться ими!

Добавление новых возможностей в существующие API

Если в программном коде на Java импортировать стороннюю библиотеку с компонентом `CalendarPicker`, вы фактически будете ограничены общедоступным API, который предлагает эта библиотека. Конечно, можно изменить исходный код компонента, но в результате вы получите компонент `CalendarPicker`, который будет экземпляром совсем другого класса, отличного от оригинала. Также можно определить подкласс `CalendarPicker`, но и в этом случае вы получите совершенно другой класс. В Java нет возможности дополнить или изменить функциональность существующего класса `CalendarPicker`.

В Kotlin дело обстоит иначе. Этот язык программирования позволит не только добавить новый метод в `CalendarPicker`, который будет доступен всем экземплярам `CalendarPicker`, но и расширить возможности классов в стандартной библиотеке, таких как `LinkedList`, и даже обобщенных классов, таких как `HashSet<String>`. На самом деле в Kotlin можно расширить даже класс `Any` и тем самым добавить новые возможности во все его экземпляры. Вот классический пример расширения на Kotlin:

```
fun Any?.toString(): String {
    if (this == null) {
        return "null"
    }
    return toString()
}
```

Этот код на Kotlin добавляет поддержку оператора `?` в *тело* метода `Object.toString`. Включив этот код в свой проект, вы сможете без опаски вызвать метод `toString` для любого объекта; если объект окажется пустой ссылкой `null`, вместо `NullPointerException` вы получите строку `"null"`. И это здорово!

Расширения можно добавлять в любом месте, в любом пакете, где только пожелаете (лишь бы это был файл на Kotlin в каталоге `src`). Например, щелкните правой кнопкой мыши на любом пакете в своем проекте, выберите в контекстном меню пункт `New > Kotlin File/Class (Создать > Файл/Класс на Kotlin)` и дайте файлу любое имя, например «`extensions`» (Android Studio автоматически добавит расширение «`.kt`» и отобразит файл в списке со значком, указывающим, что это файл на Kotlin).

Вы можете добавить в этот файл несколько функций-расширений или создать несколько аналогичных файлов, и все эти расширения будут доступны из любого класса на Kotlin в вашем проекте. Допустим, вы добавили следующий вспомогательный метод в класс `String`:

```
fun String.from(start:String): String {
    val index = indexOf(start) + start.length
    return substring(index)
}
```

После этого в любом классе на Kotlin вы сможете вызвать его и получить ожидаемый результат:

```
val original = "Hello world!"
val modified = original.from("Hello")
Log.d("MyTag", modified)
```

За более подробной информацией о расширениях в Kotlin обращайтесь к документации разработчика (<https://oreil.ly/oGVHb>).

Как работают расширения в Kotlin

Расширения в Kotlin – очень удобный механизм, позволяющий устранить многие препятствия, но имейте в виду, что это всего лишь синтаксический сахар. Вместо расширений с тем же успехом можно использовать вспомогательные классы или традиционные методы, нужно лишь передать экземпляр в параметре.

Например:

```
public class Strings {
    public static String toString(Object object) {
        if (object == null) {
            return "null";
        }
        return object.toString();
    }

    public static String from(String source, String start) {
        int index = source.indexOf(start) + start.length();
        return source.substring(index);
    }
}
```

Этот код обеспечивает ту же функциональность и насчитывает столько же строк, что и примеры расширений выше. Фактически вам просто дается возможность выбрать форму вызова: как метод объекта `myString.from("bob")` или как функцию `Strings.from(myString, "bob")`. На самом деле многие предпочитают последний вариант; на него не влияет состояние объекта, поэтому его легче тестировать и сопровождать. Но не будем углубляться в обсуждение достоинств и недостатков объектно-ориентированного и процедурного стилей программирования – это не является целью нашего обсуждения.

iOS

Одной из основных особенностей языка Swift является возможность добавления новых функций в типы. Фактически эти расширения основаны на особенности среды выполнения языка Objective C, которая называется «категорией» («category»), а не «расширением» («extension»). Эта особенность тесно интегрирована в Swift и широко используется на практике.

Добавление новых возможностей в существующие API

Рассмотрим простой пример:

```
class ExtendableObject {
    // ...
}

extension ExtendableObject {
    func helloTacos() {
        print("Hello, tacos!")
    }
}
```

Здесь сначала определяется класс `ExtendableObject`. В этом примере мы опустили тело класса, потому что оно не имеет значения для нашего обсуждения. Самое важное в данном примере – следующий фрагмент кода, определяющий расширение.

Согласно объявлению `extension ExtendableObject{...}`, наше расширение применяется к классу `ExtendableObject`. В теле этого определения определяется новый метод `helloTacos()`, который просто выводит строку "Hello, tacos!" в консоль. Чтобы воспользоваться этим новым методом, достаточно создать экземпляр объекта и вызвать `helloTacos()`, как любой другой метод, например:

```
let object = ExtendableObject()
object.helloTacos()
```

i Важно отметить, что подобные расширения позволяют добавлять новые методы в существующие типы объектов, но с их помощью нельзя добавить новые хранимые свойства – для этого нужно импортировать среду выполнения Objective C и использовать связанные значения (*associated values*). Однако делать это не рекомендуется, за исключением очень редких случаев.

Использование расширений для организации кода

Поскольку расширения являются неотъемлемой частью языка Swift, в проектах на Swift стало обычной практикой применять расширения для организации кода. Например, этот прием часто используется для размещения реализаций протоколов после определений классов.

Например:

```
protocol AwesomeProtocol {
    func beAwesome()
}

class ExtendableObject { /* ... */ }
extension ExtendableObject: AwesomeProtocol {
    func beAwesome() {
        print("You are awesome!")
    }
}
```

Здесь сначала объявляется новый протокол `AwesomeProtocol` с обязательным методом `beAwesome()`. А затем определяется `ExtendableObject`. Вообще говоря, протокол можно реализовать, объявив его как наследуемый класс в определении `ExtendableObject`:

```
class ExtendableObject: AwesomeProtocol { /* ... */}
```

Однако из-за природы Swift, ориентированной на протоколы, такой подход может стать излишне громоздким и сложным, так как количество протоколов, которые должен реализовать объект, может расти с увеличением сложности приложения. Предыдущий пример с отдельным расширением позволяет определить поддержку протокола и упростить его поиск и поддержку в будущем.

Это совершенно субъективный способ организации кода, но он превратился в стандарт де-факто в сообществе Swift. Вы, в свою очередь, можете сами для себя решить, использовать его или не использовать!

Кстати, коль скоро речь зашла о протоколах в Swift, отметим, что протоколы тоже можно расширять! Давайте посмотрим, как это делается.

Расширение протоколов

Расширение протоколов в Swift осуществляется немного иначе, чем расширение классов и структур. Синтаксис тот же, но механизм, стоящий за ним, отличается. Вот как можно расширить протокол в Swift:

```
protocol ExtendableProtocol {
    func doSomething()
}

extension ExtendableProtocol {
    func printSomething() {
        print("Something :)")
    }
}

class SomeObject: ExtendableProtocol {
    func doSomething() {
        // ...
    }
}
```

i Протокол, расширенный здесь, – это не протокол Objective-C, написанный на Swift, это чистый объект Swift. Это важно понимать, потому что расширить можно только чистые протоколы Swift. Аналогично нельзя расширить методы с префиксом `@objc`.

В этом примере создается протокол `ExtendableProtocol` с методом `doSomething()`. Следующее ниже расширение добавляет в протокол новый метод `printSomething`. Наконец, в самом низу объявляется класс `SomeObject`, реализующий протокол `ExtendableProtocol`.

Вот как можно использовать экземпляр `SomeObject`:

```
let object = SomeObject()
object.doSomething()
object.printSomething()
```

Довольно просто и не слишком отличается от использования расширений классов и структур, разве не так?

Преимущества расширений протоколов становятся особенно очевидными, когда возникает необходимость определить функциональные возможности, общие для разных объектов, не наследующих один и тот же базовый класс. Взгляните:

```
protocol Typeable { /* ... */ }

extension Typeable {
    func printType() {
        let objectType = String(describing: type(of: self))
        print("This object is a type of \(objectType)")
    }
}

class BaseClassA { }
class BaseClassB { }
class BaseClassC { }
```

```
class TacoTruck: BaseClassA, Typeable { }  
class Dog: BaseClassB, Typeable { }  
class Cat: BaseClassC, Typeable { }  
  
let tacoTruck = TacoTruck()  
let dog = Dog()  
let cat = Cat()  
  
tacoTruck.printType()  
dog.printType()  
cat.printType()
```

В этом примере сначала объявляется протокол `Typeable` и расширение для него, которое выводит тип объекта. Затем объявляются три базовых класса, никак не связанных между собой. Далее объявляются три дочерних класса – каждый наследует свой родительский класс, отличный от других, но все они реализуют протокол `Typeable`. После этого создаются экземпляры дочерних классов. И наконец, для каждого из них вызывается один тот же метод `printType`, объявленный в одном месте и функционирующий абсолютно одинаково для всех объектов.

Расширения протоколов – это мощное дополнение, реализованное в Swift, которое помогает писать более чистый, простой и легко читаемый код.

Что мы узнали

В Java нельзя добавлять новые возможности в существующие API, но в Kotlin и Swift имеются встроенные механизмы, позволяющие расширять имеющиеся объекты. Оба подхода имеют свои ограничения, но даже при всех недостатках возможность добавления новых функций нашла широкое применение в обоих языках и платформах.

В следующей главе, завершающей раздел, посвященный решению типичных задач, мы рассмотрим круг вопросов, связанных не с созданием чего-то нового, а с поддержанием того, что уже написано, – с тестированием. Обе платформы имеют развитые средства тестирования. И мы предлагаем немедленно познакомиться с ними!

Глава 14

Тестирование

Разработчики пишут код. Часто они пишут много кода. Сеть зависимостей между объектами и службами нередко сложна и запутана; правильная работа объектов зависит от доступности других объектов. Внесение изменений в одном месте даже относительно простого проекта может вызвать ошибку или сбой в другом месте. Карточный домик вашего приложения может рухнуть очень быстро.

Какое место во всем этом занимает тестирование? Тестирование дает разработчикам уверенность, что вносимые изменения не повлияют на другие части приложения. В идеале тесты должны быть автоматизированными и детерминированными, чтобы исключить возможность неоднозначного толкования, из-за которого часто появляются ошибки. К счастью, большинство современных платформ имеют встроенные средства тестирования. В Android и iOS тоже есть полноценные и невероятно мощные инструменты, которые можно использовать для разработки и тестирования кода. Давайте посмотрим, как они работают.

Задачи

В этой главе вы узнаете:

- 1) как писать и запускать модульные тесты;
- 2) как писать и запускать интеграционные тесты.

ANDROID

AOSP определяет и различает несколько типов тестов:

Модульные тесты

Это узкоспециализированные тесты, проверяющие работу какого-то одного класса. Обычно один тест проверяет единственный метод в этом классе. Если модульный тест потерпел неудачу, вы должны точно знать, где в вашем коде находится проблема. Они охватывают очень узкую область кода, потому что в реальном мире приложение включает в себя гораздо больше, чем простой вызов одного метода. Они должны быть достаточно быстрыми, чтобы их можно было выполнять после каждого изменения кода.

Интеграционные тесты

Эти тесты проверяют взаимодействие нескольких классов, чтобы убедиться, что они работают должным образом при совместном использовании. Часто интеграционные тесты структурируются так, чтобы каждый проверял одну функцию, например возможность сохранить задачу. Они тестируют больший объем кода, чем модульные тесты, но точно так же должны быть быстрыми и максимально точными.

Сквозные тесты

Тестируют комбинации функций, составляющих целые процессы. Они действуют медленно, потому что тестируют значительные части приложения и тщательно имитируют реальное использование. Они имеют самый широкий охват кода и способны подтвердить, что ваше приложение действительно работает, как требуется.

Приемы тестирования в Android, возможно, немного отличаются от приемов тестирования, с которыми вы сталкивались раньше. Классификация тестов и их семантика могут не совпадать с теми, что описываются в других руководствах по программированию.

Начнем с модульного тестирования. Модульное тестирование в Android очень похоже на модульное тестирование в любых других системах или языках. Обычно для создания тестов мы используем фреймворк JUnit, но, вообще говоря, это не является обязательным условием. Модульный тест должен быть узкоспециализированным и точно определять конкретную точку в коде, поэтому ошибки, выявленные модульными тестами, исправляются относительно просто. Например, если у вас есть метод, умножающий два числа и возвращающий результат, модульный тест может вызвать этот метод с некоторыми заранее определенными числами, чтобы убедиться, что результат соответствует ожидаемому. Допустим, ваш класс выглядит так:

Java

```
public class Maths {
    public static int multiply(int a, int b) {
        return a * b;
    }
}
```

Kotlin

```
fun multiply(a:Int, b:Int):Int {
    return a * b
}
```

Вы можете начать с очень простого теста:

Java

```
public class MathsTests {
    @Test
    public void testMultiplication() {
        int a = 2;
```

```
    int b = 3;
    int expected = 6;
    int actual = Maths.multiply(a, b);
    assertEquals(expected, actual);
}
}
```

Kotlin

```
fun testMultiplication() {
    val a = 2
    val b = 3
    val expected = 6
    val actual = Maths.multiply(a, b)
    assertEquals(expected, actual)
}
```

Имейте в виду, что подобные локальные тесты находятся в папке *test*, сгенерированной средой разработки, в родительском каталоге *src*. Инструментированные тесты, которые выполняются на реальном устройстве или в эмуляторе, находятся в подпапке *androidTest*. См. рис. 14.1 и сопровождающий его текст.

Этот тест может показаться простым, но он очень пригодится потом, когда разработчик выполнит рефакторинг, даже не подозревая, что при этом был затронут класс или метод. Тем не менее совершенно понятно, что полезность таких простых тестов имеет свой предел. Даже для такого простого метода, как `Maths.multiply`, может возникнуть масса ситуаций, приводящих к интересным результатам.

Статическая типизация в Java значительно снижает вероятность появления ошибок – поскольку оба параметра имеют тип `int`, нам не придется беспокоиться о значениях `null` или значениях, не уместяющихся в 32-разрядные числа со знаком.

В соответствии с рекомендациями по тестированию для Android модульные тесты также называют «маленькими тестами», и они должны составлять порядка 70 % от общего объема тестов.

Выше в пирамиде тестирования находятся средние и большие тесты. Средние тесты обычно совпадают с так называемыми интеграционными тестами – тестами, проверяющими работу нескольких логических блоков, классов или методов. В соответствии с рекомендациями по тестированию для Android средние тесты должны составлять порядка 20 % от общего объема тестов. Средний, или интеграционный, тест может проверить правильность использования учетных данных при входе в приложение, например: правильную обработку пустых строк, соответствие их ожидаемым шаблонам, таким как адреса электронной почты. Если пользовательские данные хранятся локально на устройстве, интеграционный тест может проверить успешность аутентификации, а в случае использования удаленной аутентификации – успешную отправку HTTP-запроса, а также сам запрос на наличие в нем соответствующих заголовков, тела, шифрования и целевого адреса.

Некоторые интеграционные тесты могут быть инструментированными. Инструментированный тест – это тест, который выполняется на реальном

устройстве или в эмуляторе, получает события ввода, использует системные часы и рисует пиксели на экране.

Библиотека Robolectric (<http://robolectric.org/>) очень популярна среди разработчиков Android и прекрасно подходит для реализации интеграционных тестов. На конференции Google I/O 2018 была представлена новая версия Robolectric 4 в комплексе с фреймворком тестирования AndroidX, заменившим большую часть библиотек Robolectric. Многие детали интеграции этих инструментов скрыты от пользователя и не должны вызывать нарушений в работе существующего тестового кода. Чтобы получить более полное представление об AndroidX и Robolectric, их интеграции и как извлечь максимальную выгоду из каждого инструмента в конкретных условиях, мы советуем обратиться к документации для разработчиков AndroidX Test (<https://oreil.ly/Wnp8A>).

Мы рекомендуем использовать Robolectric 4 и обязательно ознакомиться с документацией к этой версии. Robolectric и AndroidX избавляют от необходимости писать массу шаблонного кода, предлагая свои фиктивные классы фреймворка Android, эмуляцию жизненного цикла Activity и дополнительные возможности, такие как система Shadows, которая позволяет легко смоделировать любой существующий класс и переопределить логику его работы. Например, можно создать класс ShadowThread, который просто запускается в основном потоке при вызове start вместо запуска нового потока выполнения, или ShadowThreadPoolExecutor, который немедленно запускает свою очередь, синхронно и последовательно, помогая тем самым значительно уменьшить сложность тестирования асинхронных приложений.

Для реализации интеграционных тестов с использованием AndroidX и Robolectric необходимо знакомство с некоторыми дополнительными библиотеками. Далее мы покажем несколько примеров, но полное обсуждение приемов тестирования выходит за рамки этой главы. Обязательно загляните в документацию для разработчиков Android (<https://oreil.ly/EpCMX>), если у вас появится желание глубже изучить подходы к тестированию в Android.

И наконец, на самой вершине пирамиды находятся большие, или сквозные, тесты. Обычно все сквозные тесты являются инструментированными и могут охватывать целые «процессы». Например, представьте приложение с функцией создания открытки, которая позволяет захватить изображение с использованием камеры устройства, добавить в изображение различные графические элементы, преобразовать результат в поток данных, выгрузить его на сервер, где пользователь должен пройти аутентификацию, и сохранить изображение на сервере, чтобы потом пользователь смог получить его или поделиться с другими. Сквозной тест может запустить описанный процесс и проверить появление изображения на сервере и его доступность для других.

Инструментированное тестирование иногда называют «тестированием пользовательского интерфейса», но на самом деле интеграционные тесты, выполняемые вне инструментальной среды, тоже можно было бы назвать инструментированными. Как бы то ни было, при создании инструментированных тестов часто полезно прибегнуть к помощи Espresso. Библиотека Espresso предлагает цепочечный синтаксис и функциональную парадигму. Она может имитировать щелчки и прокрутку, а также ввод текста и даже позволяет фиксировать пользовательский ввод и сравнивать фактический вывод с ожидаемым.

Чтобы узнать больше о тестировании пользовательского интерфейса с использованием Espresso, загляните в документацию (<https://oreil.ly/ziFOJ>).

Далее вы узнаете, как писать и выполнять модульные и интеграционные тесты.

Создав новый проект в Android Studio, вы сразу же сможете найти каталоги для тестов. Для этого можно использовать представление Android с раскрывающимся списком каталогов проекта.

Откройте модуль приложения (обычно он имеет имя `app`), затем подкаталог `java`. Внутри него вы найдете как минимум три каталога: каталог с основным исходным кодом и с именем, совпадающим с именем вашего пакета (например, `my.site.appname`), и два каталога с тем же именем, но с дополнительными метками в скобках (`test`) и (`androidTest`), которые отображаются более бледным цветом, как показано на рис. 14.1.

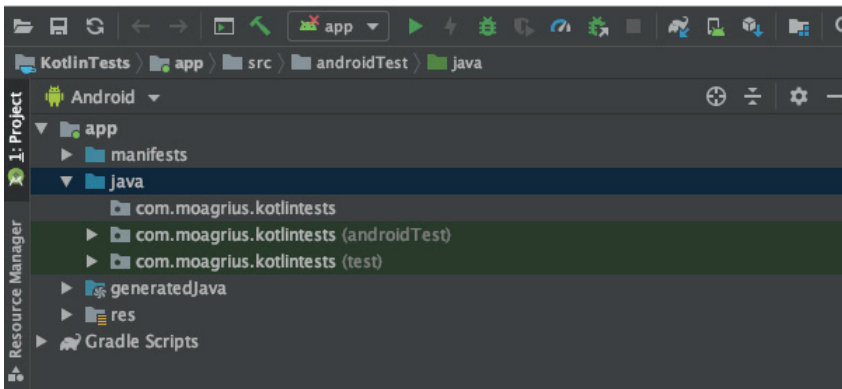


Рис. 14.1 ❖ Каталоги с тестами в Android Studio

В первой папке будет находиться весь исходный код. В папке с меткой (`test`) – модульные и интеграционные тесты. А в последней папке с меткой (`androidTest`) – тесты пользовательского интерфейса на основе Espresso, но о них мы не будем рассказывать в этой главе.

Независимо от выбора языка для проекта, Java или Kotlin, дерево каталогов проекта будет иметь одну и ту же структуру. Если потом вы добавите файлы противоположного типа (например, в проект на Kotlin добавите файлы на Java или наоборот), то обнаружите, что для каждого языка будет создан свой комплект каталогов – по одному для исходного кода, модульных и интеграционных тестов и тестов пользовательского интерфейса.

Как писать и запускать модульные тесты

Все тесты, основанные на JUnit и Robolectric, помещаются в каталог с меткой (`test`), который далее мы будем называть просто «каталогом с тестами».

Открыв эту папку, вы обнаружите в ней готовый пример тестового класса с именем `ExampleUnitTest`. Он импортирует базовые статические методы `assert` из JUnit и может включать какие-то элементарные проверки, например:

Java

```
public class ExampleUnitTest {
    @Test
    public void addition_isCorrect() {
        assertEquals(4, 2 + 2);
    }
}
```

Kotlin

```
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}
```

Это реализация простейшего модульного теста. Каждый модульный тест проверяет одну функциональную единицу кода. В предыдущем примере мало пользы, так как все проверяемые операторы находятся внутри самого теста, поэтому давайте рассмотрим немного более полезный пример:

Java

```
public class Calculator {
    public static add(int a, int b) {
        return a + b;
    }
}
```

Kotlin

```
fun add(a: Int, b: Int): Int {
    return a + b
}
```

Протестировать этот простой класс можно с помощью следующего модульного теста:

Java

```
public class CalculatorTest {
    @Test
    public void add_returnsCorrectValue_forPositiveValues() {
        int expected = 4;
        int actual = Calculator.add(2, 2);
        assertEquals(expected, actual);
    }
}
```

Kotlin

```
class AddTest {
    @Test
    fun add_returnsCorrectValue_forPositiveValues() {
        val expected = 4
    }
}
```

```
    val actual = add(2, 2)
    assertEquals(expected, actual)
  }
}
```

Обратите внимание, что методы `assert` принимают «фактическое» (`actual`) значение, полученное логикой класса, и «ожидаемое» (`expected`), которое должна сгенерировать эта логика. Они сравнивают полученные значения, и если те не совпадают (или не соответствуют результату, предполагаемому конкретными версиями метода `assert`, такими как `assertTrue` или `assertFalse`), то просто генерируют исключения, которые затем перехватываются фреймворком тестирования и помогают определить, какие тесты пройдены успешно, а какие потерпели неудачу.

Модульные тесты для процедурного кода, всегда возвращающего одинаковые результаты в ответ на одни и те же входные данные, почти всегда уместны и полезны. В случае с окружениями, основанными на состоянии (например, в объектно-ориентированном программировании на Java и Kotlin), дело может обстоять несколько сложнее.

Интегрируя свой код в платформу, такую как AOSP, разработчик полагается на определенные возможности, классы и значения. Соответственно, тесты, проверяющие этот код, могут стать более сложными, это особенно верно, когда дело доходит, например, до компонентов `Activity` или событий жизненного цикла. Именно по этой причине была создана библиотека `Robolectric` – она обеспечивает возможность организации взаимодействий с подобными механизмами инфраструктуры простым и понятным способом.

Обратите внимание, что в настоящее время целью развития библиотеки `Robolectric` является ее включение в общий фреймворк `Android Test`, и на последней конференции `Google I/O`, прошедшей перед публикацией этой книги, многие выступавшие отмечали, что поддержку `Android` в `Robolectric` скоро заменит идентичная поддержка из новых пакетов `AndroidX`. То есть привлечение `AndroidX` для тестирования имеет определенную ценность. Использование синтаксиса `Espresso` для реализации локальных и инструментированных тестов пользовательского интерфейса может дать огромные преимущества любой команде, но в то же время некоторые нетривиальные ошибки, обнаруженные нами за год, помешали нашей команде полностью мигрировать на эту библиотеку.

Когда тесты должны интегрироваться с другим кодом, особенно с малоизвестными наборами инструментов, такими как средства рисования пикселей на экране в `Android`, вы действительно можете столкнуться с существенными ограничениями. При написании тестов учитывайте целесообразность трудозатрат. Вам часто будет встречаться термин «охват»; обычно под ним понимается доля кода, охваченного тестированием, но на самом деле существует множество самых разных способов оценки охвата – кто-то подсчитывает количество строк со ссылками, кто-то подсчитывает инструкции и даже ветви условных операторов. Если требуется обеспечить 100%-ный охват для нового продукта, вам (и вашей команде), возможно, придется определить наилучший и наиболее практичный подход к тестированию. В зависимости от используемого у вас определения термина «охват» может оказаться невозможным достичь 100 %

охвата. В предыдущем тесте, проверяющем работу метода `Calculator.add`, можно предположить, что достаточно одного (или нескольких) подобного теста, и едва ли кто-то предложит проверить все возможные комбинации двух целых чисел, *но* у вас может появиться желание проверить, что произойдет при сложении двух целых чисел, сумма которых превышает максимальное целочисленное значение в вашей системе, а также при сложении двух отрицательных или нулевых значений. Как уже говорилось, вы и ваша команда должны определить область тестирования, требования, охват и приемы, и они могут сильно отличаться в разных организациях и даже в разных подразделениях внутри организации.

Как писать и запускать интеграционные тесты

Как отмечалось выше, интеграционные тесты помогают убедиться в правильной работе всего потока логики, а не отдельных логических единиц. Рассмотрим основные моменты на примере. Следующий тестовый класс использует библиотеки `Robolectric` и `AndroidX` для проверки экземпляра `Context` и событий жизненного цикла:

Java

```
@RunWith(AndroidJUnit4.class)
@Config(shadows = {ShadowAsyncTask.class})
public class LoginActivityTest {

    private Activity mActivity;

    @Before
    public void setup() {
        Intents.init();
        ActivityScenario scenario = ActivityScenario.launch(MyActivity.class);
        scenario.moveToState(Lifecycle.State.RESUMED);
        scenario.onActivity(activity -> mActivity = activity);
    }

    @After
    public void teardown() {
        Intents.release();
        mActivity = null;
    }

    @Test
    public void loginActivity_whenLaunched_shouldShowRequiredControls() {
        onView(withId(R.id.credentials)).check(matches(isDisplayed()));
        onView(withId(R.id.submit)).check(matches(isDisplayed()));
    }
}
```

Kotlin

```
@RunWith(AndroidJUnit4::class)
@Config(shadows = { ShadowAsyncTask::class })
class LoginActivityTest {
```

```

private var activity: Activity? = null

@Before
fun setup() {
    Intents.init()
    val scenario = ActivityScenario.launch(MyActivity::class.java)
    scenario.moveToState(Lifecycle.State.RESUMED)
    scenario.onActivity({ a -> activity = a })
}

@After
fun teardown() {
    Intents.release()
    activity = null
}

@Test
fun loginActivity_whenLaunched_shouldShowRequiredControls() {
    onView(withId(R.id.credentials)).check(matches(isDisplayed()))
    onView(withId(R.id.submit)).check(matches(isDisplayed()))
}
}

```

Рассмотрим подробнее этот код:

- 1) аннотация `@RunWith` просто сообщает фреймворку тестирования, что мы используем механизм запуска тестов `JUnit4`;
- 2) аннотацию `@Config` можно использовать для определения разнообразных конфигурационных параметров, и в данном случае мы указываем, что хотим использовать класс `ShadowAsyncTask`. Предположим, что этот класс переопределяет поведение класса `AsyncTask`, который по умолчанию создает для задания новый поток выполнения и просто запускает задание в существующем потоке. То есть любые асинхронные операции, выполняемые с помощью `AsyncTask`, будут выполняться синхронно и последовательно, поэтому проверить их работу будет намного проще;
- 3) аннотация `@Before` перед методом `setup` указывает, что он должен вызываться перед вызовом любого другого метода, осуществляющего тестирование. В нашем методе `setup` мы инициализируем класс `Intents` из `AndroidX`, чтобы с его помощью перехватывать и исследовать полученные намерения и определить, приводит ли нажатие кнопки к отправке намерения для запуска какого-либо `Activity`;
- 4) аннотация `@After` играет противоположную роль – она отмечает метод, который должен вызываться после каждого теста. Здесь мы просто освобождаем экземпляр `Intents` и удаляем экземпляр `Activity`;
- 5) все методы, осуществляющие тестирование, должны отмечаться аннотацией `@Test`. Имя метода может показаться излишне громоздким, но оно следует соглашению «дано – когда – затем», благодаря этому мы можем различать функциональные части, которые на первый взгляд кажутся очень похожими. В этих конкретных тестах мы используем цепочечный синтаксис `Espresso`, чтобы убедиться в видимости пары экземпляров `View`, обеспечивающих получение и отправку учетных данных пользователя.

Обратите внимание, что в предыдущем примере мы используем Espresso API внутри локального, а не инструментированного теста. До появления AndroidX это было невозможно, и весь код Espresso мог выполняться только на фактических устройствах или в эмуляторах.

iOS

В Xcode встроены отличные инструменты поддержки тестирования в iOS. Тесты в iOS подразделяются на две основные категории, модульные тесты и тесты пользовательского интерфейса, причем тестирование пользовательского интерфейса напоминает автоматизированное интеграционное тестирование в зависимости от их настройки и выполнения. А теперь, без лишних слов, углубимся в разработку модульных тестов.

Как писать и запускать модульные тесты

Прежде чем начинать писать и запускать модульные тесты, нужно добавить в проект Xcode новую цель. Для этого выберите в меню приложения пункт **File** ⇒ **New** ⇒ **Target** (Файл ⇒ Создать ⇒ Цель). Добавьте в проект новый пакет модульного тестирования iOS (**iOS Unit Testing Bundle**), чтобы создать цель для модульного тестирования. А затем выполните настройки, следуя инструкциям (для большинства параметров можно оставить значения по умолчанию), и нажмите кнопку **Finish** (Готово) в конце. В результате будет создана новая папка для модульных тестов, и в зависимости от настроек проекта в папку *Products* в левой панели в интерфейсе Xcode будет добавлена новая цель с расширением *.xctest*. Это встроенный пакет, содержащий все ваши модульные тесты и библиотеки; его можно настраивать так же, как цель *.app*, представляющую пакет приложения, то есть добавлять библиотеки, используемые в этом пакете тестов, в том числе и сторонние, упрощающие разработку тестов или предоставляющие другие функциональные возможности.

После добавления цели тестирования в нее можно добавить новый набор тестов, для этого выберите в меню пункт **File** ⇒ **New** ⇒ **File** (Файл ⇒ Создать ⇒ Файл), в открывшемся диалоге выберите **Unit Test Case Class** (Класс модульного теста) и дайте имя классу и файлу. Обычно в Xcode файлам и классам тестов даются имена, включающие имя тестируемого объекта. Это не жесткое правило, но чаще соблюдается в большинстве проектов. Например, набор тестов для класса с именем *Calculator* можно назвать как *CalculatorTests*.

В сфере тестирования в Xcode широко используется принцип «соглашения по конфигурации», и вы будете сталкиваться с ним снова и снова, добавляя тесты для запуска. Давайте разберем, в чем заключается его суть.

Откроем файл *CalculatorTests.swift* с нашим гипотетическим классом *CalculatorTests* и добавим в этот класс новый метод с именем, начинающимся со слова *test*. Например:

```
func testExample() {  
    ...  
}
```

В соответствии с соглашениями Xcode автоматически распознает тесты в файле, если их имена начинаются с `test`. Чтобы запустить набор тестов, перейдите в **Product** ⇒ **Test** (Продукт ⇒ Тест). В результате рядом с тестом в XCode появится зеленая галочка, подсказывающая, что тест выполнен успешно.

Теперь рассмотрим весь класс с тестами. Прямо сейчас `CalculatorTests` выглядит так:

```
class CalculatorTests: XCTestCase {
    override func setUp() {
    }
    override func tearDown() {
    }
    func testExample() {
    }
}
```

Обратите внимание на несколько важных аспектов. Прежде всего класс `CalculatorTests`, вмещающий все тесты для класса `Calculator`, наследует `XCTestCase`. В нем определен тест `testExample`, который будет запущен и должен выполнить некоторые проверки, каждая из которых может завершиться успехом или неудачей.

Есть несколько стандартных проверок, таких как:

- `XCTAssert()` – принимает выражение и возвращает логическое значение;
- `XCTAssertFalse()` и `XCTAssertTrue()` – проверяют конкретное логическое значение;
- `XCTAssertEquals()` и `XCTAssertNotEqual()` – проверяют равенство объектов;
- `XCTAssertNil()` и `XCTAssertNotNil()` – проверяют, является ли ссылка пустой (`nil`).

Чтобы выполнить эти проверки, достаточно просто добавить их в тело метода:

```
func testExample() {
    let success = false
    XCTAssertTrue(success)
}
```

Эта проверка приведет к неудачному завершению теста, потому что в предыдущей строке параметру `success` присваивается значение `false`. Когда проверка терпит неудачу, тест считается непройденным. Проверки довольно просты в использовании, и в одном тесте можно использовать несколько проверок.

В тестовом классе также есть методы `setUp()` и `tearDown()`. Это стандартные методы, которые вызываются до и после запуска каждого теста в этом классе соответственно. Как вы уже наверняка догадались, `setUp()` реализует настройку теста и испытуемого объекта, а `tearDown()` освобождает использованные ресурсы. Например, в `setUp()` мы могли бы создать экземпляр `Calculator` для тестирования и сохранить его в переменной `sut`, а в `tearDown()` уничтожить его после тестирования. Например:


```

class CalculatorTests: XCTestCase {
    var sut: Calculator!

    override func setUp() {
        self.setUp()
        sut = Calculator()
    }

    override func tearDown() {
        sut = nil
        self.tearDown()
    }

    func testTwoPlusTwoEqualsFour() {
        let result = sut.enter(2).add(2)
        XCTAssertEqual(result, 4)
    }
}

```

Иногда требуется протестировать асинхронные функции. Это сложно, потому что тестовый метод будет завершаться до того, как выполнится асинхронный код. Асинхронный код вообще очень сложно отлаживать. Однако механизм ожиданий в фреймворке тестирования избавляет нас от этой проблемы. Например:

```

func testAsynchronousCode() {
    let expectation = XCTestExpectation(description:
        "Asynchronous code will return true.")

    sut.enter(2).add(2).shareToTwitter { (success) in
        if success {
            expectation.fulfill()
        } else {
            XCTFail("Sharing to Twitter did not work.")
        }
    }

    waitForExpectations(timeout: 2.0) { (error) in
        XCTFail("Test failed.")
    }
}

```

В этом примере сначала создается экземпляр ожидания `XCTestExpectation` для выполнения в асинхронном коде. Затем фиктивному методу `shareToTwitter(:)` передается замыкание, которое выполнится после публикации результатов вычислений калькулятора в Twitter. Потом проверяется значение `success` и, если оно равно `true`, вызывается метод `fulfill()` экземпляра ожидания, чтобы сообщить Xcode, что тест пройден, иначе вызывается `XCTFail()`, чтобы сообщить, что произошла ошибка. В заключение вызывается `waitForExpectations(timeout:)`, чтобы дождаться завершения ожидания, установив тайм-аут равным двум секундам. Если к тому времени ожидание не выполнится, будет вызван `XCTFail()`, который сообщит, что тест не пройден.

Мы надеемся, что модульное тестирование в Xcode и iOS не выглядит таким страшным, каким могло бы вам показаться. Теперь перейдем к интегра-

ционному тестированию, то есть тестированию пользовательского интерфейса в iOS.

Что мы узнали

В этой главе мы увидели, что тестирование в Android и iOS осуществляется с использованием развитых систем, способных быстро, последовательно и безболезненно проверять утверждения. Многие разработчики, к сожалению, недооценивают пользу модульных тестов, которые действительно могут ускорить разработку за счет быстрого обнаружения ошибок, добавленных при внесении изменений в код.

А теперь, после знакомства с различными технологиями Android и iOS, мы предлагаем повеселиться. Давайте напишем приложение!

Часть II

ПРИМЕР ПРИЛОЖЕНИЯ

Во второй половине книги, начинающейся здесь, мы создадим полноценное приложение на обеих платформах, используя примеры решения задач из первой части. Здесь вы увидите, как на практике используется теоретический код из первой части.

Мы стараемся придерживаться передовых практик и рекомендаций, но не забывайте, что эта книга по своей сути является справочником. Кое-где мы можем выбирать более общие шаблоны, чем принято в программировании для той или иной платформы, чтобы показать сходство. Мы не утверждаем, что представленные здесь приложения запрограммированы особенно эффективно или оптимально, мы хотели лишь на их примерах показать обобщенные методы.

Приступим!

Глава 15

Добро пожаловать и настройка окружения

СРАВНЕНИЕ НАТИВНЫХ И КРОСС-ПЛАТФОРМЕННЫХ ИНСТРУМЕНТОВ РАЗРАБОТКИ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

Для пушей ясности познакомимся с некоторыми фактами и сделаем некоторые начальные выводы.

Как уже отмечалось во вступлении, под «нативной разработкой» мы подразумеваем программирование с использованием языков, фреймворков и сред разработки, поддерживаемых и продвигаемых создателями каждой платформы. В данном случае – Android и iOS.

Развитием и продвижением Android занимается Google. Мы не говорим, что Google является «владельцем», потому что в действительности это открытое программное обеспечение – фактически на основе Android компания Amazon создала свою ОС FireOS. Иногда мы ссылаемся на Android, используя аббревиатуру AOSP (Android Open Source Project). Для программирования приложений в Android используется Java, Kotlin или их смесь (Kotlin генерирует байт-код Java; с точки зрения среды времени выполнения между ними нет никакой разницы). С технической точки зрения приложение для Android можно написать на любом языке, который генерирует байт-код Java, – один из наших друзей однажды написал приложение для Android на Scala. Тем не менее в этой книге мы используем только Java и Kotlin. Фреймворк не имеет конкретного имени, и обычно его называют просто «фреймворк Android».

Когда мы говорим «нативная разработка», мы *не* имеем в виду разработку *в рамках* фреймворка, что можно было бы интерпретировать как использование предоставляемых системой функциональных возможностей, таких как системные часы (что очень важно на самом деле, как это ни удивительно), реализацию потоков выполнения или файловую систему. Мы также не имеем в виду использование комплекта инструментов Android Native Development Kit (он же Android NDK), который позволяет вызывать код, написанный на C или C++, из Activity или Service.

А теперь, определив предпосылки, сравним некоторые кросс-платформенные инструменты. Существует много разных продуктов для кросс-платформенной разработки, но все они используют лишь несколько подходов.

Веб-разработка

Такие продукты, как PhoneGap, позволяют разработчику писать код на HTML, CSS и JavaScript и создавать веб-страницы, которые отображаются с использованием веб-компонентов на каждой платформе. С одной стороны, писать такие приложения намного проще, в том смысле, что развить и сопровождать приходится только одну базу кода (в теории) и используемые при этом технологии и языки хорошо известны и неплохо поддерживаются. Однако в реальности крайне редко удается оставаться в рамках единой базы кода – отличия между платформами требуют использования массы условной логики. Кроме того, довольно сложно одновременно следить за обновлениями или дополнениями в каждой платформе.

Другие подходы

Существует еще ряд продуктов, целью которых является упрощение кросс-платформенной разработки. В Google имеется инструмент под названием Flutter, который использует язык Dart и свое собственное ядро для создания приложений Android и iOS на основе одной и той же базы кода. И поскольку в нем не используются существующие фреймворки, он способен запускать приложения с удвоенной частотой кадров – 120 кадров в секунду вместо 60. На достаточно мощных устройствах прокрутка может ощущаться практически на *физическом уровне* – как будто вы действительно перетаскиваете лист бумаги по гладкой поверхности. Тем не менее с помощью Flutter практически невозможно создать единую унифицированную базу кода для обеих платформ – пользователи Android ожидают увидеть другие виджеты, реакцию на ввод, переходы и т. д., нежели пользователи iOS, и наоборот. Точно так же, если не использовать преимущества платформ Android или Cocoa, уже установленных на устройстве, размер приложения, созданного с помощью Flutter, почти всегда будет значительно больше, чем размер приложения, созданного с использованием нативных инструментов.

Еще один подход, впервые предложенный в Facebook, – фреймворк React Native. Для создания приложений для обеих платформ из единой базы кода в React Native используется свой диалект JavaScript, разработанный в Facebook. Помимо некоторых недостатков, из упомянутых выше, использование этого подхода сопряжено с еще одной проблемой – вы отдаете контроль над своим приложением совершенно не связанной с вами третьей стороне (Facebook). Если в Facebook пожелают начать собирать информацию о работе вашего приложения, они смогут сделать это совершенно незаметно для вас как для разработчика, и вы ничего не сможете с этим поделать. Мы не утверждаем, что они делают или когда-либо будут делать что-то подобное; это просто пример одной из проблем, характерных для кросс-платформенных инструментов, которые могут быть незаметны на первый взгляд.

Есть и другие инструменты, но мы как авторы этой книги не хотим тратить ваше и наше время на обзор удивительных результатов труда других разработчиков. Мы просто хотим, чтобы вы поняли, почему мы искренне верим, что параллельная разработка кода для каждой платформы в отдельности – с исполь-

зованием языка и нативных инструментов для каждой платформы – может не только обеспечить огромную выгоду, но также требует вдвое больше времени, средств и внимания. На протяжении всей книги мы пытались показать, что между этими двумя платформами действительно много общего, и разработка отдельного приложения для каждой платформы с использованием нативных инструментов не является настолько сложной задачей, как могло бы показаться.

НАСТРОЙКА ОКРУЖЕНИЯ

Чтобы начать разработку приложений для Android и iOS, сначала нужно настроить окружение. К счастью, ничто не мешает установить Android Studio и XCode на один компьютер. Android Studio поддерживает несколько платформ, но, поскольку для разработки под iOS мы используем Xcode – интегрированную среду разработки от Apple, – нам понадобится компьютер с установленной macOS. Давайте сначала настроим окружение разработки для Android.

Настройка окружения разработки для Android

Для разработки приложений для Android мы используем исключительно Android Studio. Эта среда разработки поддерживается и рекомендуется компанией Google, занимающейся развитием Android. Технически разрабатывать приложения можно и без Android Studio, но это довольно сложно, и, вероятно, такой подход оправдан только в очень специфических сценариях (мы видели, как это делается, и нам это не понравилось). Поэтому предположим, что вы решили не усложнять себе жизнь и выбрали Android Studio.

Давайте посмотрим, как установить и настроить эту среду. Установку Android Studio мы завершим созданием нового проекта, поэтому опишем весь процесс в одном разделе.

Установка Android Studio

Одно время установка и настройка Android Studio были сопряжены с большим числом трудностей и предполагали выполнение множества шагов. Однако теперь этот процесс стал почти тривиальным. Более того, сейчас в состав Android Studio входит даже свой пакет JDK (Java Development Kit), так что вся работа ограничивается лишь загрузкой дистрибутива и его установкой.

На момент написания этой книги Android Studio можно было загрузить по адресу <https://developer.android.com/studio>, но вы можете также просто выполнить поиск в интернете по фразе «Android Studio скачать», если к тому моменту, когда вы будете читать эти строки, приведенный нами URL окажется недействительным.

Загрузив дистрибутив Android Studio, запустите его! Далее мы представим последовательность шагов установки, имевших место на момент написания этой книги, летом 2019 года. Эта последовательность может измениться в будущем, но, как мы надеемся, сам процесс останется прежним. Если вы увидите отличия от того, что показываем мы на скриншотах, выбирайте решения, которые считаете правильными.

Программа может предложить импортировать настройки из предыдущей установки; если вы устанавливаете среду в первый раз, пропустите шаг импорта и переходите к рис. 15.1.

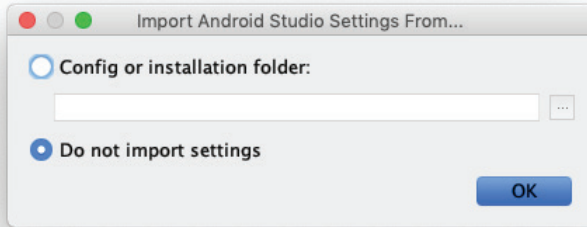


Рис. 15.1 ❖ Диалог импорта настроек

После этого Android Studio должна начать процесс инициализации. Следите за полосой прогресса. Затем появится диалог приветствия, как показано на рис. 15.2.

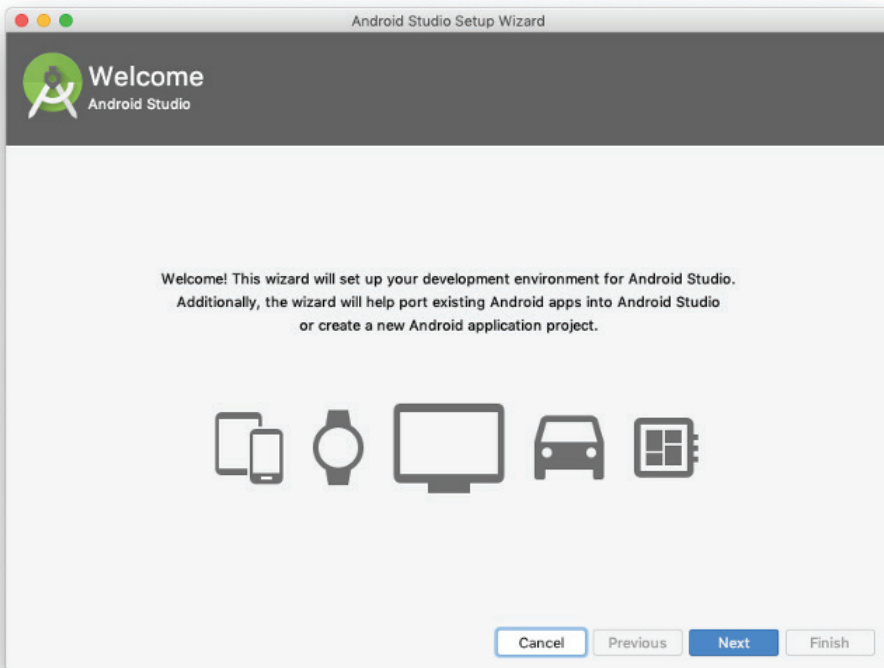


Рис. 15.2 ❖ Диалог приветствия

Далее вам будет предложено выбрать тип установки, как показано на рис. 15.3.

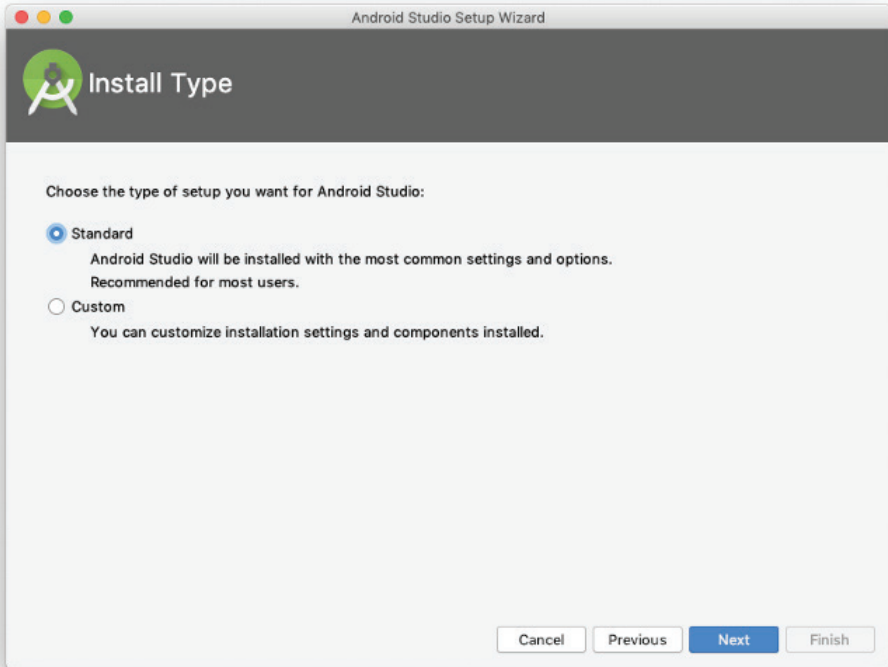


Рис. 15.3 ❖ Выбор типа установки

Прямо сейчас выберите вариант **Standard** (Стандартная). Если понадобится, дополнительные настройки можно будет выполнить потом.

После этого программа установки может предложить выбрать между светлой и темной темами оформления IDE, как показано на рис. 15.4.

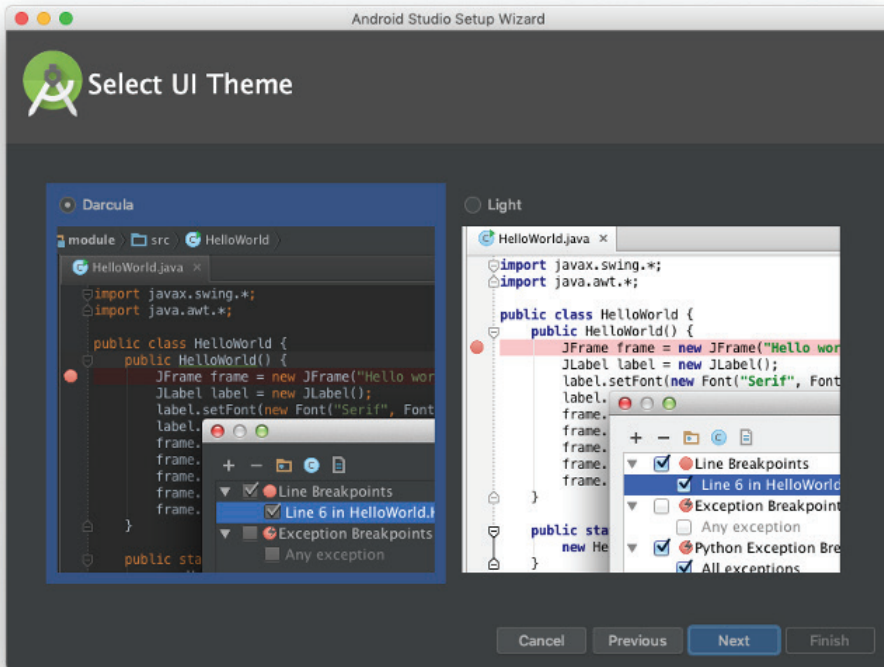


Рис. 15.4 ❖ Выбор темы оформления

Затем вам будет предложено подтвердить выбранные параметры установки, как показано на рис. 15.5.

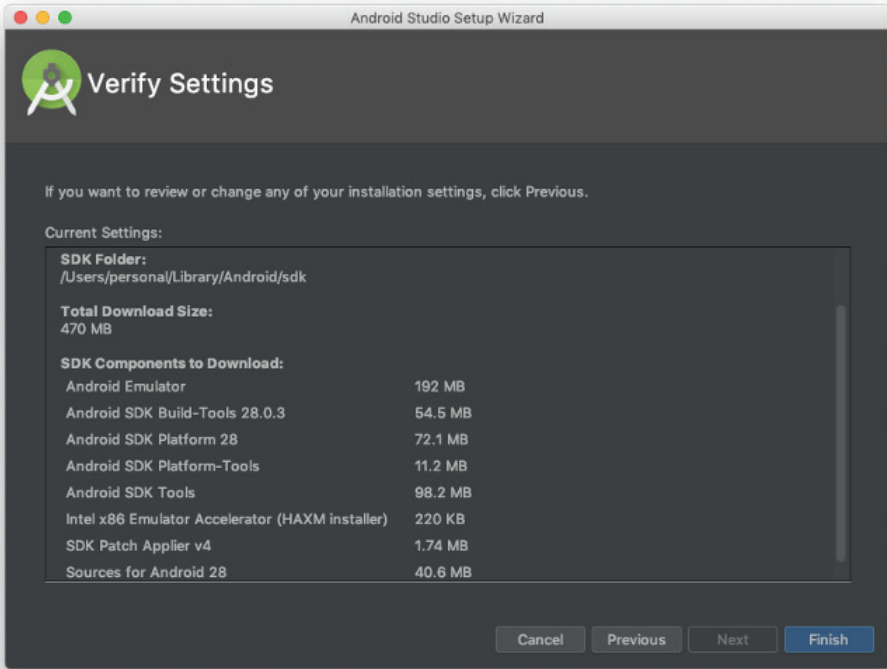


Рис. 15.5 ❖ Диалог подтверждения выбранных параметров установки

Щелкните на кнопке **Finish** (Готово), если вас все устраивает, и вы увидите, как Android Studio начнет подключаться к разным серверам для обновления своих компонентов, как показано на рис. 15.6.

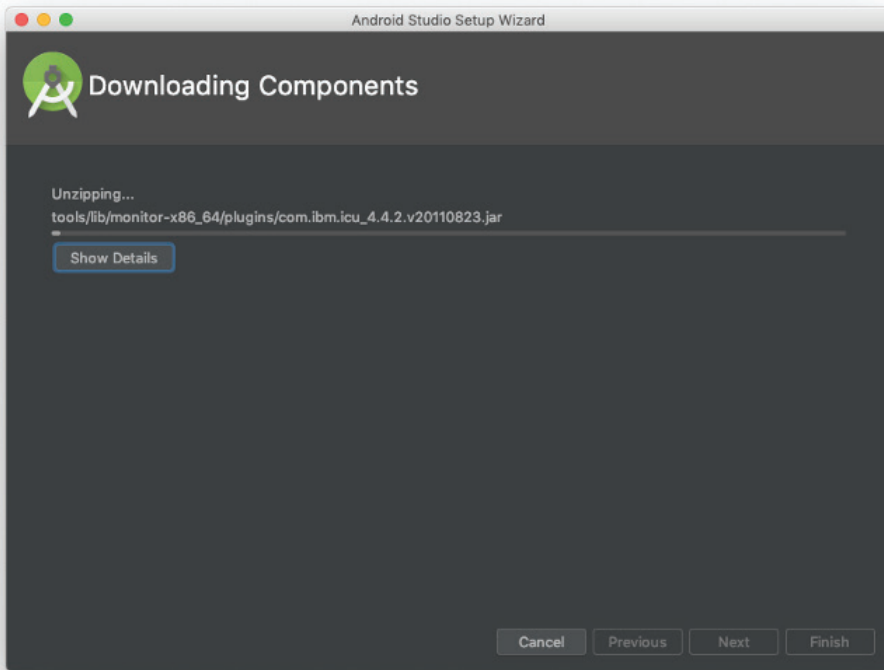


Рис. 15.6 ❖ Установка компонентов

Чтобы включить поддержку Java 8, которая используется в этой книге, добавьте следующие строки в файл *build.gradle* на уровне модуля (этот код мы взяли из документации разработчика Android):

```
android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }

    // Для проектов на Kotlin
    kotlinOptions {
        jvmTarget = "1.8"
    }
}
```

Выполнив этот шаг, вы сразу попадете в диалог создания нового проекта. Он проведет вас через процесс, который вы не раз будете использовать для создания следующих и последующих проектов в Android Studio, однако мы рассмотрим этот процесс в главе 16.

На данный момент ваша среда разработки настроена и готова к использованию!

Настройка окружения разработки для iOS

Xcode – это среда разработки, созданная и поддерживаемая компанией Apple. Она используется для разработки приложений, библиотек и проектов для iOS (и macOS). Это приложение обладает очень широкими возможностями и уже давно существует в различных формах. Сама среда Xcode успешно миновала переход от настольных ПК с PowerPC к компьютерам с процессором Intel в середине 2000-х годов и затем, с началом выпуска iPhone, на устройства с процессором ARM и операционной системой iOS. Основным языком программирования в те времена был Objective-C и только-только начал появляться Swift. Это удивительное программное обеспечение, которое будет радовать и разочаровывать вас до бесконечности.

Установка Xcode, к счастью, является одной из самых замечательных черт этой среды.

Установка и настройка Xcode

Чтобы установить Xcode, перейдите в магазин приложений macOS App Store, установленный по умолчанию на каждом компьютере с macOS. Найдите Xcode и нажмите кнопку **Install** (Установить), чтобы установить эту среду на свой компьютер.

Работая над этой книгой, мы использовали Xcode 11 и Swift 5. Версия Xcode 11 была выпущена в сентябре 2019 года. Вообще говоря, Xcode мало меняется между версиями, поэтому, если вы используете более старую или более новую версию Xcode, возможно, вам придется искать команды, имена которых могли измениться, но сам процесс настройки принципиально не изменится.

Как вы запустите Xcode в первый раз, вам будет предложено установить дополнительные компоненты, добавляющие инструменты командной строки. Мы настоятельно советуем вам установить эти инструменты, чтобы упростить себе жизнь в будущем.

Учетная запись Apple Developer

При желании вы можете создать учетную запись и заплатить 99 долларов США за участие в программе Apple Developer Program. Однако это необходимо, только если вы планируете распространять свои приложения. Если вы просто хотите попрактиковаться в разработке приложений, вам не нужно платить за это. Вы сможете создать приложение и установить его на свое устройство, если подключите его к компьютеру.

Что мы узнали

В этой вводной главе мы обсудили некоторые основополагающие вопросы. По-знакомимся с причинами, почему мы предпочитаем использовать нативные инструменты разработки вместо кросс-платформенных. Время, сэкономленное на разработке общей базы кода, часто теряется из-за необходимости тратить дополнительное время на разработку обходных решений, не предусмотренных поставщиком, отдельного кода для обработки различий в поведении пользовательского интерфейса и на ожидание поддержки сторонними библиотеками новых возможностей платформ.

Мы также потратили некоторое время на настройку и подготовку каждой среды разработки. Установили Android Studio и Xcode на наши машины. Теперь мы готовы приступить к созданию нашего первого приложения. Но сначала, в следующей главе, немного поговорим о том, что мы будем создавать.

Глава 16

Создание приложения

Представьте, что вы входите через широкие дубовые двери в Мемориальную библиотеку Данна и Льюиса в поисках знаний. Войдя внутрь, вы видите безбрежное море деревянных стеллажей, количеству которых может позавидовать даже Великая Александрийская библиотека. Вы переходите от стеллажа к стеллажу, но понятия не имеете, какие книги на них стоят и где искать нужные вам. Разочарованный и одинокий, вы собираетесь навсегда покинуть библиотеку, когда вдруг старый библиотекарь подзывает вас.

Библиотекарь пахнет знакомым запахом старых книг и красного дерева. Вы направляетесь к нему, но прежде чем успеваете подойти вплотную, он указывает на объявление, висящее на стене, в котором сообщается, что для загрузки вам доступно приложение, которое поможет найти нужные книги. Как оказывается, вам совсем не нужно наобум блуждать между стеллажами!

Но где находится это приложение? И тут библиотекарь вполголоса сообщает, что его пока не существует и вы должны его создать, после чего он исчезает во тьме коридора.

Что ж, займемся созданием этого приложения.

Теперь забудем на время о нашем библиотекаре. В главе 15 мы увидели, как установить и настроить среду разработки и создать самое простое из возможных приложений. Однако большинство приложений намного сложнее и состоят больше чем из одного экрана. Чтобы по-настоящему изучить платформу, необходимо создать что-то достаточно сложное – сложнее, чем простейшее приложение «hello world», – чтобы понять нюансы и возможности используемых технологий. С этой целью мы создадим приложение для библиотеки – да, то самое приложение, на которое сослался наш библиотекарь – чтобы помочь посетителям Мемориальной библиотеки Данна и Льюиса найти нужные им книги.

В этой главе вы узнаете:

- 1) как создать новый проект приложения;
- 2) как произвести сборку приложения;
- 3) как добавить в приложение простой экран с приветствием.

Давайте начнем поскорее, пока не взошла кровавая луна!

СОЗДАНИЕ НОВОГО ПРОЕКТА

У вас уже должна иметься настроенная среда разработки. Если это не так, вернитесь к предыдущей главе и подготовьте свою среду разработки для Android

и iOS. А если у вас все готово, приступим к работе и сначала погрузимся в Android.

Android Studio

Если вы не переходите к процедуре создания нового проекта сразу после установки, вы всегда можете запустить ее, выбрав в меню пункт **File** ⇒ **New Project** (Файл ⇒ Новый проект).

Android Studio предложит на выбор один из базовых шаблонов проектов, таких как **Basic Activity** (Простой Activity) или **Empty Activity** (Пустой Activity), или более продвинутые, такие как **Java Library** (Библиотека Java) или **Android Library** (Библиотека Android). Возможно, позже у вас появится желание попробовать создать эти проекты, но сейчас просто выберите шаблон **Empty Activity**, как показано на рис. 16.1.

Первый этап в процессе создания проекта – его настройка. На этом этапе вы должны определить местоположение приложения на диске, пространство имен и минимальный API, который ваше приложение будет поддерживать. Этот последний шаг на самом деле очень важен. Узнать номера дистрибутивов для различных версий Android можно по адресу: <https://oreil.ly/3h0X1>.

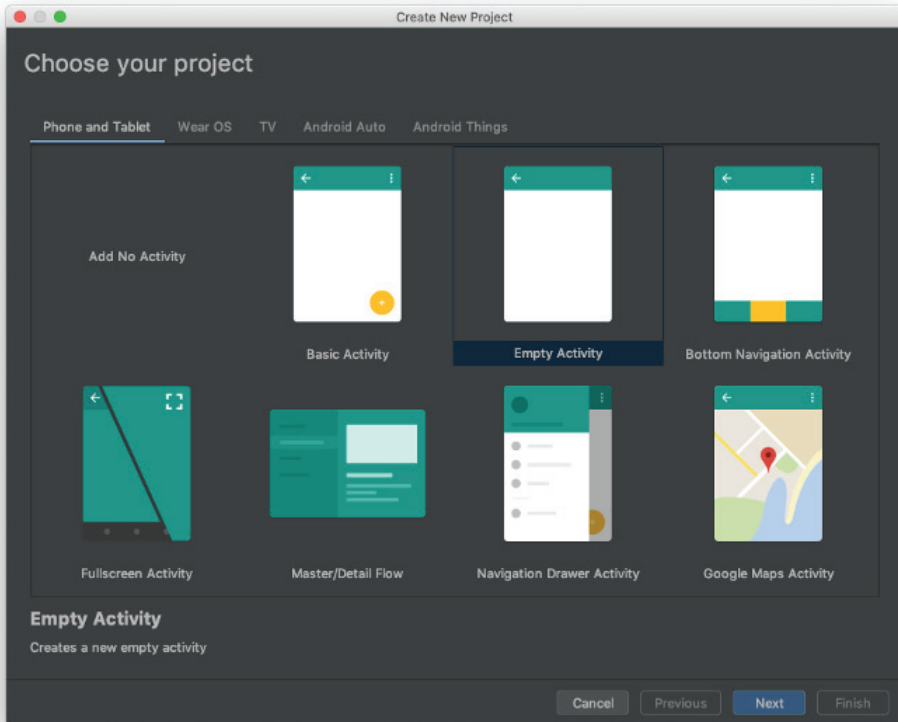


Рис. 16.1 ❖ Выбор типа проекта

На момент написания этой книги вполне безопасным выглядел выбор OS 19 с номером версии 4.4 под кодовым названием KitKat. На текущий момент эту версию имеет большинство установок. Однако если вы ориентируетесь на более продвинутых технически пользователей или на жителей богатых стран, можно выбрать OS 20 с номером версии 5.0 под кодовым названием Lollipop. Эта версия установлена примерно у 7 % пользователей во всем мире, но данный выбор заметно облегчит разработку. Android 5 шагнул далеко вперед и является линией водораздела для многих современных API. Вы можете принять любое решение, но для этого простого проекта мы возьмем за основу Kitkat, как показано на рис. 16.2.

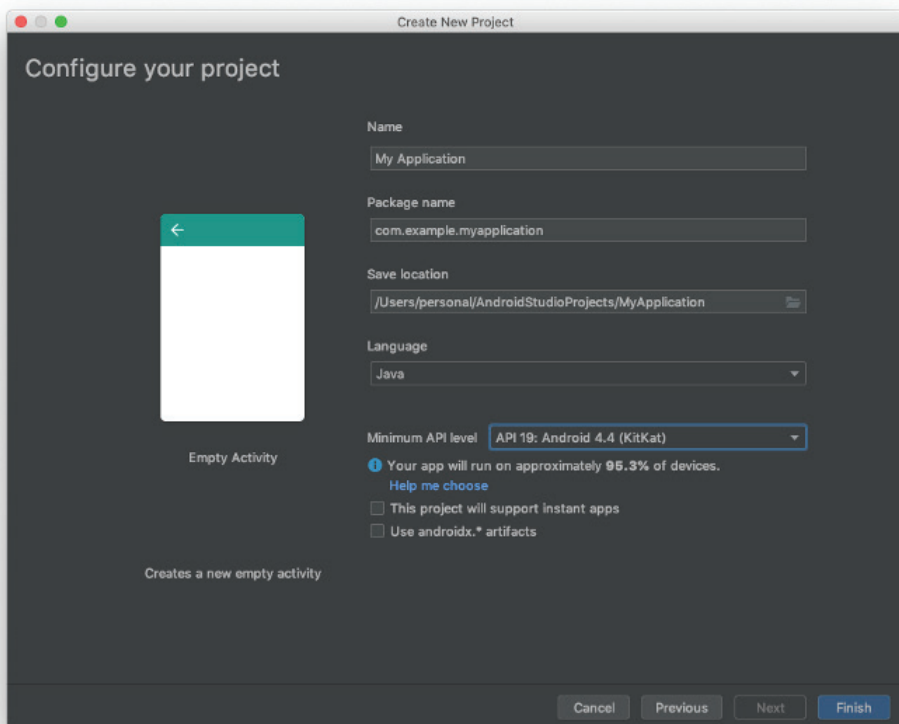


Рис. 16.2 ❖ Настройка проекта

Вот и все! Теперь запустите проект, щелкнув на кнопке **Play** (Запустить) в панели инструментов, или нажав комбинацию **Control/Command + R**, или выбрав в меню пункт **Run** ⇒ **Run App** (Запустить ⇒ Запустить приложение).

Когда это действие выполняется в первый раз, вам будет предложено подключить физическое устройство или запустить эмулятор, как показано на рис. 16.3.

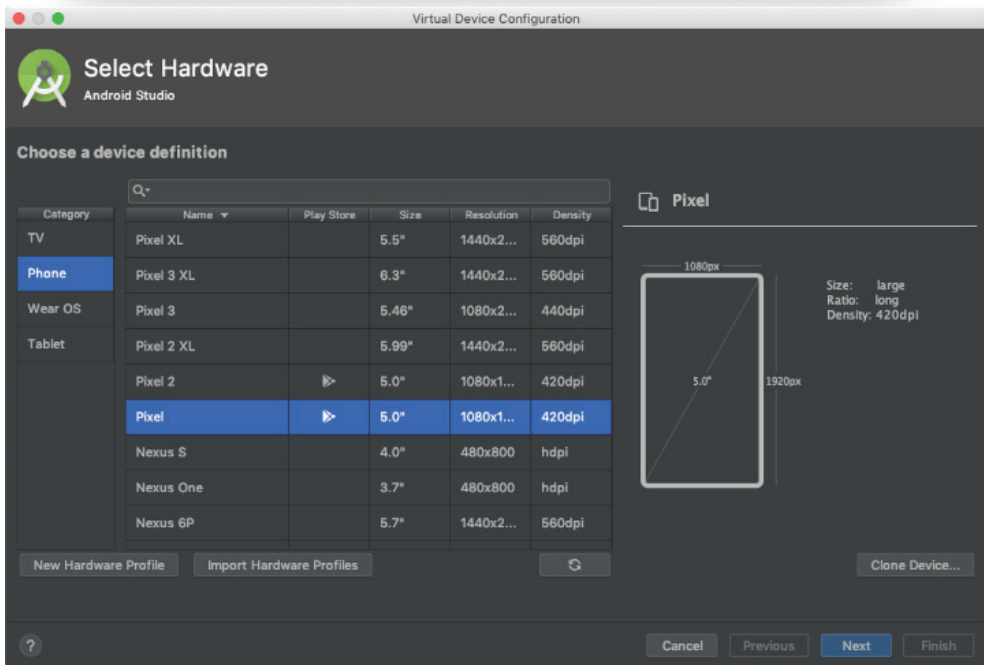
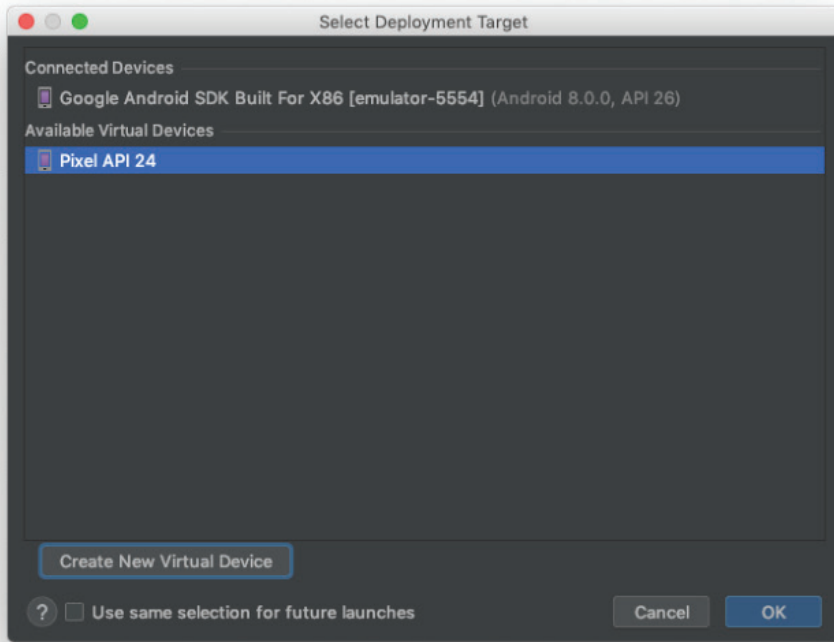


Рис. 16.3 ❖ Создание и запуск эмулятора или подключение устройства

После этого вы должны увидеть на экране сообщение «Hello World!», как показано на рис. 16.4.



Рис. 16.4 ❖ Hello World!

Также на экране должна появиться панель инструментов с надписью «My Application» (мы указали ее на этапе настройки проекта). Но откуда взялась надпись сказать «Hello World!»? Все просто, на самом деле шаблон **Empty Activity** не совсем пустой – если открыть файл *MainActivity.java*, можно увидеть ссылку на файл макета *R.layout.activity_main*. Найдите этот файл в папке *res/layout* или просто выполните щелчок на ссылке в редакторе кода, удерживая нажатой клавишу **Control/Command**. В нем вы увидите *ConstraintLayout* с дочерним элементом *TextView*. Обратите внимание, что атрибуту *android:text* в *TextView* присвоено строковое значение «Hello World!».

Давайте изменим его. Введите, например, строку «iOS is awesome!». «iOS?!» – спросите вы – да, iOS! И оставим в стороне бессмысленные баталии. Обе платформы великолепны. Вы можете любить или не любить каждую из них, но посмотрим правде в глаза – обе делают все возможное, чтобы помочь нам выразить свои идеи.

А теперь снова запустите приложение. Теперь вы должны увидеть другую надпись, как показано на рис. 16.5.

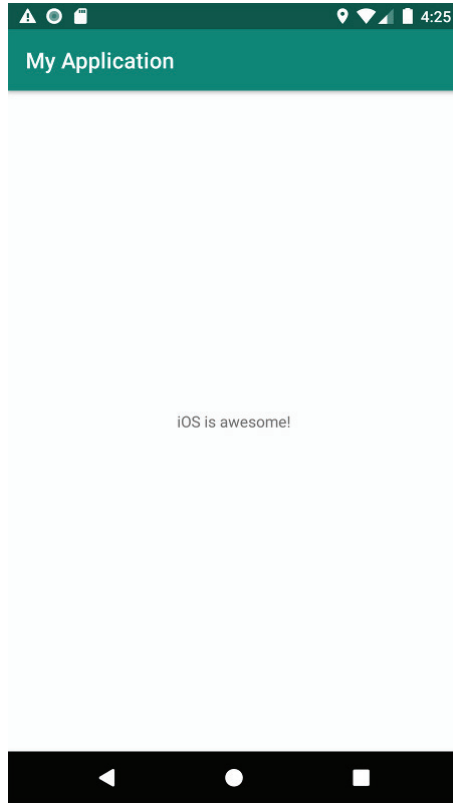


Рис. 16.5 ❖ iOS is awesome!

Итак, вы загрузили и установили Android Studio, создали простое приложение и изменили отображаемую им надпись. Это был очень простой пример, но не волнуйтесь – в следующих нескольких главах мы создадим полноценное приложение, использующее все приемы, описанные в первой части этой книги.

Xcode

В Xcode используется аналогичный процесс настройки проекта для iOS, хотя и включает чуть больше шагов. Для начала запустите Xcode, для этого перейдите в раздел **Applications** (Приложения) и дважды щелкните на значке приложения **Xcode**. На запуске Xcode отображает заставку, изображенную на рис. 16.6.

Щелкните на ссылке **Create a new Xcode project** (Создать новый проект Xcode). После этого откроется новое окно Xcode, и вам на выбор будет предложен список шаблонов, содержащих заготовки проектов с представлениями и шаблонным кодом, которые помогут быстро приступить к работе. В списке присутствуют шаблоны приложений и библиотек, но мы сосредоточимся только на приложениях. В нашем проекте будет несколько экранов, но сейчас мы выберем шаблон **Single View App** приложения с одним представлением. Выберите этот вариант и щелкните на кнопке **Next** (Далее).



Рис. 16.6 ❖ Заставка, отображаемая на этапе запуска Xcode

- ❑ Если после запуска Xcode заставка не появилась на экране, не пугайтесь! Просто выберите в меню пункт **File** ⇒ **New** ⇒ **Project** (Файл ⇒ Создать ⇒ Проект).

Далее будет предложено настроить параметры проекта. Многие из них имеют вполне подходящие значения по умолчанию. Все эти параметры можно будет изменить позже, но некоторые из них желательно установить заранее. Первый параметр, который мы должны заполнить, – **Product Name** (Название продукта). Это часть имени, которое iOS будет использовать для внутреннего представления нашего приложения, в сочетании вместе с идентификатором организации. По умолчанию это название отображается также под значком приложения на экране запуска на устройстве. Введите в этот параметр строку «Library Buddy».

Поле **Organization Identifier** (Идентификатор организации) обычно содержит перевернутое доменное имя компании или организации (или отдельного лица!). Вы можете использовать любой идентификатор по своему выбору, но мы в данном примере используем идентификатор «com.oreilly».

Убедитесь, что выбран язык Swift и сброшены все флажки, имеющие отношение к Core Data и тестам (модульным и пользовательского интерфейса). Щелкните на кнопке **Next** (Далее). Выберите папку для размещения проекта и щелкните на кнопке **Create** (Создать), чтобы создать проект. После этого вновь созданный проект откроется в окне Xcode, и в панели слева появится список файлов, составляющих его.

Щелкните на кнопке **Build and Run** (Собрать и запустить) – выглядит как кнопка воспроизведения – в левом верхнем углу окна проекта. Xcode соберет проект, откроет окно симулятора **iOS Simulator** и запустит приложение, как показано на рис. 16.7.

- ❗ Если симулятор **iOS Simulator** не выбран по умолчанию в раскрывающемся списке рядом с кнопкой **Build and Run** (Собрать и запустить), выберите его. Если симулятор отсутствует в списке, выберите в меню пункт **Window** ⇒ **Devices & Simulators** (Окно ⇒ Устройства и симуляторы), чтобы открыть диалог **Device Organizer** (Организатор устройств). Выберите **Simulators** (Симуляторы) в верхней части окна, а затем щелкните на кнопке **+** в левом нижнем углу экрана, чтобы добавить новый симулятор для разработки.

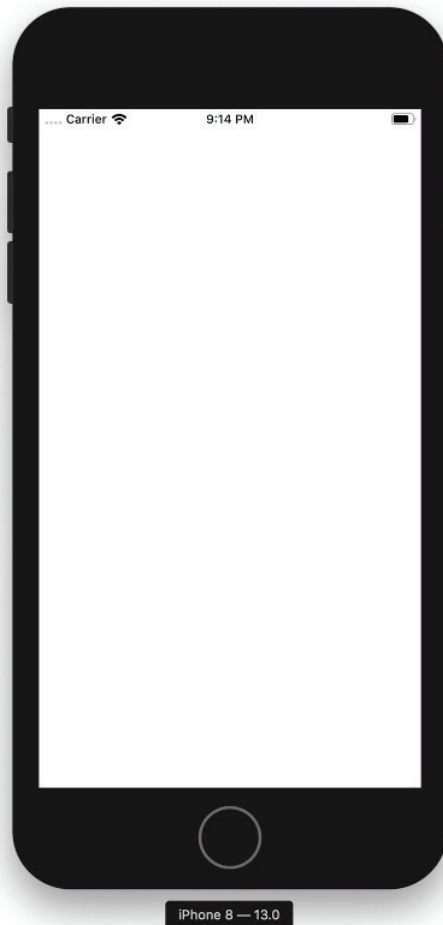


Рис. 16.7 ❖ Наше приложение, запущенное в симуляторе iOS

Чтобы не преуменьшать или не преувеличивать достоинства и недостатки каждой платформы и продемонстрировать отсутствие какой-либо предвзятости с нашей стороны, продолжим и добавим на экран нашего приложения надпись, восхваляющую Android. В списке файлов выберите *Main.storyboard*, щелкните на кнопке **+** в правом верхнем углу окна и перетащите визуальный компонент метки на пустой белый холст. Дважды щелкните на метке и введите текст на «Android is awesome!». Перетащите метку в центр представления,

соберите и запустите приложение, как уже делали это раньше, и вы увидите экран, как показано на рис. 16.8.

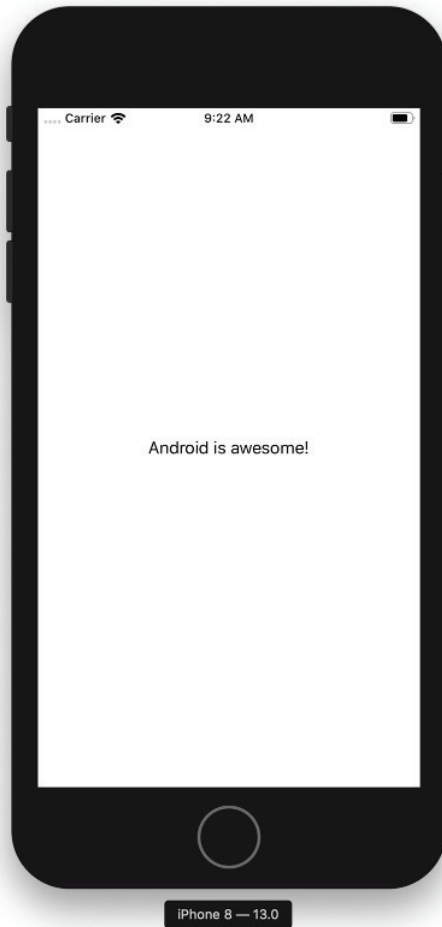


Рис. 16.8 ❖ Android is awesome!

Итак, мы создали проект, который можно собрать и запустить в симуляторе iOS. Прежде чем двинуться дальше, поговорим об архитектуре приложения, которое мы создаем.

АРХИТЕКТУРА ПРИЛОЖЕНИЯ

Приложение, которое мы создадим в следующих нескольких главах, будет иметь несколько экранов, отображающих разные данные. Сразу после запуска приложение будет отображать экран приветствия с тремя кнопками. Щелчок на любой из них перенесет вас в отдельную часть приложения: к списку доступных книг; к списку книг, отмеченных пользователем для прочтения в бу-

душем; и к экрану поиска, где пользователь сможет найти книгу по названию или автору.

Кроме этих экранов, мы также создадим отдельный экран, который будет использоваться каждым из этих трех экранов для отображения информации о конкретной книге.

Одним из преимуществ Android и iOS является отсутствие привязки к выбору конкретной структуры приложения. Вам доступно множество вариантов, но вообще наиболее предпочтительным считается выбор архитектуры MVC или MVVM. Поэтому мы реализуем архитектуру MVC в нашем приложении.

Модель–представление–контроллер

Архитектура модель–представление–контроллер (Model-View-Controller, MVC) считается одним из самых распространенных подходов к разработке приложений. По сути, это архитектурный шаблон, который определяет структуру объектов, составляющих приложение. Слово «model» (модель) в аббревиатуре MVC соответствует структуре данных, используемых приложением. Это могут быть постоянные данные (например, книги, сохраненные для дальнейшего использования) или временные, полученные в ответ на сетевой запрос. Эти данные отделены от их представления на экране контроллерами; контроллеры – это объекты, которые обеспечивают связь между моделью и представлением данных.

Обычно контроллер отвечает за получение данных из базы данных или сетевого ресурса и передает их в представление для отображения на экране. Существуют также специальные контроллеры, которые отвечают за отображение самих представлений. К их числу относятся: в Android – объекты Activity, а в iOS – UIViewControllers.

Основная цель архитектуры MVC состоит в определении границ объектов и предотвращении образования тесных связей между ними. Это упрощает анализ и сопровождение кода.

Теперь, выбрав архитектуру MVC, приступим к созданию первого экрана, который отображается сразу после того, как пользователь запустит приложение: экран приветствия.

СОЗДАНИЕ ПЕРВОГО ЭКРАНА

Как вы видели выше, в предыдущем примере, экран приложения, запущенного внутри симулятора Android и iOS, выглядит очень просто и лишен каких-либо графических элементов и данных. Исправим это.

i Обратите внимание, что в обеих системах, Android и iOS, используется понятие «экран запуска». Это статическое изображение, отображаемое на экране устройства, пока приложение выполняет операции настройки. Отметьте также, что на этом экране бессмысленно отображать какие-либо интерактивные элементы управления пользовательским интерфейсом и в течение этого времени нежелательно или невозможно выполнять сетевые запросы.

Android

В Android экран запуска появляется сразу после запуска приложения, отображается в течение периода инициализации и выводит изображение, сформированное на основе разметки XML. Это означает, что в этом экране отсутствует какая-либо логика и даже недоступны экземпляры класса `Drawable` (хотя в версиях API 26 и выше можно использовать нестандартные рисованные элементы XML, которые могут ссылаться на экземпляры `Drawable`). Также имейте в виду, что на столь раннем этапе выполнения приложения отсутствует доступ ко многим параметрам и механизмам платформы, таким как номер версии API, поэтому попытка использовать разные экраны запуска для разных версий обречена на неудачу. В следующем разделе мы подробно рассмотрим, как создать экран запуска.

Экран запуска

В Android экран запуска, отображаемый в течение инициализации приложения, имеет фон, точно соответствующий выбранной теме оформления. Это может быть любой экземпляр `Drawable`, объединяющий все необходимые операции рисования. В нашем примере мы установим черный фон и выведем в центре логотип, используя для этого XML-элемент `layer-list`, определяющий `Drawable`.

Итак, создайте файл `launch_drawable.xml` и сохраните его в папке `res/drawable`:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
  <item>
    <color android:color="#FF000000" />
  </item>
  <item>
    <bitmap
      android:gravity="center"
      android:src="@drawable/dlml_logo"
      android:tileMode="disabled" />
  </item>
</layer-list>
```

Обратите внимание на ссылку на скомпилированный ресурс с растровым изображением. В графическом редакторе по своему выбору создайте стилизованное изображение с названием приложения, графическим логотипом или просто с вашими инициалами – что вам больше по душе. Имена файлов со скомпилированными ресурсами в Android должны состоять из строчных букв, символов подчеркивания и цифр. Назовите файл с изображением `dlml_logo.xml` и сохраните в каталоге `res/drawable`. Система автоматически превратит это имя в константу в глобальном классе `R`, следуя формату `R.{resource_type}.{file_name_minus_ext}`, то есть в данном случае в константу `R.drawable.dlml_logo`, определяющую целочисленный идентификатор ресурса с изображением логотипа.

Графические ресурсы и разрешение экрана

Каждому устройству с ОС Android присваивается своя категория в зависимости от разрешения экрана: низкое разрешение (ldpi), среднее (mdpi), высокое (hdpi), сверхвысокое (xhdpi), сверхсверхвысокое (xxhdpi) и сверхсверхсверхвысокое (xxxhdpi). Категория автоматически выбирается устройством, и вам не придется определять ее самому.

Однако есть возможность воспользоваться преимуществами автоматического масштабирования изображений, используя специальные каталоги внутри папки *res*:

- */res/drawable/ldpi* – для изображений, размеры которых составляют 75 % от размеров стандартного изображения;
- */res/drawable/mdpi* – для изображений со стандартными размерами, как на веб-страницах или на фото;
- */res/drawable/hdpi* – для изображений, размеры которых составляют 150 % от размеров стандартного изображения;
- */res/drawable/xhdpi* – для изображений, размеры которых составляют 200 % от размеров стандартного изображения;
- */res/drawable/xxhdpi* – для изображений, размеры которых составляют 300 % от размеров стандартного изображения;
- */res/drawable/xxxhdpi* – для изображений, размеры которых составляют 400 % от размеров стандартного изображения.

То есть если изображение должно иметь ширину 100 пикселей и высоту 40 пикселей, вы должны создать версию с размерами, составляющими 75 % от стандартных, то есть шириной 75 пикселей и высотой 30 пикселей, и сохранить ее в каталоге */res/drawable/ldpi*. Исходное изображение сохранить в каталоге */res/drawable/mdpi*. Создать версию с размерами, увеличенными до 150 %, – 150 пикселей в ширину и 60 пикселей в высоту – и сохранить ее в каталоге */res/drawable/hdpi* и т. д. Система проверит разрешения экрана устройства и выберет подходящее изображение для отображения.

Обратите внимание, что обычно система хорошо справляется с уменьшением (но плохо с увеличением) изображений, поэтому некоторые разработчики просто создают изображения с наибольшим разрешением, которое они собираются поддерживать (обычно *xhdpi* или *xxxhdpi*), и сохраняют его в соответствующем каталоге, зная, что система уменьшит при отображении на устройстве с более низким разрешением экрана. Тем не менее не так давно в Play Store появилась новая функция *Android Bundles*, способная выбрать подходящие ресурсы для устройства и собрать файл APK, включающий только файлы, нужные конкретному устройству.

При желании можно создать изображение двойного размера и добавить его в каталог */res/drawable/xhdpi*. См. предыдущее примечание, где описываются детали, касающиеся разрешения экрана.

А теперь перейдем к вопросам определения темы оформления в файле значений (используем для этого файл *styles.xml*, который Android Studio уже должна была создать в каталоге *res/values*). Замените тему проекта по умолчанию своей собственной:

```
<?xml version="1.0" encoding="utf-8"?>
<style name="DlmlTheme" parent="Theme.AppCompat.Light.NoActionBar">
  <item name="android:windowBackground">@drawable/launch_drawable</item>
</style>
```

Очевидно, что мы могли бы и, возможно, должны определить ряд других параметров темы оформления, таких как цвет, панель инструментов, поддержка

макета координатора и т. д., но, чтобы не усложнять пример, используем максимально простую тему.

Чтобы зарегистрировать эту тему в приложении, используем манифест приложения *AndroidManifest.xml*. Мы еще не раз вернемся к этому файлу манифеста, чтобы добавить в него дополнительные настройки приложения, но будем делать это постепенно. А пока просто регистрируем нашу тему:

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

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/DlmlTheme" />

</manifest>
```

Вот и все! Теперь, когда пользователь запустит это приложение, прежде чем получить возможность взаимодействовать с ним, он увидит рисунок, который вы указали в *launch_drawable.xml*. Пока наше приложение простое и легкое, оно будет запускаться настолько быстро, что пользователь может не успеть рассмотреть экран запуска, но с добавлением новых компонентов, ресурсов, внешних библиотек и конфигураций время инициализации приложения увеличится, поэтому экран запуска послужит полезным индикатором, сообщаящим, что приложение не зависло и продолжает выполнять подготовительные операции.

iOS

После добавления экрана запуска в Android аналогичная процедура в iOS покажется вам очень похожей. Для начала откройте редактор раскадровки в Xcode. Выберите главный файл раскадровки нашего приложения, *Main.storyboard*, в навигаторе проекта слева, чтобы открыть его в редакторе раскадровки.



Существует также файл *LaunchScreen.storyboard*. Он определяет дизайн приложения в момент запуска приложения, но до его активации.

Файлы раскадровки предназначены для хранения представлений – так называемых «сцен» – и описания переходов между ними. Они могут и должны содержать несколько сцен, связанных друг с другом заранее определенными переходами, а также простые представления, такие как кнопки, метки и т. д. внутри сцен. Раскадровки могут даже ссылаться на другие раскадровки!

Когда *Main.storyboard* откроется в редакторе раскадровки, вы должны увидеть одну сцену с именем **View Controller Scene** в схеме документа слева. Эта сцена автоматически генерируется, когда приложение создается из шаблона Single View Application, как мы сделали это ранее. Мы могли бы повторно использовать эту сцену и переименовать ее, но давайте пойдем другим путем и создадим новую сцену.

Сначала щелкните на кнопке **Library** (Библиотека) – это крайняя левая кнопка в правом верхнем углу окна проекта. В результате откроется плавающее окно, откуда вы сможете перетаскивать представления и компоненты в редактор раскадровки. Прокрутите содержимое окна вниз и найдите объект **View Controller** в списке, или выполните поиск по словам «view controller».

Затем перетащите мышью объект **View Controller** на холст в редакторе раскадровки, чтобы создать новую сцену. Также можно просто дважды щелкнуть на объекте **View Controller**, и новая сцена автоматически будет добавлена на холст.

Теперь добавим надпись в это представление. Снова вернитесь к окну **Library** (Библиотека) и найдите объект **Label**. Перетащите его в только что созданную сцену. При любом наведении указателя мыши на сцену она будет выделяться синим цветом, как бы подсказывая, в какую сцену будет добавлен перетаскиваемый объект. Когда указатель мыши окажется над нужной сценой, сбросьте метку, и она появится на экране. Вы также можете заметить, что она появилась в схеме документа слева, как показано на рис. 16.9.

Давайте изменим текст метки. В редакторе раскадровки справа есть несколько инспекторов. Содержимое этих инспекторов изменяется в зависимости от выбранного объекта. Если метка все еще активна (если нет – просто щелкните на ней), щелкните на значке инспектора атрибутов (четвертый слева). После этого должен появиться инспектор атрибутов, как показано на рис. 16.9.

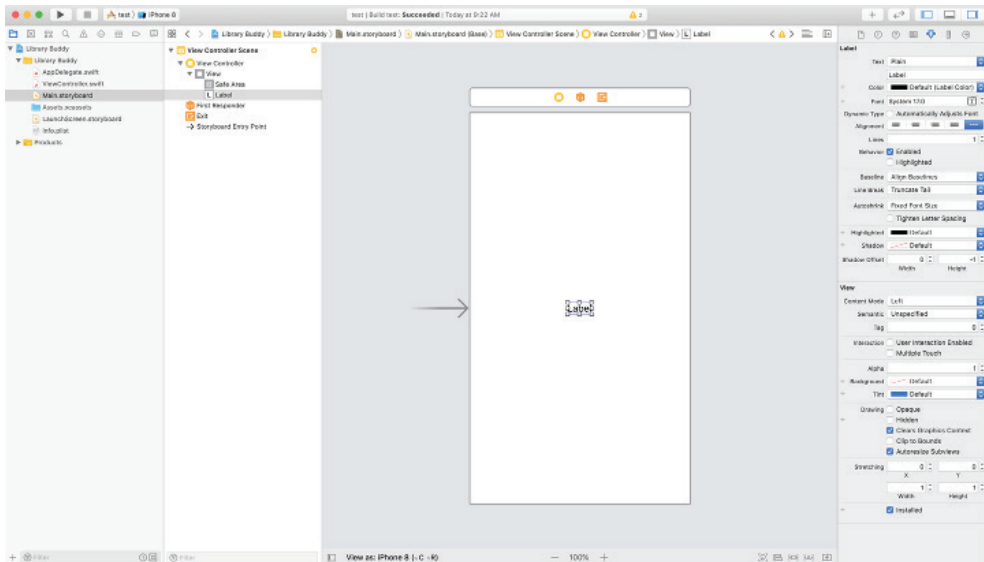


Рис. 16.9 ❖ Инспектор атрибутов в Xcode

Инспектор атрибутов – наряду с инспекторами размеров и соединений – является основным инструментом в Xcode, в котором производится настройка представлений. В подразделе **Label** можно увидеть ряд параметров, соответствующих настройкам объекта **Label**, который вы только что добавили в сцену.

Среди них имеется текстовое поле со значением «Label». Изменив это значение, можно изменить текст, отображаемый меткой в данный момент. Введите в это поле какой-нибудь другой текст, например «Welcome». При желании можете также изменить внешний вид или шрифт метки.

Настроив метку по своему вкусу, попробуйте собрать и запустить приложение, как мы делали это выше в этой главе, щелкнув на кнопке **Build and Run** (Собрать и запустить) в верхней части окна проекта. После запуска вы заметите, что ничего, собственно, не изменилось. А причина этого в следующем: обратите внимание на большую стрелку слева от оригинальной сцены, которая была сгенерирована автоматически, когда мы создали приложение из шаблона. Эта стрелка указывает на начальный контроллер представления, который будет отображаться раскадровкой.

Исправим эту проблему.

Выберите вновь добавленную сцену, щелкнув на ней в схеме документа слева или на белом прямоугольнике над имитацией экрана телефона. Активизируйте инспектор атрибутов после выбора сцены, если необходимо. В разделе **View Controller** (Контроллер представления) найдите **Is Initial View Controller** (Начальный контроллер представления), который должен быть снят. Установите этот флажок, и большая стрелка должна оказаться рядом с новой сценой.

Теперь снова соберите и запустите приложение и посмотрите, что из этого выйдет.

Вы должны увидеть в симуляторе вашу новую сцену. Ура!

Выбор главной раскадровки

Вы можете спросить, как Xcode узнает, что необходимо использовать *Main.storyboard*? Все просто: если в проекте имеется файл *Main.storyboard*, он будет использоваться как раскадровка по умолчанию, если явно не определено иное в настройках проекта. В сложных приложениях можно и рекомендуется создать несколько раскадровок. В такой ситуации может не иметься «главной» раскадровки или файл *Main.storyboard* будет содержать только структурную оболочку приложения. Но не будем забегать вперед. Просто знайте, что это соглашение по конфигурации, которое можно изменить!

Однако рано праздновать.

Мы только что добавили новый экран в приложение, но хотелось бы чего-то большего, чем простое отображение статического экрана. Было бы неплохо получить возможность ссылаться на некоторые представления на экране из кода и изменять их, чтобы сделать картинку немного более динамичной. Выше мы использовали редактор раскадровки, чтобы создать представление. Теперь заглянем внутрь механизма отображения представлений в iOS и познакомимся с контроллером представления.

Добавление контроллера представления

Чтобы создать новый контроллер представления в Xcode, который будет управлять сценой приветствия, добавим новый файл. Для этого щелкните на кнопке + в левом нижнем углу окна Xcode и выберите в контекстном меню **New File...** (Новый файл...), или в главном меню пункт **File** ⇒ **New** ⇒ **File** (Файл ⇒ Создать

⇒ Файл). В любом случае после этого откроется диалог выбора типа файла, который требуется добавить в проект.

Чтобы создать любой файл с исходным кодом на Swift, можно выбрать тип **Swift File**, но в данном конкретном случае предпочтительнее выбрать тип **Cocoa Touch Class**, потому что контроллер представления является частью Cocoa Touch – фреймворка, который управляет iOS. Сделав это, щелкните на кнопке **Next** (Далее).

На следующем экране выберите подкласс `UIViewController` и присвойте ему имя `WelcomeViewController`. Снимите флажок **Also create XIB file** (Также создать файл XIB) и выберите язык Swift, после чего щелкните на кнопке **Next** (Далее). На следующем экране оставьте местоположение файла по умолчанию и щелкните на кнопке **Create** (Создать). После этого в навигаторе проекта, слева в окне Xcode, должен появиться новый файл с именем `WelcomeViewController.swift`.

- ✔ Существует соглашение о выборе имен для контроллеров представлений, согласно которому они должны иметь окончание `ViewController`. Это стандартная практика, и мы рекомендуем выбирать для контроллеров представлений такие имена, как `WelcomeController` или `WelcomeScene`.

Этот файл содержит стандартный код:

```
import UIKit

class WelcomeViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Выполните здесь дополнительные настройки представления.
    }

    /*
    // Примечание от Марка: - Навигация

    // В приложениях, основанных на раскадровках, часто бывает желательно
    // выполнить дополнительные настройки для навигации
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Получить новый контроллер представления с помощью segue.destination
        // Передать выбранный объект в новый контроллер представления
    }
    */
}
```

В данный момент это почти пустой файл, содержащий только объявление объекта с именем `WelcomeViewController`, который наследует класс `UIViewController`. Как уже говорилось выше в этой главе, `UIViewController` – это базовый класс для всех контроллеров представлений; аналог (хотя и не полный) объекта `Activity` в Android.

Теперь, создав наш контроллер представления, подключим его к нашей сцене, созданной ранее.

Вернитесь в редактор раскадровки, выбрав файл `Main.storyboard` в навигаторе проекта слева. Когда редактор откроется, выберите нашу новую сцену конт-

роллера представления, которую мы перед этим настроили как начальную. Откройте инспектор идентичности, щелкнув на третьей слева кнопке в правом верхнем углу окна Xcode. В разделе **Custom Class** (Выбранный класс) найдите поле **Class** (Класс), в котором в настоящий момент отображается имя `UIViewController`, выделенное серым цветом. Это класс, или тип, объекта, представления. Введите в это поле имя `WelcomeViewController`.

После этого изменится название сцены в схеме документа слева. До этого она называлась `View Controller Scene`, а теперь – `Welcome View Controller Scene`. Продолжим и удалим исходную сцену, унаследованную из шаблона проекта, выбрав ее в схеме документа и щелкнув на кнопке **Delete** (Удалить).

Определение выходов представления

Как отмечалось выше, мы хотели бы иметь возможность управлять представлениями из контроллера представления. Именно это мы сейчас и реализуем, используя выход представления.

Выходы дают возможность связать представление с контроллером. Установив такую связь, мы получим возможность передавать ссылки на представления и настраивать их в программном коде. Воспользуемся этим. Вернитесь к файлу `WelcomeViewController.swift` и для начала удалите закомментированный стандартный код, как показано ниже:

```
import UIKit

class WelcomeViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        // Выполните здесь дополнительные настройки представления.
    }
}
```

Сразу за объявлением `class` добавьте строку:

```
@IBOutlet weak var headerLabel: UILabel!
```

Она добавит в класс свойство `headerLabel` с типом `UILabel`, который соответствует объекту метки в сцене. Теперь изменим цвет метки на время, пока сцена загружается, добавив следующую строку в конец существующего метода `DidLoad()`:

```
headerLabel.textColor = .red
```

Эта инструкция изменит значение свойства `textColor` и заставит метку окраситься в красный цвет, после того как представление загрузится. Весь файл `WelcomeViewController.swift` теперь должен выглядеть так:

```
import UIKit

class WelcomeViewController: UIViewController {
    @IBOutlet weak var headerLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

```
        headerLabel.textColor = .red
    }
}
```

Мы добавили выход в класс контроллера представления, но если теперь собрать и запустить приложение, ничего не изменится. Мы почти достигли цели! Нам осталось только подключить метку в раскадровке.

Связываем все вместе

Вернитесь в редактор раскадровки. В плавающем окне над сценой наведите указатель мыши на значок **Welcome View Controller**. Нажмите клавишу **Ctrl** и левую кнопку мыши и, удерживая их нажатыми, перетащите указатель мыши на метку с текстом приветствия. В результате должна появиться синяя линия, соединяющая эти два элемента. Отпустите кнопку мыши, и на экране появится плавающее окно с надписью **Outlets** и свойством `headerLabel` в нем, которое мы создали в `WelcomeViewController`. Щелкните на `headerLabel`, и окно должно исчезнуть.

Откройте инспектор соединений справа и убедитесь, что свойство `headerLabel` теперь подключено к `Header Label` в сцене.

Соберите и запустите приложение, щелкнув на кнопке **Build and Run** (Собрать и запустить) в левом верхнем углу окна проекта, и в окне эмулятора должна появиться метка с красным текстом `Welcome`.

Что мы узнали

Теперь кратко вспомним, что вы узнали в этой главе.

Сначала мы показали, как создать новый проект в `Android Studio` и `Xcode`. Затем немного поговорили об архитектуре модель–представление–контроллер (`Model-View-Controller`, `MVC`) и в общих чертах обрисовали приложение, которое будем создавать. Наконец, мы исследовали процедуру добавления нового экрана в приложение – в данном случае простого экрана приветствия. Также вы узнали, как создавать визуальные компоненты представлений, как связывать их и обращаться к ним из кода.

Уф! Это была длинная глава, и при этом мы только-только начали создавать что-то полезное. Давайте продолжим и в следующей главе посмотрим, как отображать списки данных, а также немного скорректируем оформление нашего приложения!

Глава 17

Вывод списка с данными

В предыдущей главе мы рассмотрели основы создания новых проектов в Android Studio и Xcode. Как вы помните, таинственный библиотекарь поручил нам создать приложение для его библиотеки. Это обычное дело для разработчика мобильных приложений.

В этой главе мы немного расширим наше приложение и попутно узнаем:

- как настраивать представления и управлять ими;
- как инициировать действия по нажатию кнопок;
- как выводить списки с данными;
- как осуществлять переходы между экранами.

Это будет длинная глава, поэтому не будем задерживаться – будущие посетители библиотеки ждут.

ОФОРМЛЕНИЕ ПРЕДСТАВЛЕНИЙ

В текущем виде наши приложения для Android и iOS выглядят довольно уныло и непривлекательно. Нужно создать что-то более впечатляющее, чем простая надпись «Welcome» на пустом фоне. Обе платформы имеют довольно широкий набор средств для оформления внешнего вида, с помощью которых мы можем сделать наше приложение более презентабельным и привлекательным. Мы не будем углубляться в особенности этих инструментов – это слишком глубокая тема, и их возможности практически безграничны, – а просто добавим немного оформления в приложение.

Android

Приступив к созданию приложения в главе 15, мы выбрали простейший шаблон проекта и добавили в него метку с текстом. Теперь продолжим работу над этим проектом и немного обновим экран приветствия. В этой упрощенной версии приложения мы не будем использовать аутентификацию, поэтому ограничимся выводом на экране приветствия логотипа, небольшого введения, представляющего библиотеку, а также флажка и метки с текстом **Accept Terms** (Принимаю условия использования). Добавим также кнопку для обзора содер-

жимого библиотеки, которая будет оставаться неактивной, пока пользователь не установит флажок **Accept Terms** (Принимаю условия использования). Нажатие этой кнопки будет запускать один из наших основных контроллеров пользовательского интерфейса с говорящим именем `BrowseLibraryActivity`.

Также напомним, что нам заранее неизвестны размеры экрана на устройстве пользователя, которое могло быть выпущено и в 2010 году, и в 2024-м. Также мы не можем быть уверены в выборе ориентации (альбомной или книжной), и если следовать рекомендациям по доступности, даже размер шрифта может сильно отличаться, в зависимости от настроек устройства. Поэтому нам придется прокручивать отображаемую информацию, чтобы пользователь смог прочитать ее и принять (или не принять) установленные правила или, щелкнув на кнопке, посмотреть содержимое библиотеки.

Для начала определим некоторые факты. Ресурсы в Android, такие как строки и изображения, обычно хранятся в структурах XML и идентифицируются целыми числами, сгенерированными системой, как мы рассказывали, когда описывали создание логотипа.

Добавление строковых и графических ресурсов

Итак, нам понадобятся: вводный текст, представляющий приложение библиотеки, текст с условиями использования приложения, которые пользователь должен принять, и текст для надписей на кнопке и рядом с флажком. Как вы помните, мы уже создали файл логотипа и зарегистрировали его в системе, поместив в соответствующий каталог ресурсов. Для строковых значений нам понадобится создать ресурсы другого типа: `values`. Традиционно строковые ресурсы хранятся в `/res/values/strings.xml`, в узлах `string`, имеющих атрибут `name`, по которому они идентифицируются во время компиляции; значением строки является текстовое содержимое узла.

Добавим несколько таких узлов в наш файл `strings.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">DunnAndLewisMemorial</string>
  <string name="introduction">Welcome to the app for the Dunn &
    Lewis Memorial Library! This library includes books of all
    stripes, from code books on Java 8 to code books on Java 9!
  </string>
  <string name="terms">All books in the Dunn & Lewis Memorial
    Library are copyrighted by their respective authors, and all
    state and federal copyright laws apply in full effect
    and will be enforced by the library.</string>
  <string name="terms_accept_label">Do you accept these terms?</string>
  <string name="browse_button_label">Browser</string>
</resources>
```

Теперь у нас есть не только растровое изображение логотипа, но и весь текст для отображения на экране приветствия.

Создадим новый файл макета для этого экрана:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
```

```

xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:background="@android:color/white">
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <ImageView
        android:id="@+id/logo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:adjustViewBounds="true"
        android:src="@drawable/dlml_logo" />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/introduction" />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/terms" />
    <CheckBox
        android:id="@+id/terms_checkbox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/terms_accept_label" />
    <Button
        android:id="@+id/browse_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:enabled="false"
        android:text="@string/browse_button_label" />
</LinearLayout>
</ScrollView>

```

Что мы здесь видим?

Разберем содержимое этого файла элемент за элементом:

- 1) корневой узел макета – `ScrollView` – определяет представление, которое при необходимости прокручивает свое содержимое. Мы определили не так много информации, и на большинстве экранов прокрутка может не понадобиться, но она очень пригодится при отображении на устройстве с маленьким экраном, имеющим низкое разрешение, если пользователь выбрал в настройках большой размер шрифта, или даже на стандартном устройстве в альбомной ориентации. Крайне важно, чтобы все интерактивные элементы пользовательского интерфейса, даже те, которые просто выводят информацию, были доступны для пользователя, и пред-

ставление с прокруткой является одним из традиционных способов решения этой задачи;

- 2) мы знаем, что все остальные компоненты будут располагаться вертикально – друг за другом. Этот традиционный подход к размещению используется практически во всех современных движках пользовательского интерфейса, от традиционных механизмов отображения разметки HTML веб-страниц до средств просмотра файлов PDF и документов, созданных такими редакторами, как Pages или MS Word, markdown, AsciiDoc (tor), troff, (La)TeX и т. д. Представление `LinearLayout`, предлагаемое фреймворком, предназначено как раз для такого упорядоченного вывода элементов. Важно отметить, что по умолчанию `LinearLayout` использует горизонтальную ориентацию, когда каждый следующий элемент размещается правее предыдущего; поэтому нам понадобилось определить атрибут `orientation` со значением `vertical`, чтобы добиться желаемого эффекта;
- 3) далее следует логотип. Нам нужно, чтобы изображение заполняло страницу на всю ширину, но сохраняло соотношение сторон по вертикали; поэтому мы определили следующие атрибуты:

```
android:layout_width="match_parent"
android:layout_height="wrap_content"
```

- 4) простой блок текста с описанием приложения и библиотеки;
- 5) еще один блок текста с условиями использования и ссылкой на авторские права;
- 6) экземпляр `CheckBox` с меткой. Первоначально флажок не отмечен, поэтому кнопка, следующая ниже, объявлена неактивной;
- 7) простая кнопка – экземпляр `Button`, – которую можно нажать, когда она будет активирована. Эта кнопка откроет экран, с помощью которого пользователь сможет начать знакомиться с библиотекой.

Вот и все! Поскольку наш `MainActivity` уже вызывает `setContentView` и передает скомпилированный идентификатор этого файла макета, на начальном экране приветствия теперь будут отображаться описанные элементы пользовательского интерфейса, а не просто «Hello World!», как это было раньше.

Предыдущего примера макета явно недостаточно, чтобы просто продемонстрировать все, что *действительно необходимо* для отображения пользовательского интерфейса, поэтому добавим еще несколько строк, определяющих отступы вокруг элементов и их привязку к определенным точкам экрана, чтобы изображение на экране получилось чуть более привлекательным:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">
```

```
<ImageView
    android:id="@+id/logo"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center|top"
    android:layout_marginBottom="16dp"
    android:adjustViewBounds="true"
    android:src="@drawable/dlml_logo" />

<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16dp"
    android:layout_marginEnd="16dp"
    android:layout_marginStart="16dp"
    android:fontFamily="serif"
    android:lineSpacingMultiplier="1.3"
    android:text="@string/introduction"
    android:textColor="@color/colorPrimary"
    android:textSize="14sp" />

<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16dp"
    android:layout_marginEnd="16dp"
    android:layout_marginStart="16dp"
    android:fontFamily="serif"
    android:lineSpacingMultiplier="1.3"
    android:text="@string/terms"
    android:textColor="@color/colorPrimary"
    android:textSize="14sp" />

<CheckBox
    android:id="@+id/terms_checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_marginBottom="16dp"
    android:layout_marginEnd="16dp"
    android:layout_marginStart="16dp"
    android:fontFamily="serif"
    android:lineSpacingMultiplier="1.3"
    android:text="@string/terms_accept_label"
    android:textColor="@color/colorPrimary"
    android:textSize="14sp" />

<Button
    android:id="@+id/browse_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:enabled="false"
    android:text="@string/browse_button_label" />
```

```
</LinearLayout>
```

```
</ScrollView>
```

Помимо отступов, настроек шрифтов и привязки к определенным областям экрана мы также добавили атрибут `adjustViewBounds` в элемент `ImageView` с логотипом. На данный момент мы не выполняем никаких существенных манипуляций с представлением, однако, как мне кажется, нам стоит остановиться и подробнее рассмотреть этот атрибут, потому что он часто используется на практике, но из-за нестандартного поведения может вызывать непонимание даже у опытных разработчиков.

Вот как он определяется в документации для разработчиков (<https://oreil.ly/qKі6P>):

Установите значение `true` в этом атрибуте, если требуется, чтобы `ImageView` корректировал свои границы с сохранением соотношения сторон своего изображения.

Проще говоря, если не присвоить этому атрибуту значение `true`, `ImageView` может рассматривать пробельные части своей области рисования как поля и отступы, которые не были указаны. Если вы сомневаетесь, добавьте этот атрибут со значением `true` во все `ImageView`, для которых не предусматривается особого поведения. Это может показаться избыточным, но если вам нужно, чтобы границы изображений идеально совпадали с границами экрана или других элементов, этот атрибут поможет вам добиться желаемого. А теперь продолжим!

Запустите приложение. Теперь оно должно выглядеть, как показано на рис. 17.1.

Сейчас, когда наш пользовательский интерфейс имеет более или менее привлекательный внешний вид, добавим поведение в экземпляры `CheckBox` и `Button`.

Нам нужно, чтобы после нажатия кнопки в нижней части экрана открывался пользовательский интерфейс, позволяющий просматривать книги; кнопка должна оставаться неактивной, пока пользователь не установит флажок, сообщающий о согласии с нашими условиями (в них мы просто напоминаем, что авторские права на книги распространяются и на цифровые копии). Давайте посмотрим, как реализовать реакцию на флажок в коде.

Откройте файл `MainActivity.java`, в котором находится код, управляющий начальным экраном (экраном с приветствием). Обратите внимание на строку `setContentView(R.layout.main)`, которая анализирует описание представления в файле XML из предыдущей главы и рисует его на экране. На данный момент содержимое файла выглядит примерно так:

Java

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Kotlin

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

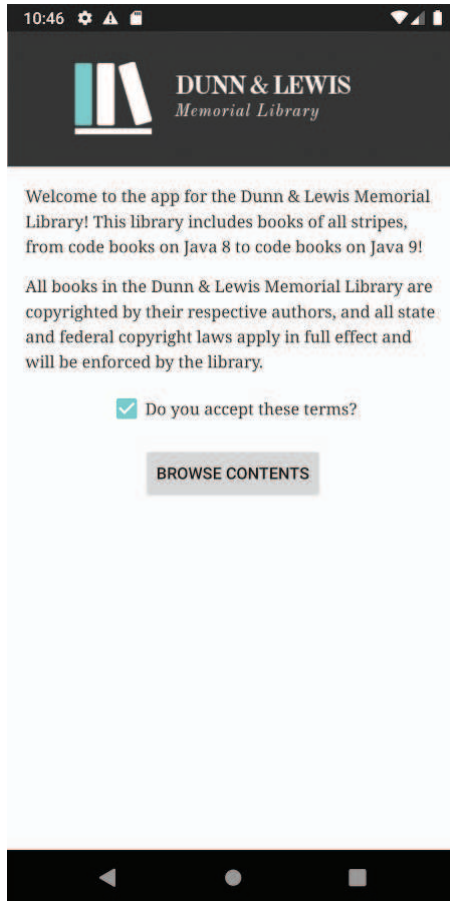


Рис. 17.1 ❖ Начальный экран

Код выглядит довольно простым, правда? Но нам нужно добавить еще кое-что, чтобы получить требуемое нам поведение.

Итак, мы знаем, что нам понадобится выполнить некоторые операции с флажком и с кнопкой, поэтому получим ссылки на них с помощью метода `findViewById`, которым обладают экземпляры `Activity` и `View`. Также добавим переменные-члены для хранения этих ссылок. Вот как теперь должен выглядеть код в `Activity`:

Java

```
public class MainActivity extends Activity {
    private CheckBox mTermsCheckbox;
    private Button mBrowseButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mTermsCheckbox = findViewById(R.id.terms_checkbox);
        mBrowseButton = findViewById(R.id.browse_button);
    }
}
```

А теперь самое интересное: предыдущая версия кода на Kotlin уже реализует все, что нам необходимо! Экземпляр `TextView` в ней доступен как `term_checkbox`, а экземпляр `Button` – как `browse_button`, и для этого не пришлось добавлять ни строчки кода. Экземпляры `Activity` в Kotlin, а также любых классов, реализующих `LayoutContainer`, будут автоматически сохранять ссылки на любые представления, описанные в макете, в переменные-члены с именами, соответствующими идентификаторам этих представлений. Убедитесь сами:

Kotlin

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

Отлично! Теперь у нас есть пользовательский интерфейс и ссылки в памяти на элементы пользовательского интерфейса, которые нам понадобятся. Вспомним, чего мы хотим добиться:

- 1) кнопка `browse_button` должна запускать новый экземпляр `Activity`, чтобы дать пользователю возможность исследовать содержимое библиотеки;
- 2) кнопка `browse_button` должна оставаться неактивной, пока пользователь не установит флажок `CheckBox` с надписью `Assert Terms`.

Чтобы реализовать первый пункт, нужно определить обработчик `View.OnClickListener` для кнопки `Button`. Сделать это можно несколькими способами, как описывалось в главе 4, посвященной пользовательскому вводу, но мы используем ссылки на методы, чтобы сделать код более простым и понятным. Наша реализация `View.OnClickListener` должна возвращать `void` и принимать единственный параметр типа `View`.

Взгляните:

Java

```
public class MainActivity extends Activity {
    private CheckBox mTermsCheckbox;
```

```

private Button mBrowseButton;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mTermsCheckbox = findViewById(R.id.terms_checkbox);
    mBrowseButton = findViewById(R.id.browse_button);
    mBrowseButton.setOnClickListener(this::browseContent);
}

private void browseContent(View view) {
    Intent intent = new Intent(this, BrowseContentActivity.class);
    startActivity(intent);
}
}

```

Kotlin

```

class MainActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        browse_button.setOnClickListener(::browseContent)
    }

    private fun browseContent(view: View) {
        val intent = Intent(this, BrowseContentActivity::class.java)
        startActivity(intent)
    }
}

```

Здесь мы назначили метод `browseContent` обработчиком события щелчка на кнопке. Если теперь попробовать собрать приложение, редактор связей `lint` сообщит об отсутствии `BrowseContentActivity`. Давайте исправим эту ошибку. Для этого создайте минимальный файл с определением подкласса `Activity`, например:

Java

```

public class BrowseContentActivity extends Activity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}

```

Kotlin

```

class BrowseContentActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }
}

```


И не забудьте добавить его в манифест приложения!

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

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/DlmlTheme">

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

        <activity android:name=".BrowseContentActivity" />

    </application>
</manifest>
```

Далее нам нужно гарантировать неактивность кнопки, пока пользователь не установит флажок *Accept Terms*. Для начала добавим атрибут `enabled="false"` в макете XML, а затем будем изменять его в коде при изменении состояния `CheckBox`. Для этого нам понадобится обработчик `OnCheckedChangeListener` – определим его, снова используя ссылку на метод. Согласно контракту `OnCheckedChangeListener`, метод-обработчик должен возвращать `void` и принимать два параметра: экземпляр `Button`, который может быть виджетом `CheckBox`, `Switch` или аналогичным элементом пользовательского интерфейса, а также логическое значение, определяющее состояние флажка. Метод `onCheckedChangeListener`, представленный ниже, соответствует этим критериям; свяжем его с флажком `CheckBox`, ссылку на который мы получили выше:

Java

```
public class MainActivity extends Activity {

    private CheckBox mTermsCheckbox;
    private Button mBrowseButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mTermsCheckbox = findViewById(R.id.terms_checkbox);
        mTermsCheckbox.setOnCheckedChangeListener(this::onCheckChanged);
        mBrowseButton = findViewById(R.id.browse_button);
        mBrowseButton.setOnClickListener(this::browseContent);
    }

    private void onCheckChanged(Button button, boolean checked) {
```

```

        mBrowseButton.setEnabled(checkered);
    }

    private void browseContent(View view) {
        Intent intent = new Intent(this, BrowseContentActivity.class);
        startActivity(intent);
    }
}

```

Kotlin

```

class MainActivity : Activity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(com.oreilly.R.layout.activity_main)
        browse_button.setOnClickListener(::browseContent)
        terms_checkbox.setOnCheckedChangeListener(::onCheckChanged)
    }

    private fun browseContent(view: View) {
        val intent = Intent(this, BrowseContentActivity::class.java)
        startActivity(intent)
    }

    private fun onCheckChanged(button: Button, checked: Boolean) {
        browse_button.isEnabled = checked
    }
}

```

Если теперь снова собрать и запустить приложение, можно заметить, что кнопка приобрела неактивный вид и не реагирует на касания. Но если коснуться флажка (или прикрепленной к нему метки), чтобы установить его, кнопка изменит свой внешний вид и будет выглядеть активной, готовой среагировать на касание. Если теперь коснуться кнопки, откроется новый экран `BrowseContentActivity`. И пусть этот экран пока пустой, но уже само его появление наглядно показывает, что основу навигации в Android составляют взаимодействия с пользователем и экземпляры `Activity`.

Теперь оставим в покое Android и посмотрим, как то же самое сделать в iOS.

iOS

Чтобы упростить себе работу с приложением в будущем, первым делом нам следует настроить его структуру и оформление. Начнем со структуры.

Структура

Первое, что необходимо, чтобы сделать приложение немного удобнее, – определить его структуру. Запустите Xcode и откройте файл `Main.storyboard`. В редакторе раскадровки щелкните на кнопке **Library** (Библиотека) вверху справа в окне проекта (мы уже использовали ее в главе 16, когда добавляли метку в начальный экран); в результате откроется плавающее окно со списком доступных объектов `UIKit`, которые можно перетащить в раскадровку.

Чтобы получить желаемую структуру и внешний вид, характерный для системы, заключим представление с приветствием в контроллер навигации. Найдите объект **Navigation Controller** в окне **Library** (Библиотека), перетащите его на холст в редакторе раскадровки и оставьте рядом с имеющимся представлением с приветствием, как показано на рис. 17.2.

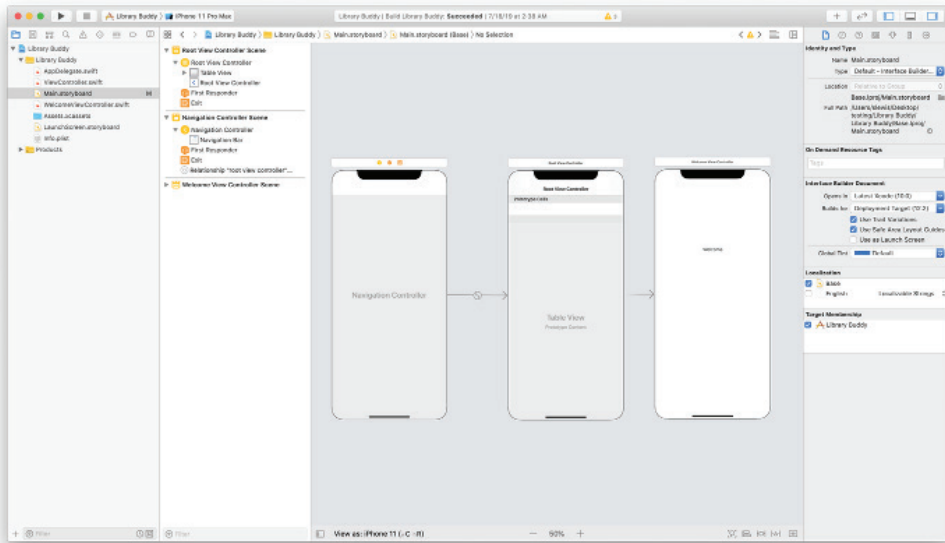


Рис. 17.2 ❖ Добавление контроллера навигации в раскадровку

Контроллер навигации – это объект, входящий в состав фреймворка UIKit, обеспечивающего работу пользовательского интерфейса в iOS. Он используется для упрощения навигации по приложению и обработки состояний переходов между представлениями, что очень пригодится нам потом. А пока самое важное для нас – это панель навигации в верхней части экрана, которая создается контроллером, чтобы помочь пользователю понять, как осуществляется навигация в приложении.

Прямо сейчас контроллер навигации не будет отображаться на экране. Исправим это.

Щелкните на компоненте **Root View Controller**, добавленном автоматически. Это первый контроллер представления, отображаемый только что добавленным контроллером навигации. Удалите его, щелкнув на кнопке **Delete** (Удалить). Каждый навигационный контроллер является экземпляром UINavigationController. Этот класс имеет стек дочерних контроллеров представлений, созданных приложением к данному моменту. Корневым контроллером представления может быть любой объект UIViewController, и фактически он является дном стека. К счастью, у нас есть прекрасный кандидат на роль корневого контроллера представления: наш экран приветствия!

Нам нужно связать эти два контроллера, подобно тому, как мы связывали метку заголовка в главе 2. Наведите указатель мыши на контроллер навигации

и, удерживая нажатой клавишу **Ctrl** и левую кнопку мыши, перетащите указатель на контроллер приветствия, чтобы сформировать связь. В появившемся модальном диалоге выберите **root view controller** (корневой контроллер представления) в разделе **Relationship Segue** (Отношение перехода).

К сожалению, если теперь запустить приложение в симуляторе, можно заметить, что почти ничего не изменилось. Причина в том, что наш контроллер представления с приветствием все еще установлен в раскадровке как начальный контроллер представления. Это означает, что iOS просто создаст экземпляр `WelcomeViewController` и будет использовать его как корневое представление окна.

Чтобы исправить эту проблему, щелкните на сцене контроллера навигации в редакторе раскадровки, откройте инспектор атрибутов и установите флажок **Is Initial View Controller** (Начальный контроллер представления). В результате большая стрелка должна переместиться к контроллеру навигации. Соберите и запустите приложение. Сейчас контроллер представления с приветствием должен отображаться в созданной нами сцене контроллера навигации.

Мы привели структуру в порядок, теперь добавим немного оформления.

Оформление

Улучшить оформление экрана приветствия можно несколькими способами. Самый простой и, пожалуй, лучший – избавиться от метки с заголовком, созданной ранее, и вывести название текущего экрана в контроллере навигации.

Сначала щелкните на метке заголовка в сцене с приветствием и удалите ее, щелкнув на кнопке **Delete** (Удалить). В результате она исчезнет с экрана и из структуры документа, слева от холста.

Затем установим заголовок экрана в контроллере навигации. Это можно сделать программно, но так как мы уже в редакторе раскадровки, сделаем это здесь. В структуре документа слева щелкните на объекте **Navigation Item** в сцене **Welcome View Controller**. Откройте инспектор атрибутов справа и введите текст `Welcome` в поле **Title**.

Также, если хотите, смените значение **Automatic** в поле **Large Title** на **Always**. Это увеличит размер отображаемого заголовка и сделает его более читабельным.

i Свойства объекта **Navigation Item** можно установить программно. Например, описанным выше атрибутам **Title** и **Large Title** соответствуют свойства `navigationItem.title` и `navigationItem.largeTitleDisplayMode` в классе `UIViewController`.

А теперь попробуйте запустить приложение в симуляторе.

Ой-ей-ей! Похоже, что мы все сломали! Как такое получилось?

Ошибки, ошибки и еще раз ошибки!

Добро пожаловать в чудесный мир ошибок времени выполнения! Как вы могли заметить, наше приложение скомпилировалось и запустилось, но потом зависло. Это обусловлено динамической природой UIKit, которая корнями уходит в историю Objective-C. Если открыть класс `WelcomeViewController` в редакторе, можно заметить, что в нем до сих пор имеется свойство `headerLabel`.

Вы можете сказать: «Но я думал, что мы удалили метку заголовка несколькими абзацами выше!» – и будете правы. Мы действительно удалили метку. Но *не удалили* ссылку на нее из класса. Есть несколько способов предотвратить подобные ошибки в будущем. Один из них – изменить тип свойства `UILabel!`, который является неявно распаковываемым необязательным типом (то есть типом, не поддерживающим значение `nil`), типом `Optional`, который поддерживает значение `nil`.

Проще говоря, iOS ожидала, что при любом взаимодействии с этим свойством в методе `viewDidLoad()` оно будет хранить действительную ссылку на объект. Поскольку мы удалили метку из сцены в редакторе раскадровки, ссылка на нее больше не сохраняется в свойстве класса. Компилятор не знает об этом и благополучно скомпилировал проект, потому что все связи устанавливаются во время выполнения, а не во время компиляции. Теперь взгляните на следующий код:

```
headerLabel.textColor = .red
```

Всякий раз, когда приложение выполняет его, система не знает, что ей делать, потому что в `headerLabel` *должна* храниться ссылка на метку, но ее нет.

На самом деле можно точно определить причину ошибки, если прочитаете сообщение о ней, появившееся в консоли, справа внизу в окне Xcode:

```
Fatal error: Unexpectedly found nil while  
    implicitly unwrapping an Optional value
```

```
(  
Фатальная ошибка: встречено значение nil при попытке  
    неявно распаковать необязательное значение  
)
```

Свойство `headerLabel` содержит значение `nil`, которого там не должно быть.

Что такое «необязательные значения»?

Типы, поддерживающие и не поддерживающие значение `null`, являются важной частью языка Kotlin, и им прямо соответствуют необязательные типы в Swift. Необязательные типы, такие как `Object?`, лучше всего рассматривать как некоторый контейнер, или коробку. Коробка может быть пустой или что-то хранить. Используя такой тип, вы говорите компилятору: «Все в порядке, если в коробке что-то есть, то это что-то имеет тип `Object`, но вообще коробка может быть и пустой». В Kotlin и Swift типы по умолчанию не поддерживают значение `null`, то есть переменным этих типов нельзя присвоить `null` или `nil` – эквивалент `null` в языке Swift.

Неявно распаковываемые необязательные типы, такие как `Object!`, – это специальные типы с поддержкой `null`, или тип `Optional` в Swift. Переменным этих типов можно присвоить значение `null`, но, объявляя неявно распаковываемые типы, вы фактически говорите компилятору: «В этой коробке будет храниться объект. В какой-то момент вместо объекта может храниться пустая ссылка, но я гарантирую, что когда потребуется вскрыть коробку, там точно будет нужный объект. Верь мне. Я разработчик и знаю, как работает мое приложение». Но если вы ошибетесь, приложение аварийно завершит работу, поэтому используйте эти типы с большой осторожностью, и только когда вы уверены, что `nil` обязательно заменит действительная ссылка.

За дополнительной информацией о необязательных типах в Swift обращайтесь к разделу «Optionals» в документации Apple (<https://oreil.ly/AMdxC>). За информацией о типах с поддержкой и без поддержки null в языке Kotlin обращайтесь к разделу «Null-безопасность» в руководстве по языку Kotlin (<https://kotlinlang.ru/docs/reference/null-safety.html>).

Если объявить свойство `headerLabel` с типом `UILabel?`, приложение будет компилироваться и запускаться без ошибок. Но это не самое правильное решение, потому что свойство все еще будет присутствовать в классе, и мы по-прежнему будем пытаться присвоить красный цвет свойству `textColor` метки `headerLabel`, но поскольку в `headerLabel` будет храниться значение `nil`, это приведет к тому, что ошибка будет просто игнорироваться. Позже, когда мы вернемся к редактированию класса, мы можем не вспомнить, что удалили метку заголовка из представления, и потратить немало времени, пытаясь выяснить, что делает эта строка. Как мне кажется, лучше оставить `@IBOutlet` как неявно распаковываемый необязательный тип и позволить приложению завершиться с ошибкой после запуска, чтобы она напоминала о необходимости удалить этот код после удаления связанного представления.

А сейчас удалим это свойство из класса `WelcomeViewController`. Попутно добавим несколько строк кода, чтобы отобразить увеличенный заголовок, который мы установили в редакторе раскадровки, в навигационной панели контроллера представления (по умолчанию он выключен). Наш класс должен теперь выглядеть так:

```
class WelcomeViewController: UIViewController {
    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        navigationController?.navigationBar.prefersLargeTitles = true
    }
    override func viewWillDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        navigationController?.navigationBar.prefersLargeTitles = false
    }
}
```

Щелкните на кнопке **Stop** (Стоп) рядом с кнопкой **Build and Run** (Собрать и запустить) в левом верхнем углу окна проекта, чтобы остановить запущенное приложение. Затем щелкните на кнопке **Play** (Запустить), чтобы собрать и запустить проект. Теперь экран приложения должен выглядеть, как показано на рис. 17.3.

Сейчас, когда мы исправили ошибку в приложении, можно пойти дальше и настроить цвет фона и другие свойства экрана приветствия, но мы оставим это вам как самостоятельное упражнение.

Мы потратили довольно много времени, улучшая экран приветствия нашего приложения, и заложили основу для будущей интерактивности. Однако, как вы наверняка помните, наша цель – создать к концу этой главы приложение, отображающее некоторые данные, поэтому продолжим и добавим кнопку.

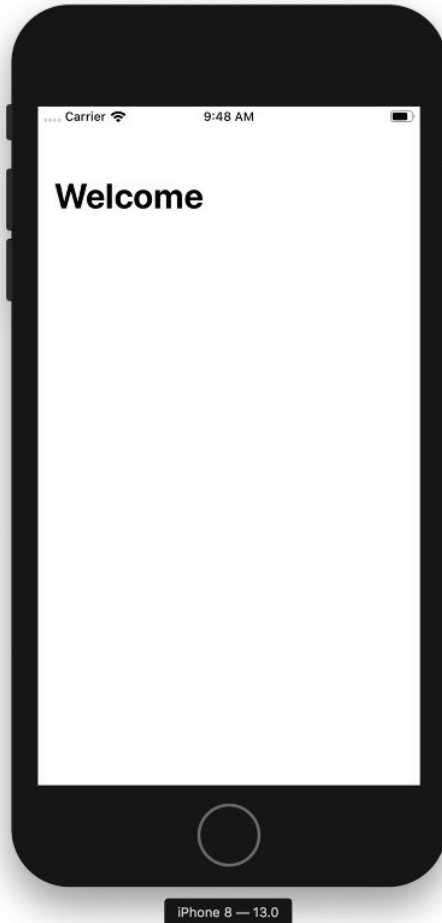


Рис. 17.3 ❖ Экран приветствия после добавления оформления

ДОБАВЛЕНИЕ КНОПКИ

К сожалению, мы как разработчики этого удивительного приложения не можем просто взять и нажать кнопку, чтобы все получилось само собой, как по волшебству. Мы хотим сказать, что должны сами добавить кнопку в приложение, которую пользователи смогут нажимать для получения списка книг. Эта конкретная кнопка будет подписана как **Catalog** (Каталог) и отображать весь каталог библиотеки в виде списка на экране.

Если хотите, можете вернуться немного назад и прочитать, как мы действовали в Android; а теперь сделаем то же самое в iOS.

iOS

Чтобы добавить кнопку в представление, откройте *Main.storyboard* в редакторе раскадровки и щелкните на кнопке **Library** (Библиотека) для открытия библиотеки компонентов пользовательского интерфейса. Затем найдите объект **Button** и перетащите его на экран приветствия в редакторе. В результате на холсте должна появиться кнопка с надписью **Button**.

Сейчас кнопка находится там, куда вы ее сбросили, но в идеале хотелось бы, чтобы она отображалась в центре экрана. Это легко сделать, если ориентироваться на одно конкретное устройство. Но если вы помните, существует несколько размерных классов экранов и форм-факторов устройств, работающих под управлением iOS. К счастью, в iOS есть надежная инфраструктура автоматического размещения представлений с названием **Auto Layout**.

Механизм **Auto Layout** позиционирует представления внутри сцены, опираясь на predefined ограничения. В главе 2 приводится отличное описание **Auto Layout**. В нем рассказывается, как устанавливать ограничения с помощью редактора раскадровки в Xcode, и сейчас мы воспользуемся этой возможностью.

Выберите кнопку, щелкните на кнопке **Align** (Выровнять) – третья ниже и правее холста в редакторе раскадровки – и установите флажки **Horizontally in Container** (По горизонтали в контейнере) и **Vertically in Container** (По вертикали в контейнере). Щелкните на кнопке **Add 2 Constraints** (Добавить 2 ограничения), чтобы добавить эти ограничения.

После этого кнопка должна автоматически переместиться в центр представления. Таким способом мы сообщаем операционной системе iOS, что она должна поместить объект в центр контейнера по горизонтали и вертикали. Суть этих ограничений очевидна, но что такое «контейнер»? В данном случае контейнером является корневое представление, в котором находится объект, то есть корневое представление, в которое встроена вся сцена, определяемая свойством `view` класса `WelcomeViewController`. Если бы, например, кнопка находилась в другом представлении, она была бы помещена в центр этого вложенного представления.

Теперь, поместив кнопку в нужное место, займемся ее оформлением. Выберите кнопку и в инспекторе атрибутов введите текст **Catalog** в поле **Title**. Если теперь собрать и запустить приложение, вы должны увидеть кнопку в центре экрана.

Эта кнопка будет отображать каталог нашей библиотеки. Но прежде чем вывести этот каталог на экран, его нужно создать!

СПИСКИ, СПИСКИ И ЕЩЕ РАЗ СПИСКИ!

Отображение списков данных – одна из самых распространенных задач в разработке мобильных приложений. Существуют целые категории приложений, главной целью которых является вывод списков. Что такое каталог книг, как не гигантский список книг? К счастью, в Android и iOS есть несколько великолепных компонентов для работы со списками. В Android это `RecyclerView`, вхо-

дящий в состав библиотеки поддержки, но когда мы писали эти строки, он был перемещен в пакет `androidx`. На данный момент библиотека поддержки хорошо протестирована, и хотя можно ожидать некоторых улучшений или обновлений, я полагаю, что общедоступный API пакета `androidx` не претерпит существенных изменений.

Обратите внимание, что в Android компоненты этого типа прошли довольно долгий путь эволюционного развития. Первоначально имелся компонент `ListView`, прекрасно справлявшийся с прокруткой и отображением списков элементов, но предполагавший, что разработчики сами позаботятся о своевременном создании и удалении компонентов, представляющих элементы списка (что выгодно с точки зрения управления памятью). Имелся также компонент `GridView`, который, как вы, наверное, догадались, позволял выводить элементы в виде горизонтальных строк и вертикальных столбцов, а не только в форме вертикального списка, как `ListView`. На данный момент оба компонента считаются устаревшими и не рекомендуются для использования. Экземпляры `RecyclerView` используют в своей работе диспетчер макетов, с помощью которого отображают элементы в виде вертикального списка или по сетке (на самом деле можно использовать любой макет, который вы сможете придумать и реализовать). В iOS для вывода списков можно использовать компоненты `UITableView` и `UICollectionView`.

Добавление нового представления каталога

Начнем с того, что добавим новый контроллер пользовательского интерфейса с пустым списком в нем.

В Android создадим новый файл макета для `BrowseContentActivity`, содержащий только `RecyclerView`. Назовем его `res/layout/activity_browse.xml`. Мы можем также включить в макет панель инструментов для навигации и контекстные меню, а еще макет координатора для управления анимацией `SnackBar` или позицией `FloatingActionButton`, но не будем пока усложнять себе жизнь и начнем с самого простого:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/browse_content_recyclerview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/white" />
```

Назначим этот макет представлением содержимого в классе `BrowseContentActivity` внутри метода `onCreate`, добавив следующую строку сразу после вызова `super`:

Java

```
setContentView(R.layout.activity_browse);
```

Kotlin

```
setContentView(R.layout.activity_browse)
```

Позже мы добавим в код настройку источника данных и определим поведение и эффекты макета, но пока это все, что нам нужно.

В iOS все то же самое можно получить, практически не покидая редактор раскадровки. Для отображения каталога мы будем использовать компонент `UITableView`, потому что с ним немного проще работать. Сначала откройте файл *Main.storyboard* в Xcode и щелкните на кнопке **Library** (Библиотека) в окне проекта, чтобы открыть библиотеку объектов. Затем перетащите на холст новый контроллер представления и во вновь созданный экран этого контроллера перетащите табличное представление. Наконец, выберите табличное представление, щелкните на кнопке **Add New Constraints** (Добавить новые ограничения) в правом нижнем углу холста редактора раскадровки. Установите все поля равными 0 и щелкните на кнопке **Add 4 Constraints** (Добавить 4 ограничения); в результате этого табличное представление получит поля со всех сторон шириной 0 пикселей и займет всю ширину и высоту экрана.

В заключение выберите только что созданную сцену контроллера представления. В инспекторе атрибутов введите текст `Catalog` в поле **Title**, чтобы присвоить имя контроллеру представления.

Теперь в обоих наших приложениях есть представление для отображения списка, но они пока недоступны. Давайте исправим это!

Подключение кнопки

Какой смысл иметь кнопку, которая ничего не делает?

Оставим этот риторический вопрос и сосредоточимся на кнопке открытия каталога, которая присутствует на нашем экране приветствия, но пока ничего не делает. Цель этой кнопки, как мы только что сказали, – открыть каталог книг в библиотеке. Но сейчас она ничего не делает.

Свяжем эту кнопку с переходом. О переходах довольно подробно рассказывалось в главе 1. Фактически они описывают связь между двумя представлениями. Определяются переходы в раскадровке и могут запускаться через соединения в раскадровке или программно. Для простоты в этом примере мы выполним все соединения в редакторе раскадровки.

Наведите указатель мыши на кнопку, нажмите клавишу **Ctrl** и, удерживая ее, нажмите левую кнопку мыши. Перетащите соединение на нашу новую сцену с каталогом и отпустите кнопку мыши. В появившемся диалоге, в разделе **Action Segues** (Действие перехода), выберите **Show** (Показать).

Вот и все. Теперь, когда пользователь щелкнет на кнопке, приложение отобразит представление каталога. Соберите и запустите приложение, нажмите кнопку, и вы должны увидеть представление списка, как показано на рис. 17.4.

Кнопка подключена и работает! Теперь у нас есть представление списка, но пока в нем не отображаются никаких данных. Наконец, пришло время поговорить о самих данных.

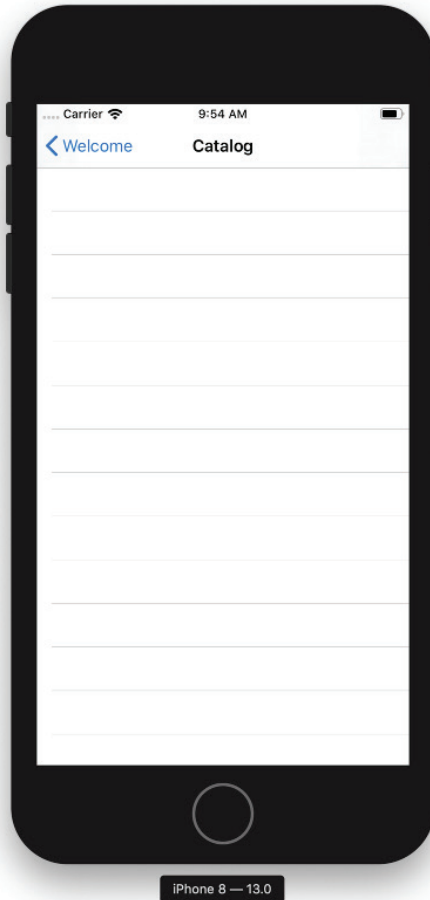


Рис. 17.4 ❖ Строки без данных – это метафора жизни?
Нет, просто пустое табличное представление

Книги

Если походить по библиотеке, для которой мы создаем приложение, можно заметить, что в ней **ОЧЕНЬ МНОГО** книг. Есть научно-популярные книги. Есть книги с картинками. Есть книги без картинок. Есть маленькие книги и большие книги, старые книги, книги для чтения на пляже, книги для чтения в ванной или за бокалом красного вина – есть самые разные книги!

Но какую информацию об этих книгах отображать?

К счастью, все книги имеют общие характеристики, среди которых:

- название;
- автор;
- номер ISBN;
- число страниц;
- жанр.

Чтобы вывести эту информацию в списке, нам нужно ее организовать. Конечно, можно передавать все эти свойства в строки списка по отдельности, но на практике для этой цели обычно создается модель данных, описывающая информацию, хранимую в этих данных, – гораздо проще добавить новое поле в общий тип, чем добавлять его в каждый метод, обрабатывающий информацию о книге.

Давайте определим объект, представляющий книгу, который мы сможем использовать для заполнения элементов списка. Итак, как такой объект может выглядеть в Java и Swift? Самое замечательное, что в обоих языках определение объекта выглядит почти одинаково.

Для представления книг в Android мы будем использовать стандартное определение класса. Такие классы иногда называют также «моделями» или «классами данных», но отметим, что в Kotlin последние определяются особым образом (как вы увидите ниже). На самом деле функционально модели ничем не отличаются от любых других классов. Некоторые разработчики предпочитают использовать «РОВО» («Plain Old Java Objects» – простые объекты Java) – классы, которые не наследуют никаких других классов и не реализуют никаких интерфейсов, но вообще это дело вкуса. Также в классах моделей не принято использовать логические операции (методов), кроме методов чтения/записи свойств, однако, с технической точки зрения, это совершенно необязательно.

Согласно рекомендациям Android, предлагающим не использовать общедоступные члены, мы определим поля для хранения информации, а также методы чтения/записи для доступа к этим полям. Вот определение нашего класса Book:

```
public class Book {  
  
    String mTitle;  
    String[] mAuthors;  
    String mIsbn;  
    int mPageCount;  
    boolean mIsFiction;  
  
    public String getTitle() {  
        return mTitle;  
    }  
  
    public void setTitle(String title) {  
        mTitle = title;  
    }  
  
    public String[] getAuthors() {  
        return mAuthors;  
    }  
  
    public void setAuthors(String[] authors) {  
        mAuthors = authors;  
    }  
  
    public String getIsbn() {  
        return mIsbn;  
    }  
}
```

```

public void setIsbn(String isbn) {
    mIsbn = isbn;
}

public int getPageCount() {
    return mPageCount;
}

public void setPageCount(int pageCount) {
    mPageCount = pageCount;
}

public boolean isFiction() {
    return mIsFiction;
}

public void setFiction(boolean fiction) {
    mIsFiction = fiction;
}
}

```

Такое определение может показаться немного громоздким для ввода вручную, однако Android Studio с радостью возьмет рутину на себя и сгенерирует методы чтения/записи из свойств и даже правильно подберет имена для них, следуя правилам венгерской нотации (то есть для свойства `mName` будут сгенерированы методы `getName` и `setName`, а не `getMname`).

А теперь обратимся к Kotlin, который блистательно проявляет себя в этом примере, позволяя записать намного более компактное определение класса `Book`:

```

data class Book (
    val title: String,
    val authors: List<String>,
    val isbn: String,
    val pageCount: Int,
    val isFiction: Boolean
)

```

Дополнительно можно включить определение конструктора без аргументов, чтобы получить возможность создавать пустые экземпляры `Book`:

```

data class Book (
    val title: String,
    val authors: List<String>,
    val isbn: String,
    val pageCount: Int,
    val isFiction: Boolean
) {
    constructor() : this("", mutableListOf<String>(), "", 0, false) {
        // ничего не делает
    }
}

```

Этот код не только включает конструктор по умолчанию, принимающий все поля, но также объединяет его с самим определением. Кроме того, все поля определены как значения `val` и, следовательно, являются неизменяемыми (в Java такие поля можно определить с ключевым словом `final`), однако, как вы увидите далее, нам не требуется устанавливать значения по умолчанию, которые будут изменяться при создании экземпляра. Эти значения определяются в конструкторе, что, в свою очередь, гарантирует наличие единственного неизменного значения для каждого члена.

Теперь перейдем в Xcode. Добавьте в проект новый файл Swift с именем *Book.swift*, выбрав в меню пункт **File** ⇒ **New** ⇒ **File** (Файл ⇒ Создать ⇒ Файл). Мы могли бы определить класс, но не будем усложнять и создадим структуру. Классы в Swift являются ссылочными объектами; иначе говоря, передавая объект, вы фактически передаете ссылку на него. Структуры, напротив, являются объектами-значениями, похожими на простые типы данных, такие как строки. Это делает операции копирования безопаснее и проще в многопоточном окружении. Вам не нужно беспокоиться о том, что другой объект изменит данные в этом объекте, потому что копироваться будет сам объект, а не его адрес.

Вот как выглядит определение объекта *Book* на языке Swift:

```
struct Book {
    let title: String
    let authors: [String]
    let isbn: String
    let pageCount: Int
    let fiction: Bool
}
```

Неизменяемые свойства

В структуре *Book*, которую мы только что определили, все свойства определены с ключевым словом `let`. В языке Swift это означает, что они, эти свойства, неизменяемые, то есть их нельзя изменить. Это также означает, что их значения должны передаваться в инициализатор. К счастью, компилятор способен правильно интерпретировать объявления структур и создавать инициализаторы по умолчанию, принимающие все свойства, поэтому нет необходимости писать свой шаблонный код.

Аналогично, в примере на Kotlin мы объявили свойства-члены с ключевым словом `val`, дающим тот же результат, который только что был описан. После установки `val`-свойства его нельзя изменить, хотя это не относится к содержимому свойств, таких как `Array` или `List`. Свойства `val` этих типов тоже нельзя изменить после установки, но значения элементов массива или списка изменить можно. Обратите внимание, что если бы мы использовали `var` вместо `val`, свойства-члены были бы изменяемыми, и их можно было бы изменить позже.

В Java того же эффекта можно добиться с помощью ключевого слова `final`, но тогда придется немного видоизменить класс, чтобы заставить его работать, как задумано (например, в примерах на Swift и Kotlin вместо использования методов записи потребовалось бы назначать значения всем свойствам только один раз, в конструкторе).

Итак, мы определили модели данных, создали представление каталога и подключили кнопку вызова этого каталога. Теперь, когда у нас есть все не-

обходимое, чтобы приступить к реализации отображения данных, вернемся к нашим новым представлениям списка.

ЗАПОЛНЕНИЕ ПРЕДСТАВЛЕНИЯ СПИСКА

В настоящее время мы не получаем данных ни из сетевой службы, ни из файловой системы. (Об этом мы поговорим в следующих главах!) Поэтому используем пока статические данные в классе, чтобы с их помощью продемонстрировать, как правильно заполнять представления списков в Android и iOS. Начнем с Android.

Android

Итак, у нас есть экземпляр RecyclerView, и мы готовы пустить его в дело, но, пока нет данных, мы ничего не увидим. Исправим эту проблему. Временно определим некоторые данные прямо в коде, в виде статической переменной класса, *но, ради всего святого, никогда не делайте этого в промышленном коде*. И не волнуйтесь, мы ликвидируем данный недостаток позже.

Вот несколько примеров книг – вскоре мы используем аналогичную структуру в iOS:

Java

```
public class Book {  
    public static final Book[] SAMPLE_DATA = {  
        new Book("Dragons Love Tacos",  
            new String[]{"Adam Rubin", "Daniel Salmieri"},  
            "978-0803736801", 40, true),  
        new Book("Fight Club",  
            new String[]{"Chuck Palahniuk"},  
            "978-0393039764", 208, true),  
        new Book("2001: A Space Odyssey",  
            new String[]{"Arthur C. Clarke"},  
            "978-0451457998", 296, true),  
        new Book("Ulysses",  
            new String[]{"James Joyce"},  
            "978-1420953961", 682, true),  
        new Book("Catch-22",  
            new String[]{"Joseph Heller"},  
            "978-1451626650", 544, true),  
        new Book("The Stand",  
            new String[]{"Stephen King"},  
            "978-0307947307", 1200, true),  
        new Book("On The Road",  
            new String[]{"Jack Kerouac"},  
            "978-0143105466", 416, true),  
        new Book("Heart of Darkness",  
            new String[]{"Joseph Conrad"},  
            "978-1503275928", 78, true),
```

```

    new Book("A Brief History of Time",
            new String[]{"Stephen Hawking"},
            "978-0553380163", 212, false),
    new Book("Dispatches",
            new String[]{"Michael Herr"},
            "978-0679735250", 272, false),
    new Book("Harry Potter and Prisoner of Azkaban",
            new String[]{"J.K. Rowling"},
            "978-0439136365", 448, true)
};

private final String mTitle;
private final String[] mAuthors;
private final String mIsbn;
private final int mPageCount;
private final boolean mIsFiction;

public Book(String title, String[] authors, String isbn, int pageCount,
            boolean isFiction) {
    mTitle = title;
    mAuthors = authors;
    mIsbn = isbn;
    mPageCount = pageCount;
    mIsFiction = isFiction;
}

public Book() {}

public String getTitle() {
    return mTitle;
}

public String[] getAuthors() {
    return mAuthors;
}

public String getIsbn() {
    return mIsbn;
}

public int getPageCount() {
    return mPageCount;
}

public boolean isFiction() {
    return mIsFiction;
}
}

```

Kotlin

```

data class Book(val title: String, val authors: List<String>,
               val isbn: String, val pageCount: Int,
               val isFiction: Boolean) {

    constructor() : this("", mutableListOf<String>(), "", 0, false) {
        // ничего не делает
    }
}

```



```

companion object {
    val SAMPLE_DATA = arrayOf(
        Book("Fight Club", listOf("Chuck Palahniuk"),
            "978-0393039764", 208, true),
        Book("2001: A Space Odyssey",
            listOf("Arthur C. Clarke"), "978-0451457998", 296, true),
        Book("Ulysses",
            listOf("James Joyce"), "978-1420953961", 682, true),
        Book("Catch-22",
            listOf("Joseph Heller"), "978-1451626650", 544, true),
        Book("The Stand",
            listOf("Stephen King"), "978-0307947307", 1200, true),
        Book("On The Road",
            listOf("Jack Kerouac"), "978-0143105466", 416, true),
        Book("Heart of Darkness",
            listOf("Joseph Conrad"), "978-1503275928", 78, true),
        Book("A Brief History of Time",
            listOf("Stephen Hawking"), "978-0553380163", 212, false),
        Book("Dispatches",
            arrayOf("Michael Herr"), "978-0679735250", 272, false),
        Book("Harry Potter and Prisoner of Azkaban",
            listOf("J.K. Rowling"), "978-0439136365", 448, true),
        Book("Dragons Love Tacos",
            listOf("Adam Rubin", "Daniel Salmieri"), "978-0803736801",
            40, true))
    }
}

```

Это наши данные. Имейте в виду, что данные для вывода в представлении списка могут меняться с течением времени – мы могли бы получать их с удаленного сервера, а затем показывать кешированную версию из локальной базы данных. Также у нас может иметься служба поиска, позволяющая сортировать или фильтровать данные. По этой и другим причинам источник данных не подключен напрямую к RecyclerView, а управляется специальным компонентом, образующим мост между пользовательским интерфейсом и данными, который часто называют *адаптером*. Шаблон «Адаптер» широко используется в Android в сочетании, например, с ViewPager, Spinner, Gallery и многими другими; конкретный класс Adapter, используемый представлением RecyclerView, – это RecyclerView.Adapter.

Класс Adapter реализует несколько контрактов. Сначала мы продемонстрируем действующий пример с массивом Book.SAMPLE_DATA в качестве источника данных, а затем углубимся в реализации конкретных методов:

Java

```

public class BrowseBooksAdapter
    extends RecyclerView.Adapter<BrowseBooksAdapter.ViewHolder> {
    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        return new ViewHolder(new TextView(parent.getContext()));
    }
}

```

```

@Override
public void onBindViewHolder(ViewHolder holder, int position) {
    holder.mTextView.setText(Book.SAMPLE_DATA[position].getTitle());
}

@Override
public int getItemCount() {
    return Book.SAMPLE_DATA.length;
}

public static class ViewHolder extends RecyclerView.ViewHolder {
    private TextView mTextView;
    public ViewHolder(@NonNull View itemView) {
        super(itemView);
        mTextView = (TextView) itemView;
    }
}
}

```

Kotlin

```

class BrowseBooksAdapter() :
    RecyclerView.Adapter<BrowseBooksAdapter.ViewHolder>() {
    override fun onCreateViewHolder(parent: ViewGroup,
        viewType: Int): ViewHolder {
        return ViewHolder(TextView(parent.context))
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val book = Book.SAMPLE_DATA[position]
        holder.textView.text = book.title
    }

    override fun getItemCount(): Int {
        return Book.SAMPLE_DATA.size
    }

    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val textView: TextView = itemView as TextView
    }
}

```

Если этот код кажется вам запутанным и непрозрачным, то вы правы! Фактически идея RecyclerView изначально была просто *соглашением*. Мы (разработчики Android) использовали раньше виджет ListView, но со временем пришли к выводу, что использование определенных шаблонов позволяет экономить память и увеличивать производительность. Без шаблона повторного использования элементов списка простой поиск, возвращающий несколько тысяч элементов, легко мог бы потребить больше ресурсов, чем доступно приложению, особенно на устройствах, выпускавшихся в то время.

Когда класс RecyclerView был включен в библиотеку поддержки (до того, как мы начали использовать современный пакет androidX), это соглашение приобрело осязаемое воплощение в коде, но без понимания исторических предпосылок некоторые из этих шаблонов и контрактов выглядят малопонятными и непрозрачными!

В любом случае, в настоящее время в Android имеется проверенный и высокопроизводительный виджет, поэтому потратим немного времени на его изучение. Начнем с самого низа – статического класса `ViewHolder`. Суперкласс `RecyclerView.ViewHolder` является абстрактным, поэтому, даже когда не требуется расширять его поведение, вам все равно придется определить его подкласс. Если бы суперкласс не был абстрактным, мы могли бы просто привести элемент `itemView` к типу `TextView` в методе `onBindViewHolder` без определения своего подкласса. Кажется, мы сделали слишком большой шаг! Давайте отступим немного назад и пойдем медленнее.

В самом простом случае `ViewHolder` – это просто пакет ссылок на экземпляры `View` дочерних представлений. Например, если в каждом элементе вашего списка имеется `ImageView` для отображения миниатюрного изображения, `TextView` для обозначения заголовка и еще один `TextView` для отображения списка авторов книги, экземпляр `ViewHolder` будет включать ссылки на эти три представления, как показано ниже:

Java

```
public static class ViewHolder extends RecyclerView.ViewHolder {
    private ImageView mThumbnailView;
    private TextView mTitleView;
    private TextView mAuthorsView;
    public ViewHolder(@NonNull View itemView) {
        super(itemView);
        mThumbnailView = itemView.findViewById(R.id.row_thumb);
        mTitleView = itemView.findViewById(R.id.row_title);
        mAuthorsView = itemView.findViewById(R.id.row_authors);
    }
}
```

Kotlin

```
class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    val thumbnailView: ImageView = itemView.findViewById(R.id.row_thumb)
    val titleView: TextView = itemView.findViewById(R.id.row_title)
    val authorsView: TextView = itemView.findViewById(R.id.row_authors)
}
```

У вас может возникнуть справедливый вопрос: откуда берутся эти экземпляры `View`? Отложим на время `ViewHolder` и обратимся к методу `onCreateViewHolder`. В предыдущем примере мы просто передаем `TextView` конструктору, но в более сложном пользовательском интерфейсе, например в котором мы использовали для объяснения `ViewHolder`, нужно создать дерево представлений, которое можно определить в макете XML. Каждый элемент списка, например, может выглядеть так:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
```

```

<ImageView
    android:id="@+id/row_thumb"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

<LinearLayout
    android:orientation="vertical"
    android:layout_width="0dp"
    android:layout_weight="1"
    android:layout_height="wrap_content">

    <TextView
        android:id="@+id/row_title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/row_authors"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
</LinearLayout>

```

Обратите внимание на трюк с атрибутом `weight` в `LinearLayout`, упоминавшийся в главе 2.

Сохраните этот листинг в файле `complex_book_row.xml` в папке `res/layout`.

При использовании таких сложных элементов списка в `ViewHolder` наш метод `onCreateViewHolder` будет выглядеть так:

Java

```

@Override
public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    LayoutInflater inflater = LayoutInflater.from(parent.getContext());
    View itemView = inflater.inflate(R.layout.complex_book_row, parent, false);
    return new ViewHolder(itemView);
}

```

Kotlin

```

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    val inflater = LayoutInflater.from(parent.context)
    val itemView = inflater.inflate(R.layout.complex_book_row, parent, false)
    return ViewHolder(itemView)
}

```

Теперь должно быть понятнее, как все работает. Если нет, попробуйте представить, как `RecyclerView` повторно использует элементы...

При первом размещении `RecyclerView` использует метод `onCreateViewHolder`, а также некоторые внутренние функции, чтобы создать достаточное число элементов для заполнения видимой области. Когда пользователь прокручивает страницу и первый элемент выходит за границы видимой части представления, все дерево этого элемента *перерабатывается* (`recycled`) и подготавливается для повторного использования – оно отсоединяется от окна и добавляется

в пул свободных элементов для последующего использования. Когда невидимый элемент списка оказывается в видимой области представления, из-за прокрутки или изменения размера RecyclerView, вызывается метод `onBindViewHolder` объекта `RecyclerView.Adapter`, который получает позицию (она часто используется в качестве индекса для обращения к источнику данных) и заполняет недавно скрытое, но теперь вновь используемое дерево представлений. В нашем сложном примере метод `onBindViewHolder` мог бы выглядеть так:

Java

```
@Override
public void onBindViewHolder(ViewHolder holder, int position) {
    Book book = Book.SAMPLE_DATA[position];
    // допустим, что можно использовать ISBN книги для загрузки с сервера
    // миниатюры с изображением обложки. Сохраним изображение в переменной bitmap
    Bitmap bitmap = // ...
    holder.mThumbnail.setImageBitmap(bitmap);
    holder.mTitleView.setText(book.getTitle());
    holder.mAuthorsView.setText(TextUtils.join(" ", book.getAuthors()));
}
```

Kotlin

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val book = Book.SAMPLE_DATA[position]
    // допустим, что можно использовать ISBN книги для загрузки с сервера
    // миниатюры с изображением обложки. Сохраним изображение в переменной bitmap
    val bitmap = // ...
    holder.thumbnailView.imageBitmap = bitmap
    holder.titleView.text = book.title
    holder.authorsView.text = TextUtils.join(" ", book.authors)
}
```

Теперь нетрудно понять, как работает простой пример, сравнив его с более полной версией.

Нам осталось только связать все воедино, а также сообщить представлению RecyclerView, как позиционировать дочерние элементы. Как уже говорилось выше, существуют диспетчеры компоновки для вертикальных списков, горизонтальных списков и представлений в виде сетки, а также есть возможность определить свой подкласс `LayoutManager` с настраиваемым поведением.

Выбрав и создав экземпляр `LayoutManager`, свяжите его с `RecyclerView`:

Java

```
recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

Kotlin

```
browse_content_recyclerview.layoutManager = LinearLayoutManager(this)
```

и подключите экземпляр `Adapter` к `RecyclerView` с помощью `setAdapter`:

Java

```
recyclerView.setAdapter(new BrowseBooksAdapter());
```

Kotlin

```
browse_content_recyclerview.adapter = BrowseBooksAdapter()
```

Вот первая черновая версия Activity для просмотра каталога:

Java

```
public class BrowseContentActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_browse);
        RecyclerView recyclerView = findViewById(R.id.browse_content_recyclerview);
        recyclerView.setLayoutManager(new LinearLayoutManager(this));
        recyclerView.setAdapter(new BrowseBooksAdapter());
    }
}
```

Kotlin

```
class BrowseContentActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_browse)
        browse_content_recyclerview.adapter = BrowserBooksAdapter()
        browse_content_recyclerview.layoutManager = LinearLayoutManager(this)
    }
}
```

Если сейчас запустить приложение и нажать кнопку, запускающую BrowseBooksActivity, должен появиться вертикальный список с определенными нами книгами. Если размер шрифта достаточно велик или экран достаточно мал (вы можете поэкспериментировать с отступами или полями в TextView), вы увидите, что этот список прокручивается так, будто весь набор данных доступен постоянно, – это волшебство RecyclerView.

Мы получили работающий список книг в Android. Давайте теперь создадим такой же список в iOS!

iOS

Как рассказывалось выше, для отображения каталога книг в iOS мы будем использовать UITableView. Собираясь реализовать свою логику, первое, что мы должны сделать, – заменить UIViewController своим собственным контроллером представления в редакторе раскадровки. Создайте в Xcode новый объект Cocoa Touch Class с именем CatalogViewController, наследующий UIViewController (**File** ⇒ **New** ⇒ **File** (Файл ⇒ Создать ⇒ Файл)), и добавьте его в проект. Оставьте пока этот файл как есть и выберите файл *Main.storyboard*, чтобы открыть его в редакторе раскадровки.

В редакторе раскадровки выберите сцену каталога. В инспекторе идентичности введите в поле **Class** имя только что созданного класса CatalogViewController.

Взглянув на табличное представление внутри сцены, можно заметить, что сейчас оно пустое. Если бы мы попытались использовать табличное представление прямо сейчас, у нас ничего не получилось бы, потому что в настоящее время отсутствуют прототипы ячеек. Прототипы ячеек – это шаблоны, определяемые в раскадровке, экземпляры которых автоматически создаются табличным представлением для отображения содержимого элемента списка. Каждый элемент может включать различные типы ячеек. Прототипы определяются и используются всеми ячейками.

Чтобы добавить прототип ячейки, выберите табличное представление в сцене, откройте инспектор атрибутов и увеличьте значение в поле **Prototype Cells**. Эта операция добавит новый элемент в таблицу, отображаемую в редакторе раскадровки. Однако, чтобы использовать ячейки этого типа, нам нужна возможность ссылаться на этот тип в коде. Для этого выберите ячейку в табличном представлении в редакторе и присвойте ей уникальный идентификатор в поле **Identifier**. Пока присвоим идентификатор `CatalogTableViewCell`.

Выше мы решили, что в этой главе будут использоваться статические данные. Давайте используем те же книги, что были перечислены в разделе «Android». Дополнительно расширим структуру `Book` и добавим расширение с удобным вспомогательным методом:

```
struct Book {
    let title: String
    let authors: [String]
    let isbn: String
    let pageCount: Int
    let fiction: Bool
}

extension Book {
    static let sampleData: [Book] = [
        Book(title: "Fight Club", authors: ["Chuck Palahniuk"],
            isbn: "978-0393039764", pageCount: 208, fiction: true),
        Book(title: "2001: A Space Odyssey", authors: ["Arthur C. Clarke"],
            isbn: "978-0451457998", pageCount: 296, fiction: true),
        Book(title: "Ulysses", authors: ["James Joyce"],
            isbn: "978-1420953961", pageCount: 682, fiction: true),
        Book(title: "Catch-22", authors: ["Joseph Heller"],
            isbn: "978-1451626650", pageCount: 544, fiction: true),
        Book(title: "The Stand", authors: ["Stephen King"],
            isbn: "978-0307947307", pageCount: 1200, fiction: true),
        Book(title: "On The Road", authors: ["Jack Kerouac"],
            isbn: "978-0143105466", pageCount: 416, fiction: true),
        Book(title: "Heart of Darkness", authors: ["Joseph Conrad"],
            isbn: "978-1503275928", pageCount: 78, fiction: true),
        Book(title: "A Brief History of Time", authors: ["Stephen Hawking"],
            isbn: "978-0553380163", pageCount: 212, fiction: false),
        Book(title: "Dispatches", authors: ["Michael Herr"],
            isbn: "978-0679735250", pageCount: 272, fiction: false),
        Book(title: "Harry Potter and Prisoner of Azkaban",
            authors: ["J.K. Rowling"],
            isbn: "978-0439136365", pageCount: 448, fiction: true),
```

```

    Book(title: "Dragons Love Tacos",
      authors: ["Adam Rubin", "Daniel Salmieri"],
      isbn: "978-0803736801", pageCount: 40, fiction: true),
  ]
}

```

Хотя пока это не очевидно, но мы не будем передавать сразу все данные в табличное представление. Одной из удивительных особенностей UITableView и UICollectionView является их способность обслуживать огромные объемы данных, потому что они используют только часть данных, почти так же, как RecyclerView в Android.

Чтобы заполнить табличное представление, нам нужно создать источник данных. Обычно для этого создается объект, реализующий протокол UITableViewDataSource, и связывается с объектом табличного представления. Однако в этом примере мы не будем усложнять и реализуем протокол непосредственно в CatalogViewController.

! Обычно контроллеры UIViewController сами реализуют протоколы источников данных для табличных представлений. Это нормально для простых приложений, но такой подход нарушает принцип разделения ответственности, проповедуемый более строгой архитектурой MVC. Кроме того, это приводит к раздуванию кода контроллера, что является распространенной проблемой. Разработчик, будь осторожен!

Добавим расширение для CatalogViewController, реализующее UITableViewDataSource:

```

class CatalogViewController: UIViewController {
    ...
}

extension CatalogViewController: UITableViewDataSource {
}

```

Протокол определяет ряд необязательных методов, но кроме них есть два обязательных метода, которые вы обязаны реализовать для заполнения табличного представления: tableView(_:numberOfRowsInSection:) и tableView(_:cellForRowAt:). Рассмотрим первый из них.

Табличное представление будет вызывать tableView(_:numberOfRowsInSection:), чтобы определить количество элементов в каждом разделе данных. В нашем табличном представлении будет только один раздел, поэтому можно просто вернуть количество элементов в статических данных, которые мы определили ранее:

```

func tableView(_ tableView: UITableView,
  numberOfRowsInSection section: Int) -> Int {
    return Book.sampleData.count
}

```

i По умолчанию табличное представление содержит только один раздел. Чтобы определить несколько разделов (с их заголовками), нужно реализовать метод numberOfSections(in:).

В этом методе мы сообщаем табличному представлению количество элементов. Теперь реализуем следующий метод, `tableView(_:cellForRowAt:)`, определяющий логику заполнения ячеек.

Этот метод должен: создать экземпляр ячейки табличного представления и заполнить эту ячейку соответствующими данными. Вот, собственно, сам метод:

```
func tableView(_ tableView: UITableView,
              cellForRowAt indexPath: IndexPath) -> UITableViewCell {

    // Получить ячейку из пула свободных ячеек
    let cell =
        tableView.dequeueReusableCell(withIdentifier: "CatalogTableViewCell",
                                    for: indexPath)

    // Найти книгу, соответствующую заполняемой ячейке
    let book = Book.sampleData[indexPath.row]

    // Заполнить метку с заголовком ячейки названием книги
    cell.textLabel?.text = book.title

    return cell
}
```

Первым делом мы получаем экземпляр ячейки для табличного представления из пула незанятых ячеек, используя при этом идентификатор (`CatalogTableViewCell`), присвоенный при создании прототипа ячейки. Затем получаем книгу из источника данных, используя свойство `indexPath.row`. Этот метод вызывается для каждой отображаемой ячейки, поэтому при извлечении третьего элемента это свойство будет иметь значение 2 и мы сможем извлечь третий элемент из массива (т. е. это порядковый номер элемента, если считать с нуля). Затем мы заполняем текстовую метку в ячейке, записывая в нее название книги, и возвращаем ячейку, чтобы табличное представление могло отобразить ее.

Наконец, нам осталось сделать еще один шаг: сообщить табличному представлению, что оно может использовать этот контроллер представления в роли источника данных.

Вернитесь к файлу *Main.storyboard* в редакторе раскадровки. Нажмите и удерживайте клавишу **Control**, наведите указатель мыши на таблицу в сцене каталога, нажмите левую кнопку мыши и перетащите объект `Catalog` в структуру документа слева от холста. В появившемся диалоге выберите `dataSource` в разделе **Outlets**, чтобы связать табличное представление с контроллером представления в качестве источника данных.

Щелкните на кнопке **Build and Run** (Собрать и запустить) и проверьте, как работает кнопка перехода к каталогу: после нажатия этой кнопки должен открыться список с примерами данных.

Что мы узнали

Теперь кратко вспомним, что мы узнали в этой главе.

Сначала мы узнали, как структурировать наши экраны и добавить немного оформления. Затем узнали, как организовать переходы между представления-

ми с использованием кнопок. Обсудили модели данных и увидели, насколько похожими они могут быть в разных платформах. Мы также создали простой объект модели, представляющий книгу, который будем использовать и улучшать в последующих главах. Наконец, мы определили данные и отобразили их в виде списка на экране каталога библиотеки.

Мы сделали это! Мы реализовали в нашем приложении отображение названий книг, пусть и хранящихся в форме статических данных. До сих пор мы развлекались на уровне представлений и контроллеров, но в следующей главе мы углубимся в работу с моделями данных. Вперед, друзья!

Глава 18

Моделирование каталога библиотеки

К настоящему моменту мы продвинулись далеко вперед в создании нашего приложения для библиотеки; создали несколько основных структурных элементов для приложения, реализовали переход между экранами и вывод некоторых данных. К сожалению, пока мы используем статические данные, жестко зашитые в код, но вскоре мы исправим это. Чтобы продолжить разработку остальной части приложения, нам нужно вернуться к модели данных и внимательно рассмотреть ее. Это позволит нам создать нечто более надежное и удобное в обслуживании.

Сразу отметим, что статические данные – это не страшно. Фактически мы будем продолжать использовать «статические» данные, пока не перейдем к реализации сетевых операций в главе 20. Основная проблема в текущей реализации обслуживания списка книг заключается в сложности переноса данных в другие места внутри приложения.

Сейчас мы отображаем книги только в виде списка. Однако потом нам потребуется искать книги, сохранять их в закладках и многое другое. Давайте тщательно продумаем организацию данных в приложении, более приспособленную к будущим изменениям.

ДИНАМИЧЕСКИЕ ДАННЫЕ В ПРЕДСТАВЛЕНИЯХ СПИСКОВ

Наш список книг – отличная отправная точка для поиска способов более динамичного отображения данных. Как вы наверняка помните, в настоящее время список книг хранится во временном свойстве `SAMPLE_DATA` основного объекта `Book`, что упрощает его использование.

Однако, добавляя новые представления для отображения данных, мы быстро обнаружим, что такой подход ведет к дублированию кода, что довольно неудобно для обслуживания. Давайте внесем небольшие усовершенствования, добавив слой между нашим контроллером пользовательского интерфейса и списком.

Android

В некоторых компонентах Android, таких как медиаплеер `ExoPlayer`, поддерживается понятие «источника данных». Также в Android существует понятие «контент-провайдера» – компонента, например, предоставляющего информацию о контактах в телефоне для использования в вашем приложении. Однако компоненты, которые используют шаблон «Адаптер», обычно оставляют возможность использовать произвольные источники данных. Данные могут быть определены непосредственно в коде, в виде списка объектов, в XML-файле на устройстве, в документе JSON, хранящемся в памяти или получаемом от веб-службы.

В нашей пробной версии приложения мы просто добавили все наши данные в статический массив объектов `Book`. Несмотря на определенные удобства, такой массив сложно поддерживать в актуальном состоянии – всякий раз, когда в библиотеку поступит новая книга, нам придется обновить этот массив и опубликовать новую версию приложения, надеясь, что все наши пользователи прилежно обновят его. В долгосрочной перспективе мы, скорее всего, придем к использованию веб-службы, которая может обновляться администраторами библиотеки, и будем извлекать список книг из нее, периодически обращаясь к соответствующим конечным точкам службы. Также мы непременно захотим сохранить данные локально, а если у нас появится желание реализовать возможность сортировки или фильтрации с использованием хорошо известного и проверенного SQL, мы наверняка захотим преобразовать ответы JSON, возвращаемые веб-службой, в записи в базе данных.

Но все это – отдаленная перспектива, а пока давайте создадим простой интерфейс, предоставляющий необходимую информацию нашему адаптеру, и обновим адаптер, чтобы он соответствовал этому контракту. После этого мы сможем вносить изменения по мере развития наших требований и возможностей.

Давайте взглянем на наш адаптер еще раз и посмотрим, как его изменить, чтобы он удовлетворял изменившимся требованиям:

Java

```
public class
BrowseBooksAdapter extends RecyclerView.Adapter<BrowseBooksAdapter.ViewHolder> {

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        return new ViewHolder(new TextView(parent.getContext()));
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        holder.mTextView.setText(Book.SAMPLE_DATA[position].getTitle());
    }

    @Override
    public int getItemCount() {
        return Book.SAMPLE_DATA.length;
    }
}
```

```
public static class ViewHolder extends RecyclerView.ViewHolder {
    private TextView mTextView;
    public ViewHolder(@NonNull View itemView) {
        super(itemView);
        mTextView = (TextView) itemView;
    }
}
```

Kotlin

```
class BrowseBooksAdapter :
    RecyclerView.Adapter<BrowseBooksAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        return ViewHolder(TextView(parent.context))
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.mTextView.text = Book.SAMPLE_DATA[position].title
    }

    override fun getItemCount(): Int {
        return Book.SAMPLE_DATA.size
    }

    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        private val mTextView: TextView
        init {
            mTextView = itemView as TextView
        }
    }
}
```

Как видите, мы дважды обращаемся к статическому источнику данных – один раз при прокрутке представления, когда требуется обновить видимый элемент (в данном случае название книги), и второй раз, чтобы определить общее количество книг в нашем каталоге и RecyclerView смог узнать, когда прекратить прокрутку. Интерфейс подобного источника данных можно было бы определить так:

Java

```
public interface BookDataSource {
    Book get(int index);
    int size();
}
```

Kotlin

```
interface BookDataSource {
    operator fun get(index: Int): Book
    val size: Int
}
```

Теперь изменим адаптер, чтобы он мог принимать любой класс, реализующий этот интерфейс:

Java

```

public class
BrowseBooksAdapter extends RecyclerView.Adapter<BrowseBooksAdapter.ViewHolder> {
    private final BookDataSource mSource;

    public BrowseBooksAdapter(BookDataSource source) {
        super();
        mSource = source;
    }

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        return new ViewHolder(new TextView(parent.getContext()));
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        Book book = mSource.get(position);
        holder.textView.setText(book.getTitle());
    }

    @Override
    public int getItemCount() {
        return mSource.size();
    }

    public static class ViewHolder extends RecyclerView.ViewHolder {
        private TextView textView;
        public ViewHolder(@NonNull View itemView) {
            super(itemView);
            textView = (TextView) itemView;
        }
    }
}

```

Kotlin

```

class BrowseBooksAdapter(private val source: BookDataSource) :
    RecyclerView.Adapter<BrowseBooksAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        return ViewHolder(TextView(parent.context))
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val book = source[position]
        holder.textView.text = book.title
    }

    override fun getItemCount(): Int {
        return source.size
    }

    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val textView: TextView = itemView as TextView
    }
}

```

Ничего сложного! Если мы захотим продолжать использовать статический массив, можно создать простой класс, реализующий интерфейс `BookDataSource`, но использующий список, зашитый в код:

Java

```
public class HardCodedBookDataSource implements BookDataSource {
    public Book get(int index) {
        return Book.SAMPLE_DATA[index];
    }

    public int size() {
        return Book.SAMPLE_DATA.length;
    }
}
```

Kotlin

```
class HardCodedBookDataSource : BookDataSource {
    override val size: Int
    get() = Book.SAMPLE_DATA.size

    override operator fun get(index: Int): Book {
        return Book.SAMPLE_DATA[index]
    }
}
```

Возможно, вы не заметили, но контракт, определяемый интерфейсом `BookDataSource`, неявно поддерживается всеми реализациями `List<Book>` – то есть предоставить источник данных можно, как показано ниже:

Java

```
public class Books extends ArrayList<Book> implements BookDataSource {}
```

Kotlin

```
class Books : ArrayList<Book>(), BookDataSource
```

Определив источник данных таким способом, вы автоматически получите в свое распоряжение все замечательные операции, поддерживаемые классом `List`, такие как `add`, `addAll`, `set`, `remove`, `subList` и т. д.

И даже притом что мы все еще используем статический список книг, нам не придется вносить никаких изменений в наш `RecyclerView` или адаптер `BrowseBooksAdapter`, за исключением использования нового источника в адаптере, который может получать данные от веб-службы или читать их из локальной базы данных. Отлично! Наш список стал более динамичным, даже притом что использует те же данные, что и раньше.

Мы продолжим подготовку к удалению статического массива книг далее в этой главе, а сейчас посмотрим, как то же самое реализовать в iOS.

iOS

По аналогии с реализацией шаблона «Адаптер» в Android, нам нужно создать некоторый промежуточный слой, отделяющий контроллер от исходных дан-

ных. С этой целью можно добавить новый объект, с которым будет взаимодействовать контроллер. Как вы наверняка помните, в контроллере `CatalogViewController` мы реализовали поддержку протокола `UITableViewDataSource` для передачи данных в табличное представление.

Это помогло нам на первом этапе, но почему бы не создать отдельный объект для прямой передачи данных в табличное представление? Контроллер мог бы управлять этим объектом источника данных и табличным представлением. Так мы сможем отделить контроллер от представления – в данном случае от табличного представления – и слоя данных.

Добавим в проект новый файл с классом `ListDataSource`, выбрав в меню пункт **File** ⇒ **New** ⇒ **File** (Файл ⇒ Создать ⇒ Файл). Этот новый класс будет играть роль источника данных для нашего табличного представления. Пока этот файл содержит довольно простой код:

```
import UIKit

class ListDataSource: NSObject {
}

extension ListDataSource: UITableViewDataSource {
    func tableView(_ tableView: UITableView,
                  numberOfRowsInSection section: Int) -> Int {
        return Book.sampleData.count
    }

    func tableView(_ tableView: UITableView,
                  cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        // Получить ячейку из пула свободных ячеек
        let cell =
            tableView.dequeueReusableCell(withIdentifier: "CatalogTableViewCell",
                                       for: indexPath)

        // Найти книгу, соответствующую заполняемой ячейке
        let book = Book.sampleData[indexPath.row]

        // Заполнить метку с заголовком ячейки названием книги
        cell.textLabel?.text = book.title

        return cell
    }
}
```

Если часть этого кода показалась вам знакомой, то вы не ошиблись! Это тот же код, который использовался в `CatalogViewController` для отображения списка книг в представлении каталога. Поместив этот код в отдельный файл, мы можем теперь удалить весь код поддержки протокола `UITableViewDataSource` из `CatalogViewController`. Определение класса `CatalogViewController` теперь должно выглядеть так:

```
class CatalogViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```



```

    // Выполните здесь дополнительные настройки представления.
  }
}

```

Если теперь попытаться собрать и запустить приложение, вы обнаружите, что из этого ничего хорошего не получится. Класс `CatalogViewController` лишился методов реализации протокола источника данных, необходимых табличному представлению, но связь между ним и представлением все еще определена в `Main.Storyboard`. Из-за этого возникает ошибка SIGABORT и приложение завершается.

Печально. Давайте посмотрим, можно ли исправить эту проблему. Попутно, исправляя проблему, посмотрим также, можно ли заставить табличное представление в `CatalogViewController` напрямую использовать наш новый класс `ListDataSource` в качестве источника данных.

Прежде всего удалим связь в `Main.storyboard`. Откройте раскадровку и выберите табличное представление. В панели справа откройте инспектор соединений. Внутри вы увидите соединение с `dataSource`. Щелкните на кнопке с изображением крестика, чтобы удалить это соединение из раскадровки.

Теперь задействуем в табличном представлении новый объект источника данных. Для этого нужно добавить в контроллер связь с табличным представлением. Сделать это можно с помощью **Assistant Editor** (Помощник редактора) в Xcode. Щелкните кнопку **Assistant Editor** (Помощник редактора) вверху справа в окне проекта. В результате откроется исходный код `CatalogViewController`.

Теперь, когда контроллер представления активен в обоих окнах, наведите указатель мыши на табличное представление, нажмите клавишу **Ctrl** и, удерживая ее, нажмите левую кнопку мыши. Перетащите соединение в окно кода, как вы делали это, подключая кнопку перехода к экрану каталога в предыдущей главе, прямо под определение класса `CatalogViewController`, как показано ниже, и отпустите кнопку мыши:

```

class CatalogViewController: UIViewController {
    // Перетащите связь сюда, в эту строку
    override func viewDidLoad() {
        super.viewDidLoad()
        // Выполните здесь дополнительные настройки представления.
    }
}

```

Появится модальный диалог с вопросом, желаете ли вы создать выход, определяющий связь между контроллером и конкретным представлением, или действие для обработки управляющего события (например, нажатия кнопки). Оставьте все как есть и дайте новому выходу имя `tableView`.

Теперь определение `CatalogViewController` должно выглядеть так:

```

class CatalogViewController: UIViewController {
    @IBOutlet weak var tableView: UITableView!
    override func viewDidLoad() {

```

```

    super.viewDidLoad()
    // Выполните здесь дополнительные настройки представления.
  }
}

```

Мы добавили в раскадровку связь между контроллером представления и нашим табличным представлением. Это необходимо для настройки источника данных в табличном представлении внутри класса контроллера. Вот как это делается:

```

class CatalogViewController: UIViewController {
    @IBOutlet weak var tableView: UITableView!
    lazy var dataSource: ListDataSource = {
        return ListDataSource()
    }()
    override func viewDidLoad() {
        super.viewDidLoad()
        tableView.dataSource = dataSource
    }
}

```

Рассмотрим данный код подробнее.

Первое, на что следует обратить внимание, – мы добавили в `CatalogViewController` новое свойство `dataSource` с отложенной инициализацией (`lazy`). Оператор `lazy` откладывает создание экземпляра объекта до первой попытки обратиться к свойству. Это общепринятая практика для источников данных, потому что для их создания и инициализации может потребоваться значительное время. В нашем случае используется легковесный источник данных, но сам прием стоит того, чтобы взять его на вооружение. Кроме того, оператор `lazy` позволяет добавить код инициализации. В данном случае мы просто создаем новый экземпляр `ListDataSource`.

Также мы добавили новый код в `viewDidLoad`. Этот метод является одним из обработчиков событий жизненного цикла представления; когда он вызывается, можно с уверенностью утверждать, что все выходы и действия подключены и все представления загружены. Этот метод вызывается только один раз в течение жизни представления, и это нормально, потому что он нужен лишь один раз.

Внутри `viewDidLoad` мы записываем в свойство `dataSource` табличного представления ссылку на источник данных, созданный внутри контроллера. Если теперь собрать и запустить приложение, вы увидите, что каталог загружается, как и раньше.

У кого-то из вас может возникнуть вопрос: «Зачем все эти сложности с созданием полноценного слоя данных? Разве не проще, когда контроллеры передают данные представлениям напрямую?»

Скажем честно: проще.

Но давайте не будем забывать, что в нашем приложении все еще используются статические данные. Подумайте, что случится, когда мы перейдем к использованию динамических данных и вы поймете, какие возможности открывает отделение данных от слоя представления!

ПРИШЛО ВРЕМЯ ВЕРНУТЬ ОБЪЕКТЫ МОДЕЛИ В РЕАЛЬНОСТЬ

Итак, наш объект `Book` имеет несколько свойств. Но давайте будем честными: большинство книг содержат гораздо больше метаданных, чем те, что мы рассматривали до сих пор. Кроме того, если мы захотим изменить структуру данных, нам придется выпустить новую версию приложения. При этом старая и новая версии приложения должны сохранять совместимость друг с другом. Дополнительные сложности возникнут, когда мы добавим возможность сохранения книг в закладках.

Как сказал бы доктор Фил (да, тот самый доктор Фил¹): «Пришло время вернуть объекты модели в реальность».

К счастью, у нас есть два пути:

- 1) перейти от использования жестко определенной структуры данных к чему-то более гибкому, например JSON;
- 2) перейти от использования статического источника данных к другому, более динамическому источнику данных, например находящемуся на удаленном сервере.

Как упоминалось выше в этой главе, вариант под номером 2 мы рассмотрим в главе 20 данной книги, а здесь исследуем вариант 1: переход на JSON.

JSON для одного, JSON для ВСЕГО

Мы решили использовать JSON. Вот пример, как могли бы выглядеть наши объекты `Book` в формате JSON:

```
{
  "title": "...",
  "authors": ["..."],
  "isbn": "...",
  "pageCount": 0,
  "fiction": true
}
```

Фактически, если взять данные из свойства `sampleData`, которое мы добавили в наш объект `Book`, преобразовать их в формат JSON и сохранить в файл с именем *catalog.json*, мы получим следующий результат:

```
[
  {
    "title": "Fight Club",
    "authors": ["Chuck Palahniuk"],
    "isbn": "978-0393039764",
    "pageCount": 208,
    "fiction": true
  },
  {
    "title": "2001: A Space Odyssey",
```

¹ https://ru.wikipedia.org/wiki/Макпроу_Фил. – Прим. перев.

```
    "authors": ["Arthur C. Clarke"],
    "isbn": "978-0451457998",
    "pageCount": 296,
    "fiction": true
  },
  {
    "title": "Ulysses",
    "authors": ["James Joyce"],
    "isbn": "978-1420953961",
    "pageCount": 682,
    "fiction": true
  },
  {
    "title": "Catch-22",
    "authors": ["Joseph Heller"],
    "isbn": "978-1451626650",
    "pageCount": 544,
    "fiction": true
  },
  {
    "title": "The Stand",
    "authors": ["Stephen King"],
    "isbn": "978-0307947307",
    "pageCount": 1200,
    "fiction": true
  },
  {
    "title": "On The Road",
    "authors": ["Jack Kerouac"],
    "isbn": "978-0143105466",
    "pageCount": 416,
    "fiction": true
  },
  {
    "title": "Heart of Darkness",
    "authors": ["Joseph Conrad"],
    "isbn": "978-1503275928",
    "pageCount": 78,
    "fiction": true
  },
  {
    "title": "A Brief History of Time",
    "authors": ["Stephen Hawking"],
    "isbn": "978-0553380163",
    "pageCount": 212,
    "fiction": false
  },
  {
    "title": "Dispatches",
    "authors": ["Michael Herr"],
    "isbn": "978-0679735250",
    "pageCount": 272,
```

```

    "fiction": false
  },
  {
    "title": "Harry Potter and Prisoner of Azkaban",
    "authors": ["J.K. Rowling"],
    "isbn": "978-0439136365",
    "pageCount": 448,
    "fiction": true
  },
  {
    "title": "Dragons Love Tacos",
    "authors": ["Adam Rubin", "Daniel Salmieri"],
    "isbn": "978-0803736801",
    "pageCount": 40,
    "fiction": true
  }
]

```

Если хотите, можете добавить в этот файл другие книги: пришло время личных фаворитов и тайных увлечений.

ПЕРЕКЛЮЧЕНИЕ СЛОЯ ДАННЫХ НА ИСПОЛЬЗОВАНИЕ JSON

Как вы помните, в нашем слое данных мы до сих пор используем массив `sampleData` из объекта `Book`. Давайте удалим это свойство из обоих проектов и перейдем к использованию файла JSON.

Android

Итак, теперь у нас есть файл `catalog.json` со всеми объектами `Book` в формате JSON. Для Android-приложения этот файл следует сохранить в виде ресурса в папке уровня проекта `/assets/`. Эта папка имеет специальные атрибуты и поддержку, упрощающую операции с ней, но в данный момент она может отсутствовать. Если вы не видите папку с именем `assets` в каталоге проекта, выполните следующие действия: в представлении проекта «Android», в списке файлов слева, щелкните правой кнопкой мыши на имени проекта и выберите пункт **Create** (Создать), затем **Folder** (Папка), потом **Assets Folder** (Папка активов). Далее создайте новый текстовый файл в этом каталоге, скопируйте в него предыдущий код JSON и сохраните файл с именем `catalog.json`.

Давайте переделаем наш класс `BookDataSource` так, чтобы он использовал этот файл вместо статических данных. Для начала прочитаем файл в потоке пользовательского интерфейса, но это только для примера, вообще же, загружая данные с диска или из сети, вы должны выполнять эту операцию в фоновом потоке.

В главе 6 мы показывали, как прочитать файл с диска, а в главе 12 продемонстрировали пару способов преобразования текста в формате JSON в действительные экземпляры объектов Java. Здесь мы используем оба способа.

Реализуем необходимую функциональность в классе источника данных. Со временем вы можете найти более подходящее место для этой логики, а пока

начнем с базового источника данных `List<Book>`, добавив в него код, который будет читать `catalog.json` при создании экземпляра и преобразовывать каждый объект JSON в экземпляр `Book`:

Java

```
public class Books extends ArrayList<Book> implements BookDataSource {

    public Books(Context context) {
        super();
        try {
            readBooksFromDisk(context);
        } catch (Exception e) {
            Log.d("MyApp", "There was a problem reading Books json from disk: " +
                e.getMessage());
        }
    }

    private void readBooksFromDisk(Context context) throws IOException {
        try {
            InputStream inputStream = context.getAssets().open("catalog.json");
            StringBuilder builder = new StringBuilder();
            int b = inputStream.read();
            while (b != -1) {
                builder.append((char) b);
                b = inputStream.read();
            }
            String json = builder.toString();
            List<Book> books = new Gson().fromJson(json,
                new TypeToken<List<Book>>() {}.getType()); addAll(books);
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
        }
    }
}
```

Kotlin

```
class Books(context: Context) : ArrayList<Book>(), BookDataSource {

    init {
        try {
            readBooksFromDisk(context)
        } catch (e: Exception) {
            Log.d("MyApp", "There was a problem reading Books json from disk: " + e.message)
        }
    }

    @Throws(IOException::class)
    private fun readBooksFromDisk(context: Context) {
        try {
            val inputStream? = context.assets.open("catalog.json");
            val builder = StringBuilder()
        }
    }
}
```

```

    var b = inputStream.read()
    while (b != -1) {
        builder.append(b.toChar())
        b = inputStream.read()
    }
    val json = builder.toString()
    val books = Gson().fromJson(json, object : TypeToken<List<Book>>() {}.type)
    addAll(books)
} finally {
    inputStream?.close()
}
}
}
}

```

Если теперь запустить приложение, вы получите массив экземпляров `Book`, число которых равно числу объектов JSON в файле `catalog.json`; однако если вы использовали реализацию на Java, эти экземпляры будут пустыми. Каждое свойство получит значение по умолчанию. Это объясняется использованием венгерской нотации в определениях свойств в классе `Book`:

```

public class Book {
    private final String mTitle;
    private final String[] mAuthors;
    private final String mIsbn;
    private final int mPageCount;
    private final boolean mIsFiction;
    ...
}

```

Когда библиотека `Gson` просматривает класс `Book` и пытается найти свойства, соответствующие свойствам объекта JSON, она обнаружит, что в экземпляре класса `Book` имеется свойство `mTitle`, но нет свойства `title` – с другой стороны, в файле JSON нет ссылок на свойства `mTitle`, `mIsbn` или `mAnything` – в нем используются более понятные человеку имена свойств. К счастью для нас, эта проблема настолько распространена в Android, что для ее устранения придумано простое решение, основанное на использовании аннотации `SerializedName` из библиотеки `Gson`. Добавьте эту аннотацию перед каждым свойством и укажите в ней строку с именем соответствующего свойства в файле JSON, например в коде на Java:

```

public class Book {
    public static final Book[] SAMPLE_DATA = {
        new Book("Fight Club", new String[]{"Chuck Palahniuk"},
            "978-0393039764", 208, true),
        new Book("2001: A Space Odyssey", new String[]{"Arthur C. Clarke"},
            "978-0451457998", 296, true),
        new Book("Ulysses", new String[]{"James Joyce"},
            "978-1420953961", 682, true),
        new Book("Catch-22", new String[]{"Joseph Heller"},
            "978-1451626650", 544, true),
        new Book("The Stand", new String[]{"Stephen King"},
            "978-0307947307", 1200, true),
    }
}

```

```
new Book("On The Road", new String[]{"Jack Kerouac"},
        "978-0143105466", 416, true),
new Book("Heart of Darkness", new String[]{"Joseph Conrad"},
        "978-1503275928", 78, true),
new Book("A Brief History of Time", new String[]{"Stephen Hawking"},
        "978-0553380163", 212, false),
new Book("Dispatches", new String[]{"Michael Herr"},
        "978-0679735250", 272, false),
new Book("Harry Potter and Prisoner of Azkaban", new String[]{"J.K. Rowling"},
        "978-0439136365", 448, true),
new Book("Dragons Love Tacos", new String[]{"Adam Rubin", "Daniel Salmieri"},
        "978-0803736801", 40, true)
};

@SerializedName("title")
private String mTitle;
@SerializedName("authors")
private String[] mAuthors;
@SerializedName("isbn")
private String mIsbn;
@SerializedName("pageCount")
private int mPageCount;
@SerializedName("fiction")
private boolean mIsFiction;

public Book(String title, String[] authors, String isbn, int pageCount,
            boolean isFiction) {
    mTitle = title;
    mAuthors = authors;
    mIsbn = isbn;
    mPageCount = pageCount;
    mIsFiction = isFiction;
}

public String getTitle() {
    return mTitle;
}

public void setTitle(String title) {
    mTitle = title;
}

public String[] getAuthors() {
    return mAuthors;
}

public void setAuthors(String[] authors) {
    mAuthors = authors;
}

public String getIsbn() {
    return mIsbn;
}

public void setIsbn(String isbn) {
    mIsbn = isbn;
}
```



```

}

public int getPageCount() {
    return mPageCount;
}

public void setPageCount(int pageCount) {
    mPageCount = pageCount;
}

public boolean isFiction() {
    return mIsFiction;
}

public void setFiction(boolean fiction) {
    mIsFiction = fiction;
}
}

```

На этот раз экземпляры `Book` должны быть заполнены соответствующим образом. Отлично! Теперь у нас есть действующий протокол сериализации. Рассмотрим некоторые особенности реализации, которые вы могли упустить из виду. Первое, что вы могли пропустить в предыдущем коде, – следующий трудно читаемый фрагмент:

Java

```
List<Book> books = new Gson().fromJson(json, new TypeToken<List<Book>>() {}.getType());
```

Kotlin

```
val books: List<Book> = Gson().fromJson(json, object:TypeToken<List<Book>>() {}.type)
```

Библиотека `Gson` отлично справляется с преобразованием одного экземпляра `Book` в строку `JSON` и наоборот, но когда в игру вступают коллекции, все становится немного сложнее. Как вариант можно создать класс обобщенного типа:

Java

```
public class Books extends ArrayList<Book> {}
```

Kotlin

```
class Books : ArrayList<Book>()
```

Эту коллекцию можно заполнить с помощью традиционного метода `fromJson` из библиотеки `Gson`:

```
String json = // ... строка с несколькими объектами Book в формате JSON
Books books = new Gson().fromJson(json, Books.class);
```

Также можно использовать класс `TypeToken`, генерирующий экземпляры обобщенного типа динамически. Для этого нужно создать экземпляр нового подкласса `TypeToken` и для него вызвать метод `getType`, например:

```
new TypeToken<GENERICIZED_DATA_TYPE>() {
    // ничего не делает
}.getType();
```

В этом примере `GENERICIZED_DATA_TYPE` представляет произвольное количество типов данных, которые могут быть обобщенными типами. Это может быть простой тип, как, например, `new TokenType<Date>...`, или тип со множеством вложенных слоев, как, например, `new TokenType<List<Map<String, List<Date>>>>...`

Но вернемся к нашему примеру. Этот новый источник данных автоматически будет совместим с адаптером и представлением `RecyclerView`, и вам ничего не придется менять в коде адаптера; главное – не забыть передать параметр `Context` конструктору источника данных, чтобы тот мог найти правильный каталог с файлами:

Java

```
recyclerView.setAdapter(new BrowseBooksAdapter(new Books(context));
```

Kotlin

```
recyclerView.adapter = BrowseBooksAdapter(Books(context))
```

Другой интересный трюк основывается на том факте, что `Books` сам является списком объектов `Book`, поэтому файл можно прочитать не с использованием измененного источника данных из предыдущего кода, а с помощью вспомогательного метода и библиотеки `Gson` для преобразования содержимого непосредственно в источник данных. Предположим, что мы вернулись к исходному классу `Books` без методов:

Java

```
public class Books extends ArrayList<Book> implements BookDataSource {}
```

Kotlin

```
class Books : ArrayList<Book>(), BookDataSource
```

Предположим также, что у нас есть методы, описанные в главе 6. Мы можем реализовать следующий метод `onCreate` в `Activity`:

Java

```
public class BrowseContentActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_browse);
        RecyclerView recyclerView = findViewById(R.id.browse_content_recyclerview);
        try {
            InputStream stream = getAssets().open("catalog.json");
            String json = Files.getStringFromStream();
            Books source = new Gson().fromJson(json, Books.class);
            BrowseBooksAdapter adapter = new BrowseBooksAdapter(source);
            recyclerView.setAdapter(adapter);
            recyclerView.setLayoutManager(new LinearLayoutManager(this));
        } catch (IOException e) {
            Log.d("MyApp", "Oops! Something went wrong trying to read our catalog json");
        }
    }
}
```

Kotlin

```

class BrowseContentActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_browse)
        val recyclerView = findViewById<RecyclerView>(R.id.browse_content_recyclerview)
        try {
            val stream = assets.open("catalog.json")
            val json = Files.getStringFromStream(stream)
            val source = Gson().fromJson<Books>(json, Books::class)
            val adapter = BrowseBooksAdapter(source)
            recyclerView.adapter = adapter
            recyclerView.layoutManager = LinearLayoutManager(this)
        } catch (e: IOException) {
            Log.d("MyApp", "Oops! Something went wrong trying to read our catalog json")
        }
    }
}

```

Поскольку экземпляр `Books` является также списком `ArrayList`, он будет заполняться информацией о книгах из файла `catalog.json` и неявно соответствовать требованиям контракта `get` и `size` источника данных `BookDataSource`.

Теперь нам не нужна статическая структура данных `SAMPLE_DATA` в классе `Book`, и мы можем удалить ее. Одной из удобных возможностей в `Android Studio` является поддержка рефакторинга. Кроме переименования методов или переменных по всему проекту, эта возможность позволяет также выполнить безопасное удаление переменной, когда фактическое удаление выполняется, только если переменная больше нигде не используется. Попробуйте выполнить эту операцию с переменной `Book.SAMPLE_DATA` и убедитесь, что это безопасное удаление.

iOS

Самый простой способ работы с форматом JSON в iOS основан на использовании встроенного протокола `Codable`. Этот протокол является комбинацией протоколов `Encodable` и `Decodable`. Они помогают компилятору Swift анализировать исходный код.

За дополнительной информацией о протоколе `Codable` обращайтесь к главе 12. А сейчас рассмотрим пример, иллюстрирующий простоту обработки данных JSON с помощью протокола `Codable`.

Добавим поддержку `Codable` в структуру `Book`:

```

struct Book: Codable {
    let title: String
    let authors: [String]
    let isbn: String
    let pageCount: Int
    let fiction: Bool
}

```

Вот и все. Обратите также внимание, что мы удалили свойство `sampleData` из определения объекта. Это сделано намеренно, потому что теперь в `ListDataSource` мы будем использовать недавно созданный файл `catalog.json` с каталогом книг вместо `sampleData`.

Вот обновленная версия `ListDataSource`:

```
class ListDataSource: NSObject {
    lazy var data: [Book] = {
        do {
            guard let rawCatalogData =
                try? Data(contentsOf:
                    Bundle.main.bundleURL.appendingPathComponent("catalog.json")) else {
                return []
            }
            return try JSONDecoder().decode([Book].self, from: rawCatalogData)
        } catch {
            print("Catalog.json was not found or is not decodable.")
        }
    }
    return []
}

extension ListDataSource: UITableViewDataSource {
    func tableView(_ tableView: UITableView,
                  numberOfRowsInSection section: Int) -> Int {
        return data.count
    }

    func tableView(_ tableView: UITableView,
                  cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        // Получить ячейку из пула свободных ячеек
        let cell =
            tableView.dequeueReusableCell(withIdentifier: "CatalogTableViewCell",
                                       for: indexPath)

        // Найти книгу, соответствующую заполняемой ячейке
        let book = data[indexPath.row]

        // Заполнить метку с заголовком ячейки названием книги
        cell.textLabel?.text = book.title

        return cell
    }
}
```

Мы добавили новое свойство `data` с отложенной инициализацией (`lazy`) для хранения массива книг. При первом обращении к свойству содержимое файла `catalog.json` будет прочитано в объект `rawCatalogData` с типом `Data` и передано объекту `JSONDecoder`, который преобразует этот объект в тип `[Book]`.

Обратите внимание, что это свойство имеет тип `[Book]`, а не просто `Book`. Причина в том, что файл JSON на самом деле содержит массив объектов. Компилятор Swift правильно интерпретирует это объявление при динамическом преобразовании `catalog.json` в объекты. Если файл не будет найден или компилятору Swift не удастся его декодировать, в результате будет получен пустой массив `[]`.

Теперь дополнительно изменим код поддержки табличного представления в соответствии с протоколом `UITableViewDataSource`. Вот строки, на которые нужно обратить внимание:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
    return data.count  
}
```

Обратите также внимание на `data[indexPath.row]` внутри метода `tableView(_:cellForRowAt:)`.

Соберите и запустите приложение, и вы увидите, что оно работает так же, как раньше, – отображает список книг, доступных в библиотеке, – но теперь книги передаются в табличное представление динамически, через файл JSON, включенный в пакет приложения.

Что мы узнали

В этой короткой главе мы узнали:

- как разделить тесно связанные компоненты добавлением нового слоя данных;
- как создать отдельный объект, предназначенный специально для передачи данных в представление;
- как перейти на использование JSON-файла в качестве источника данных.

В следующей главе мы сделаем еще шаг в этом же направлении и добавим в слой данных дополнительные функции для сохранения результатов поиска и понравившихся книг в закладках. Оставайтесь с нами!

Глава 19

Сохранность данных

Если подвести итоги разработки нашего приложения, достигнутые к настоящему моменту, они получатся довольно приятными. Мы реализовали внушительный список возможностей. Извлекаемые данные хранятся в переносимом формате JSON, и они функционально надежны. Теперь можно двигаться дальше.

ДЕТАЛИЗАЦИЯ ИНФОРМАЦИИ О КНИГАХ

Для начала добавим новый экран, на котором будем показывать дополнительную информацию о выбранной книге. В настоящий момент приложение не позволяет увидеть никакой другой информации о книгах, кроме их названий. А знаете ли вы, что книги имеют массу информации о них самих? Это и название, и имена авторов, и код ISBN... Этот список можно продолжать и продолжать! И для многих данная информация очень важна.

Мы должны дать возможность увидеть эту информацию. В частности, в нашем распоряжении имеются следующие сведения в объекте модели Book:

- title;
- authors;
- isbn;
- pageCount;
- fiction.

Это, конечно, не масса, но достаточно много. К тому же мы собираемся немного расширить этот экран, но не будем забегать вперед и начнем с самого простого.

Android

Если вы прочитали главы 2 и 15, многое, о чем рассказывается в этом разделе, покажется вам знакомым.

Сначала определим макет XML. Как вы уже знаете, нам нужно показать каждое свойство, имеющееся в экземпляре Book, поэтому просто отобразим их в виде вертикального списка и выполним оформление программно. И снова заключим `LinearLayout` в `ScrollView`, чтобы информацию можно было отобразить на любом экране, с любыми размерами и любым разрешением, а также с любыми настройками доступности, как, например, выбор крупного шрифта для отображения текста.

```

<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/white">
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textview_title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/textview_authors"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/textview_isbn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/textview_pagecount"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/textview_isfiction"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
</ScrollView>

```

Сохраните этот XML-код в файле *res/layout/activity_detail.xml*.

Нам нужно, чтобы в этих текстовых представлениях отображались полные метки, поэтому используем строки-заполнители, добавив их в файл *strings.xml*. Строка-заполнитель – это строка, содержащая специальные символы форматирования, которые должны замещаться значениями переменных. Более подробную информацию о строках-заполнителях ищите в документации Java с описанием метода `String.format`.

Добавьте в *strings.xml* следующие строки:

```

<string name="detail_title">Book Title: %s</string>
<string name="detail_authors">Book Authors: %s</string>
<string name="detail_isbn">Book ISBN: %s</string>
<string name="detail_pagecount">Book Page Count: %d</string>
<string name="detail_isfiction">Book : %b</string>

```

Нам также необходим контроллер пользовательского интерфейса для отображения макета, а также для управления поведением компонентов. В данном случае эту роль будет играть *Activity*.

Мы должны передать этому контроллеру некоторую информацию о книге, но сделать это в *Android* не так-то просто. Экземпляр *Activity* создается программно и непрозрачно, но, как рассказывалось в главе 1, простые данные можно передать в экземпляре *Bundle* с помощью намерения *Intent*, запускающего контроллер.

В приложениях, подобных этому, можно встретить два разных подхода. Иногда один из них подходит лучше, иногда другой, но в большинстве случаев выбор от личных предпочтений разработчиков. И да, не стесняйтесь экспериментировать с другими подходами – наша команда, например, использует для решения этой задачи совершенно другой и очень нестандартный подход, но его обсуждение выходит за рамки данной главы.

Итак, подход № 1: сериализовать весь объект и передать его как строку *String* (или массив байтов *byte[]*), а затем десериализовать в контроллере. Все это реализуется очень просто, но помните, что емкость экземпляра *Bundle* ограничена – она составляет 1 Мбайт – и распределяется между всеми операциями, даже теми, о существовании которых вы можете и не подозревать.

Подход № 2: передать какой-то уникальный идентификатор, например идентификационный номер или *URI*, а затем по этому идентификатору извлечь информацию из другого источника, например из локальной базы данных, хранилища *JSON* или даже удаленного сервера.

Пока для простоты мы используем первый подход. Этот экземпляр *Activity* будет принимать из своего объекта *Intent* строку в формате *JSON*, содержащую экземпляр *Book* в сериализованном виде. Присвоим этой строке идентификатор *BOOK_JSON* и сохраним его в константе. Затем десериализуем строку в методе *onCreate* и сохраним полученные значения в соответствующих свойствах.

Java

```
public class BookDetailActivity extends Activity {
    public static final String BOOK_JSON = "BOOK_JSON";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_detail);

        // получить экземпляр Book из сериализованной строки
        String json = getIntent().getStringExtra(BOOK_JSON);
        Book book = new Gson().fromJson(json, Book.class);

        // заполнить поля представления
        Resources resources = getResources();

        String title = resources.getString(R.string.detail_title, book.getTitle());
        ((TextView) findViewById(R.id.textview_title)).setText(title);

        String authors = TextUtils.join(", ", book.getAuthors());
```



```

    authors = resources.getString(R.string.detail_authors, authors);
    ((TextView) findViewById(R.id.textview_authors)).setText(authors);

    String isbn = resources.getString(R.string.detail_isbn, book.getIsbn());
    ((TextView) findViewById(R.id.textview_isbn)).setText(isbn);

    String pageCount = resources.getString(R.string.detail_pagecount,
    book.getPageCount());
    ((TextView) findViewById(R.id.textview_pagecount)).setText(pageCount);

    String fiction = resources.getString(R.string.detail_isfiction, book.isFiction());
    ((TextView) findViewById(R.id.textview_isfiction)).setText(fiction);
}
}

```

Kotlin

```

class BookDetailActivity : Activity() {
    companion object {
        val BOOK_JSON = "BOOK_JSON"
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_detail)
        // получить экземпляр Book из сериализованной строки
        val json = intent.getStringExtra(BOOK_JSON)
        val book = Gson().fromJson(json, Book::class.java)
        // заполнить поля представления
        textview_title.text = resources.getString(R.string.detail_title, book.title)
        textview_authors.text = resources.getString(R.string.detail_authors,
            TextUtils.join(", ", book.authors))
        textview_isbn.text = resources.getString(R.string.detail_isbn, book.isbn)
        textview_pagecount.text = resources.getString(R.string.detail_pagecount,
            book.pageCount)
        textview_isfiction.text = resources.getString(R.string.detail_isfiction,
            book.isFiction)
    }
}

```

Не забудьте также зарегистрировать новый объект Activity в манифесте приложения:

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

Отлично! Мы создали Activity с пользовательским интерфейсом, отображающим всю информацию, имеющуюся в экземпляре Book. Теперь вернемся к представлению списка, созданному в главе 17, и реализуем обработку события касания, по которому будем передавать выбранный экземпляр Book из BrowseContentActivity в новый BookDetailActivity. Прежде всего мы должны добавить в элемент списка (на данный момент это экземпляр TextView) метод View.OnClickListener, который извлекает соответствующий экземпляр Book и запускает BookDetailActivity. Это можно сделать только один раз, на этапе создания – в методе onCreateViewHolder. Поскольку эти представления постоянно

освобождаются и используются повторно, нужно своевременно обновлять связь между элементом списка и соответствующей книгой, что можно сделать в цикле связывания и обновления, представленного методом `onBindViewHolder`.

Вот как должна выглядеть обновленная версия `BrowseBooksAdapter`:

Java

```
public class
BrowseBooksAdapter extends RecyclerView.Adapter<BrowseBooksAdapter.ViewHolder> {
    private final BookDataSource mSource;

    public BrowseBookAdapter(BookDataSource source) {
        super();
        mSource = source;
    }

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        return new ViewHolder(new TextView(parent.getContext()));
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        Book book = mSource.get(position);
        holder.mTextView.setTag(book);
        holder.mTextView.setText(book.getTitle());
    }

    @Override
    public int getItemCount() {
        return mSource.size();
    }

    public static class ViewHolder extends RecyclerView.ViewHolder {
        private TextView mTextView;
        public ViewHolder(@NonNull View itemView) {
            super(itemView);
            mTextView = (TextView) itemView;
            mTextView.setOnClickListener(this::showBook);
        }

        private void showBook(View view) {
            Book book = (Book) view.getTag();
            String json = new Gson().toJson(book);
            Intent intent = new Intent(view.context, BookDetailActivity.class);
            intent.putExtra(BookDetailActivity.BOOK_JSON, json);
            view.getContext().startActivity(intent);
        }
    }
}
```

Kotlin

```
class BrowseBooksAdapter(private val source: BookDataSource) :
    RecyclerView.Adapter<BrowseBooksAdapter.ViewHolder>() {
```

```

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    return ViewHolder(Textview(parent.context))
}

override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val book = source.get(position)
    holder.textview.tag = book
    holder.textview.text = book.title
}

override fun getItemCount(): Int {
    return source.size
}

class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    val textview: TextView = itemView as TextView

    init {
        textview.setOnClickListener { v -> showBook(v) }
    }

    private fun showBook(view: View) {
        val book = view.tag as Book
        val json = Gson().toJson(book)
        val intent = Intent(view.context, BookDetailActivity::class.java)
        intent.putExtra(BookDetailActivity.BOOK_JSON, json)
        view.context.startActivity(intent)
    }
}
}

```

Если теперь запустить приложение, то при каждом касании элемента списка в каталоге должен открываться экран с подробной информацией о книге, имеющейся у нас. Поздравляем, вы только что освоили один из шаблонов, широко используемых в Android и вообще в программировании пользовательского интерфейса! Улучите момент и погладите себя по голове!

iOS

Для начала откройте *Main.storyboard*. Перетащите новый объект **View Controller** на холст из панели **Library** (Библиотека), как мы делали это, добавляя в приложение другие экраны. В настоящее время этот экран изолирован от остальной части приложения, поэтому подключим его через переход, чтобы его можно было открыть позже.

В этом редактор раскадровки бесподобен! Если щелкнуть на ячейке табличного представления в сцене каталога – той, что мы создали в главе 17, – в структуре документа в редакторе раскадровки, вы сможете перетащить ее, удерживая нажатой клавишу **Ctrl**, в нужную точку и подключить переход *непосредственно* к самой ячейке.

В появившемся диалоге выберите **Show** (Показать) в разделе **Action Segues** (Действие перехода). Если теперь собрать и запустить приложение, вы увидите, что касание книги в каталоге помещает в стек представлений новый конт-

роллер – пустой в настоящее время. Однако уже сейчас наблюдаются некоторые странности в отображении интерфейса, которые нужно исправить. Если вернуться в представление каталога, можно заметить, что ячейка табличного представления продолжает иметь вид выбранной. Кроме того, наше новое представление вообще ничего не содержит – это просто белый экран. Исправим это!

Добавим деталей для вывода деталей

Для исправления обоих недостатков нужно создать свой контроллер представления для новой сцены. Добавьте в проект новый файл с определением класса `DetailViewController`. Он должен наследовать `UIViewController`, как показано ниже:

```
import UIKit

class DetailViewController: UIViewController {
}
```

Предыдущий опыт создания списка, а также опыт вывода детальной информации в Android подсказывают, что мы должны добавить несколько представлений в наш контроллер. Все они будут экземплярами меток, поэтому продолжим и добавим несколько выходов; мы подключим их позже в редакторе раскладки.

После добавления выходов контроллер представления должен выглядеть так:

```
import UIKit

class DetailViewController: UIViewController {
    @IBOutlet var titleLabel: UILabel!
    @IBOutlet var authorsLabel: UILabel!
    @IBOutlet var isbnLabel: UILabel!
    @IBOutlet var pageCountLabel: UILabel!
    @IBOutlet var fictionLabel: UILabel!
}
```

Но как мы будем заполнять эти свойства?

Мы знаем, что исходная информация поступит в форме объекта `Book`, поэтому создадим метод `populate(from:)`, который принимает аргумент `Book` и заполняет метки значениями его свойств. Этот объект будет передаваться контроллером представления каталога во время перехода. Вот как должна выглядеть окончательная версия класса с этим методом:

```
import UIKit

class DetailViewController: UIViewController {
    @IBOutlet var titleLabel: UILabel!
    @IBOutlet var authorsLabel: UILabel!
    @IBOutlet var isbnLabel: UILabel!
    @IBOutlet var pageCountLabel: UILabel!
    @IBOutlet var fictionLabel: UILabel!

    func populate(from book: Book) {
```

```

        titleLabel.text = book.title
        // перечислить авторов в строке через запятую
        authorsLabel.text = book.authors.join(separator: ", ")
        isbnLabel.text = book.isbn
        pageCountLabel.text = book.pageCount.description
        // По логическому флагу определить категорию книги
        fictionLabel.text = book.fiction ? "Fiction" : "Nonfiction"
    }
}

```

Теперь вернемся к `CatalogViewController` и подключим это соединение. Как вы помните, наш переход запускается автоматически, потому что мы подключили его к самой ячейке табличного представления. Подготовка к переходу происходит в контроллере представления, вызвавшем переход. Существует даже специальный метод, который можно переопределить, чтобы добавить свои подготовительные операции: `prepare(for:sender:)`. Он принимает в первом параметре объект `UIStoryboardSegue`, содержащий целевой контроллер представления, в данном случае – наш `DetailViewController`.

Но, прежде чем использовать этот метод, нужно получить экземпляр книги, соответствующий ячейке. Для этого добавим в `ListDataSource` новый метод `book(for:)`, который будет получать индекс из табличного представления и возвращать книгу с этим индексом из источника данных:

```

func book(for indexPath: IndexPath) -> Book {
    return data[indexPath.row]
}

```

Теперь переопределим метод `prepare(for:sender:)` в `CatalogViewController`, в котором заполним целевое представление перед переходом:

```

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let detailViewController = segue.destination as? DetailViewController,
        let indexPath = tableView.indexPathForSelectedRow {
        detailViewController.populate(from: dataSource.book(for: indexPath))
    }
}

```

Метод получился коротким, но для вас очень важно понять, что в нем происходит. Сначала проверяется тип целевого контроллера, это должен быть контроллер `DetailViewController`. Затем проверяется, имеется ли в табличном представлении выбранный элемент. Обе эти проверки важны, потому что данный метод вызывается для каждого перехода, запускаемого этим контроллером представления. Если оба условия выполняются, для экземпляра `DetailViewController` вызывается метод, который мы создали ранее, чтобы получить экземпляр книги.

Теперь, прежде чем оставить этот файл, вернемся в редактор раскладки и добавим еще одно исправление: сбросим выделение элемента в табличном представлении. Это необходимо, потому что в качестве базового класса мы использовали стандартный `UIViewController` вместо `UITableViewController`. Нам нужно, чтобы выделение сбрасывалось при каждом появлении представления, что обеспечит красивую анимацию затухания и даст нашим пользовате-

лям возможность заметить, какой элемент они выбрали. Используем для этого метод `viewDidAppear(_:)`, являющийся частью жизненного цикла контроллера представления.

Вот как должна выглядеть окончательная версия нашего класса `CatalogViewController`:

```
import UIKit

class CatalogViewController: UIViewController {

    @IBOutlet weak var tableView: UITableView!
    lazy var dataSource: ListDataSource = {
        return ListDataSource()
    }()

    override func viewDidLoad() {
        super.viewDidLoad()
        tableView.dataSource = dataSource
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)
        if let indexPath = tableView.indexPathForSelectedRow {
            tableView.deselectRow(at: indexPath, animated: true)
        }
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if let detailViewController = segue.destination as? DetailViewController,
            let indexPath = tableView.indexPathForSelectedRow {
            detailViewController.populate(from: dataSource.book(for: indexPath))
        }
    }
}
```

И последнее, что нужно сделать с новым представлением, – добавить в сцену метки для новых выходов в `DetailViewController`. Вы можете расположить их, как вам заблагорассудится. Проявите творческий подход! При желании можете заглянуть в пример проекта, доступный в нашей репозитории [GitHub](#). В нем мы использовали представления стека (из панели **Library** (Библиотека)), чтобы получить возможность автоматически сжимать и распаивать их в зависимости от размера экрана. Однако вы можете пойти по более простому пути: перетащить в сцену метки и добавить некоторые ограничения `Auto Layout`. Вы должны получить нечто, похожее на рис. 19.1.

Теперь переключим эту сцену на использование нашего класса `DetailViewController`. Для этого откройте инспектор идентичности справа в окне редактора и замените `UIViewController` на `DetailViewController`. Далее, удерживая клавишу **Ctrl**, перетащите объект контроллера представления из схемы документа на каждую метку и для каждой выберите соответствующий выход из числа тех, что мы определили ранее.

Теперь соберите и запустите проект, перейдите в каталог и коснитесь любого элемента. В ответ должен открыться новый экран с подробной информацией из этого элемента. Здорово!

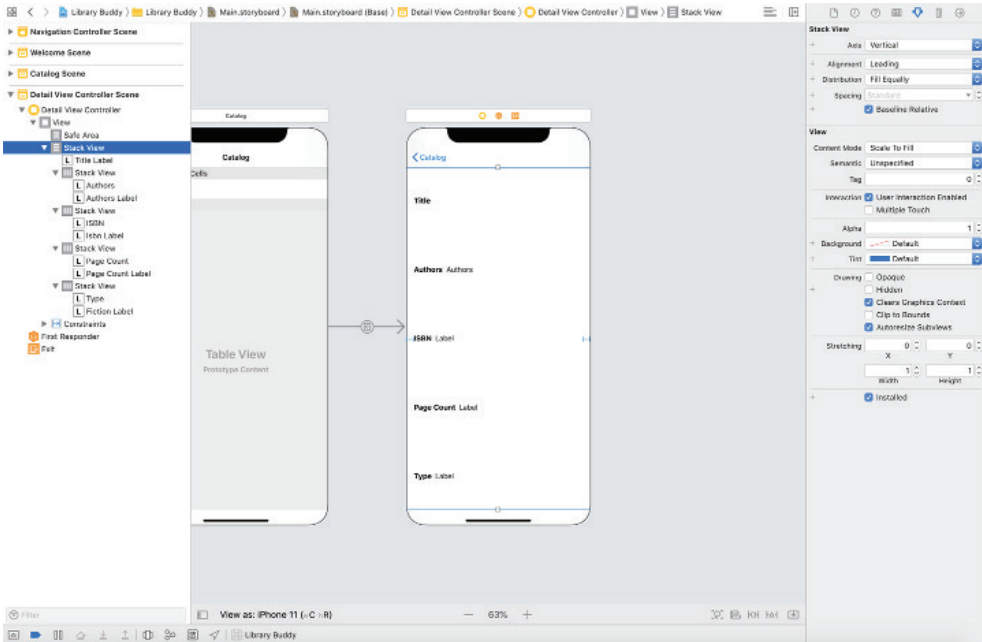


Рис. 19.1 ❖ Вы можете заметить, что представления расположились довольно далеко друг от друга (это нормально!)

СОХРАНЕНИЕ КНИГ ДЛЯ ПОСЛЕДУЮЩЕГО ИСПОЛЬЗОВАНИЯ

Теперь давайте подумаем, что можно сделать, если пользователь, прочитав подробную информацию о книгах, сочтет ее интересной и решит сохранить в закладках? Можно, конечно, сохранить книгу в памяти приложения, но эта информация исчезнет, как только пользователь закроет приложение. Можно сохранить книгу в файловой системе, в формате JSON, но этот способ быстро станет слишком громоздким, если пользователь попытается сохранить в закладках слишком большое число книг, и к тому же он не очень эффективен. Набор данных, который мы используем в этом примере, невелик, но библиотека, для которой мы создаем приложение, имеет ОГРОМНОЕ количество книг.

Другой важный аспект: манипулирование этой информацией, например сортировка, фильтрация и разбиение на страницы, легко реализовать, если использовать один из популярных механизмов хранения данных, такой как Core Data, база данных SQL, Realm или Room и др. Этого нельзя сказать о файлах JSON. Кроме того, размер начинает превращаться в большую проблему, когда каждый файл должен поддерживать структуру, порядок и имена ключей для хранимых в нем объектов. Для небольших файлов или файлов, хранящих единичные объекты, применение формата JSON не только уместно, но и предпочтительно, однако для библиотеки книг, которую нужно сопровождать, необходимо нечто более надежное: база данных.

Пока мы размышляем о выборе лучшего подхода к хранению данных (в этой главе речь идет о создании собственного хранилища), не будем останавливать-

ся и добавим кнопку, коснувшись которой, пользователь сможет поместить книгу в закладки.

Android

Поскольку мы не используем ActionBar в нашем приложении, мы немного отклонимся от обычного пути и поместим кнопку в тот же пользовательский интерфейс, где находятся информационные представления TextView. Просто добавьте XML-элемент Button в конец LinearLayout и присвойте ей идентификатор, `button_save`. Если вы захотите отделить ее визуально, добавьте атрибут `android:layout_gravity="center"`, например так:

```
<Button
    android:id="@+id/button_save"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Mark as Favorite" />
```

Теперь макет представления с детальной информацией должен выглядеть так:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">
        <TextView
            android:id="@+id/textview_title"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
        <TextView
            android:id="@+id/textview_authors"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
        <TextView
            android:id="@+id/textview_isbn"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
        <TextView
            android:id="@+id/textview_pagecount"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
        <TextView
            android:id="@+id/textview_isfiction"
```



```

        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

<Button
    android:id="@+id/button_save"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Mark as Favorite" />

</LinearLayout>

</ScrollView>

```

Чуть позже, после реализации слоя хранения данных, мы добавим обработчик касания кнопки, который сохранит книгу в закладках. Оставайтесь с нами!

Это оказалось очень просто. Теперь сделаем то же самое в iOS, правда, это будет чуть сложнее.

iOS

Добавим нашу кнопку на панель навигации. Это довольно распространенное место для таких действий. Откройте *Main.storyboard* и зайдите в сцену с представлением детальной информации. Найдите **Navigation Item** в панели **Library** (Библиотека) и перетащите его на сцену. В результате будет создан элемент навигации, связанный с панелью навигации в контроллере представления, которую мы сможем править в редакторе раскадровки. Выберите элемент навигации с меткой **Title** в структуре документа. Откройте инспектор атрибутов и очистите поле **Title**. После этого название объекта в структуре документа изменится на **Navigation Item**.

i Если выбрать тип перехода *Push*, он автоматически будет добавлен средой Xcode, потому что *Push* является типом перехода, предназначенным для контроллеров навигации. Однако этот тип устарел и требует вручную добавить дополнительную контекстную информацию в файл раскадровки. Конечно, все необходимое можно сделать в коде, но подход на основе раскадровки проще. Как минимум, он помогает сохранить настройки пользовательского интерфейса в одном месте, не разбрасывая их по файлам с исходным кодом.

Теперь в панели **Library** (Библиотека) выполните поиск по строке *Bar Button Item* и перетащите найденный элемент в правую часть панели навигации в сцене. В инспекторе атрибутов для этого элемента введите в поле **Title** строку *Save Book*. После этого сцена должна выглядеть, как показано на рис. 19.2.

Создадим обработчик для этой новой кнопки.

Откройте **Assistant Editor** (Помощник редактора) в Xcode, отображающий код и интерфейс рядом. Удерживая нажатой клавишу **Ctrl**, перетащите новую кнопку в исходный код внутри контроллера представления под объявлением метода *populate(for:)*, в самом низу класса. В результате внутри *DetailViewController* появится новый метод:

```

@IBAction func saveBook(_ sender: Any) {
}

```

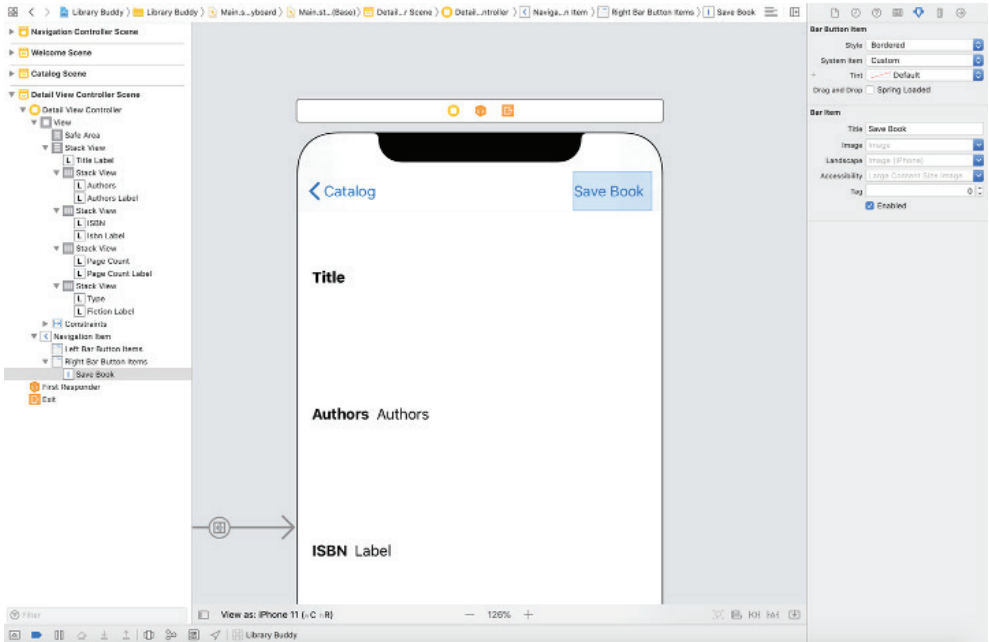


Рис. 19.2 ❖ Кнопка с текстом **Save Book** (Сохранить книгу) в панели навигации

Если теперь собрать и запустить проект, вы увидите кнопку в правом верхнем углу экрана с подробными сведениями о книге, но пока она ничего не делает. Исправим этот недостаток.

ЗАПИСЬ КНИГ В ХРАНИЛИЩЕ

Как упоминалось выше, мы должны реализовать возможность сохранения книг для последующего использования. Нам нужно производительное хранилище, учитывая размер библиотеки, которое обеспечит сохранность данных между запусками приложения и позволит сохранять и извлекать данные. Проще говоря, нам нужен слой хранения данных.

Вдохните поглубже!

Создавая кросс-платформенный набор приложений, подобных показанным в этой книге, вы обнаружите, что в области хранения данных системы Android и iOS предлагают совершенно разные наборы инструментов. Мы могли бы использовать кросс-платформенное решение, например Realm, но в интересах сохранения верности каждой платформе мы используем инструменты, наиболее типичные для каждой платформы.

Однако прежде чем мы начнем говорить об Android и iOS по отдельности, определим общие цели нашего слоя хранения данных. У нас есть вполне конкретные требования:

- когда пользователь коснется кнопки **Save Book** (Сохранить книгу), приложение должно записать идентификатор книги в некоторое хранилище;

- по желанию пользователя приложение должно прочитать список идентификаторов из хранилища и отобразить список соответствующих книг;
- на экране с подробной информацией должен присутствовать некоторый признак, подсказывающий, что книга уже имеется в закладках;
- пользователь должен иметь возможность удалить выбранную им книгу из закладок.

Первые два пункта в этом списке будут реализованы в Android и iOS совершенно по-разному. Однако, поскольку мы используем архитектуру MVC, основные отличия будут сосредоточены на уровне Model. Уровни View и Controller (то есть последние два элемента в этом списке требований) будут иметь архитектурно схожую реализацию.

А теперь приступим к созданию слоя хранения данных.

Android

Как уже не раз упоминалось, в Android есть несколько способов хранения данных, и на момент написания этой книги компания Google предлагала использовать библиотеку Room. Однако мы используем SQLite по нескольким причинам, наиболее важной из которых являются зрелость и распространенность SQL, фактически ставшего золотым стандартом. SQLite дает возможность хранить данные только на локальном устройстве, но поддерживает тот же базовый синтаксис, правила и операции, что и все основные базы данных SQL, такие как PostgreSQL, MySQL и MSSql. Конечно, вы можете поэкспериментировать с другими подходами, такими как Room. Также популярностью пользуется библиотека Realm, которая к тому же предлагает преимущество кросс-платформенности.

Независимо от того, какое решение примете вы, в этой книге мы посмотрим, как создать слой хранения данных на основе SQLite.

Прежде всего так же, как при использовании любой другой системы управления реляционными базами данных, нам нужна хотя бы одна таблица. В нашем примере мы используем три таблицы: одну для экземпляров Book, одну для хранения имен авторов (в виде строки) и одну для связки первых двух. Таблицы, предназначенные для связки, иногда называют сквозными таблицами, таблицами мостов, сводными таблицами или таблицами соединений. Это очень распространенная парадигма, поэтому мы посчитали, что ее применение послужит отличным и простым примером использования SQL.

Далее мы используем стандартный синтаксис SQL «создания таблиц». Более подробную информацию о каждой инструкции и ключевом слове вы найдете в документации разработчика, однако мы постараемся дать вам достаточно информации, чтобы вы могли приступить к работе.

```
CREATE TABLE IF NOT EXISTS BOOKS (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    TITLE TEXT,
    ISBN TEXT,
    PAGECOUNT INTEGER,
    IS_FICTION INTEGER);
```

```
CREATE TABLE IF NOT EXISTS AUTHORS (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    NAME TEXT);

CREATE TABLE IF NOT EXISTS BOOK_AUTHORS (
    BOOK_ID INTEGER REFERENCES BOOKS(ID),
    AUTHOR_ID INTEGER REFERENCES AUTHORS(ID))
```

Нам нужно, чтобы при первом запуске приложение создавало эти три таблицы, чтобы сразу же иметь к ним доступ. Рассказывая о `SQLiteOpenHelper` в главе 7, мы говорили, что эту операцию можно выполнить в переопределенном методе `onCreate`. Он вызывается только один раз – при первом запуске приложения после установки. Обратите внимание, что он не будет вызван, пока вы в первый раз не используете подкласс для получения экземпляра базы данных с помощью `getReadableDatabase` или `getWritableDatabase`. Мы сами всегда предпочитаем последний из них, потому что получаем возможность обновить хранилище данных.

Вот как это может выглядеть:

Java

```
public class DBHelper extends SQLiteOpenHelper {

    public DBHelper(Context context, String name,
        SQLiteDatabase.CursorFactory factory,
        int version) {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase database) {
        database.execSQL(BOOKS_CREATE_TABLE);
        database.execSQL(AUTHORS_CREATE_TABLE);
        database.execSQL(BRIDGE_CREATE_TABLE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase database, int i, int i1) {
        // ничего не делает
    }

    public static final String BOOKS_TABLE_NAME = "BOOKS";
    public static final String BOOKS_COLUMN_TITLE = "TITLE";
    public static final String BOOKS_COLUMN_ISBN = "ISBN";
    public static final String BOOKS_COLUMN_PAGECOUNT = "PAGECOUNT";
    public static final String BOOKS_COLUMN_ISFICTION = "ISFICTION";

    public static final String AUTHORS_TABLE_NAME = "AUTHORS";
    public static final String AUTHORS_COLUMN_ID = "ID";
    public static final String AUTHORS_COLUMN_NAME = "NAME";

    public static final String BRIDGE_TABLE_NAME = "BOOK_AUTHOR_BRIDGE";
    public static final String BRIDGE_COLUMN_BOOK_ID = "BOOK_ID";
    public static final String BRIDGE_COLUMN_AUTHOR_ID = "AUTHOR_ID";

    public static final String BOOKS_CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " +
```

```

BOOKS_TABLE_NAME +
    " (" +
        BOOKS_COLUMN_TITLE + " TEXT," +
        BOOKS_COLUMN_ISBN + " TEXT," +
        BOOKS_COLUMN_PAGECOUNT + " INTEGER," +
        BOOKS_COLUMN_ISFICTION + " INTEGER);"

public static final String AUTHORS_CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " +
AUTHORS_TABLE_NAME +
    " (" +
        AUTHORS_COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
        AUTHORS_COLUMN_NAME + " TEXT)";

public static final String BRIDGE_CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " +
BRIDGE_TABLE_NAME +
    " (" +
        BRIDGE_COLUMN_BOOK_ID + " INTEGER REFERENCES BOOKS," +
        BRIDGE_COLUMN_AUTHOR_ID + " INTEGER REFERENCES AUTHORS)";
}

```

Kotlin

```

class DbHelper(context: Context, name: String, factory: SQLiteDatabase.CursorFactory,
    version: Int) :
    SQLiteOpenHelper(context, name, factory, version) {
    override fun onCreate(database: SQLiteDatabase) {
        database.execSQL(BOOKS_CREATE_TABLE)
        database.execSQL(AUTHORS_CREATE_TABLE)
        database.execSQL(BRIDGE_CREATE_TABLE)
    }

    override fun onUpgrade(database: SQLiteDatabase, i: Int, i1: Int) {
        // ничего не делает
    }

    companion object {
        const val BOOKS_TABLE_NAME = "BOOKS"
        const val BOOKS_COLUMN_TITLE = "TITLE"
        const val BOOKS_COLUMN_ISBN = "ISBN"
        const val BOOKS_COLUMN_PAGECOUNT = "PAGECOUNT"
        const val BOOKS_COLUMN_ISFICTION = "ISFICTION"

        const val AUTHORS_TABLE_NAME = "AUTHORS"
        const val AUTHORS_COLUMN_ID = "ID"
        const val AUTHORS_COLUMN_NAME = "NAME"

        const val BRIDGE_TABLE_NAME = "BOOK_AUTHOR_BRIDGE"
        const val BRIDGE_COLUMN_BOOK_ID = "BOOK_ID"
        const val BRIDGE_COLUMN_AUTHOR_ID = "AUTHOR_ID"

        const val BOOKS_CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " + BOOKS_TABLE_NAME +
            " (" +
                BOOKS_COLUMN_TITLE + " TEXT," +
                BOOKS_COLUMN_ISBN + " TEXT," +
                BOOKS_COLUMN_PAGECOUNT + " INTEGER," +

```

```

    BOOKS_COLUMN_ISFICTION + " INTEGER);"

const val AUTHORS_CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " + AUTHORS_TABLE_NAME +
" (" + AUTHORS_COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
  AUTHORS_COLUMN_NAME + " TEXT)"

const val BRIDGE_CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " + BRIDGE_TABLE_NAME +
" (" +
  BRIDGE_COLUMN_BOOK_ID + " INTEGER REFERENCES BOOKS," +
  BRIDGE_COLUMN_AUTHOR_ID + " INTEGER REFERENCES AUTHORS)"
}
}

```

Итак, у нас есть новенькая база данных, еще пахнущая свежей краской, но она пустая – все наши данные все еще находятся в *catalog.json*. Исправим это. Прежде всего нам понадобятся методы чтения и записи экземпляров Book в базу данных, поэтому добавим их:

Java

```

public class BookTransactions {

    public static Book read(SQLiteDatabase database, String isbn) {
        Cursor cursor = database.query(
            DBHelper.BOOKS_TABLE_NAME, // Строка, имя запрашиваемой таблицы
            null, // Строка, массив столбцов
            DBHelper.BOOKS_COLUMN_ISBN + " = ?", // Строка, предложение WHERE
            new String[] { isbn }, // Строка, массив значений для предложения WHERE
            null, // GROUP BY
            null, // HAVING
            null, // ORDER BY
            "1" // Строка, предложение LIMIT
        );

        if (cursor.moveToNext()) {
            val book = Book(
                cursor.getString(0),
                mutableListOf(),
                isbn,
                cursor.getInt(2),
                cursor.getInt(3) == 1
            )
            readAuthors(database, book);
            return Book()
        }
        cursor.close();
        return Book()
    }

    public static void write(SQLiteDatabase database, Book book) {
        ContentValues contentValues = new ContentValues();
        contentValues.put(DBHelper.BOOKS_COLUMN_ISBN, book.getIsbn());
        contentValues.put(DBHelper.BOOKS_COLUMN_TITLE, book.getTitle());
        contentValues.put(DBHelper.BOOKS_COLUMN_PAGECOUNT, book.getPageCount());
        contentValues.put(DBHelper.BOOKS_COLUMN_ISFICTION, book.isFiction());
    }
}

```

```

database.beginTransaction();
try {
    insertOrUpdate(
        database,
        DbHelper.BOOKS_TABLE_NAME,
        contentValues,
        DbHelper.BOOKS_COLUMN_ISBN + " = ? ",
        new String[] { book.getIsbn() });
    for (String author : book.getAuthors()) {
        contentValues = new ContentValues();
        contentValues.put(DbHelper.AUTHORS_COLUMN_NAME, author);
        database.insertWithOnConflict(DbHelper.AUTHORS_TABLE_NAME, null,
            contentValues, SQLiteDatabase.CONFLICT_IGNORE);
    }
    database.setTransactionSuccessful();
    Log.d("MyTag", "wrote data to database");
} catch (Exception e) {
    Log.d("MyTag", "there was a problem inserting data: " + e.getMessage());
} finally {
    Log.d("MyTag", "finally");
    database.endTransaction();
}
}

public static boolean insertOrUpdate(SQLiteDatabase database, String table,
    ContentValues values,
    String where, String[] args) {
    long rowId = database.insertWithOnConflict(table, null, values,
        SQLiteDatabase.CONFLICT_IGNORE);

    if (rowId < 0) {
        database.update(table, values, where, args);
        return false;
    }
    return true;
}

public static void readAuthors(SQLiteDatabase database, Book book) {
    Cursor cursor = database.query(
        DbHelper.AUTHORS_TABLE_NAME
        + " join "
        + DbHelper.BRIDGE_TABLE_NAME
        + " on "
        + DbHelper.AUTHORS_TABLE_NAME + "." + DbHelper.AUTHORS_COLUMN_ID
        + " = "
        // Строка, имя запрашиваемой таблицы
        + DbHelper.BRIDGE_TABLE_NAME + "." + DbHelper.BRIDGE_COLUMN_AUTHOR_ID,
        new String[] { DbHelper.AUTHORS_COLUMN_NAME }, // Строка, массив столбцов
        DbHelper.BRIDGE_COLUMN_AUTHOR_ID + " = ?", // Строка, предложение WHERE
        new String[] { book.getIsbn() }, // Строка, массив значений для WHERE
        null, // GROUP BY
        null, // HAVING
        null // ORDER BY
    );
}

```

```

String[] authors = new String[cursor.getCount()];
int i = 0;
while (!cursor.isAfterLast()) {
    cursor.moveToNext();
    authors[i++] = cursor.getString(0);
}
book.setAuthors(authors);
}
}

```

Kotlin

```

object BookTransactions {
    fun read(database: SQLiteDatabase, isbn: String): Book {
        val cursor = database.query(
            DbHelper.BOOKS_TABLE_NAME,
            null,
            DbHelper.BOOKS_COLUMN_ISBN + " = ?",
            arrayOf(isbn), null, null, null, "1"
        )
        val book = Book()
        book.isbn = isbn
        if (cursor.moveToNext()) {
            book.title = cursor.getString(0)
            book.pageCount = cursor.getInt(2)
            book.isFiction = cursor.getInt(3) == 1
            readAuthors(database, book)
        }
        cursor.close()
        return book
    }

    fun write(database: SQLiteDatabase, book: Book) {
        var contentValues = ContentValues()
        contentValues.put(DbHelper.BOOKS_COLUMN_ISBN, book.isbn)
        contentValues.put(DbHelper.BOOKS_COLUMN_TITLE, book.title)
        contentValues.put(DbHelper.BOOKS_COLUMN_PAGECOUNT, book.pageCount)
        contentValues.put(DbHelper.BOOKS_COLUMN_ISFICTION, book.isFiction)
        database.beginTransaction()
        try {
            insertOrUpdate(
                database,
                DbHelper.BOOKS_TABLE_NAME,
                contentValues,
                DbHelper.BOOKS_COLUMN_ISBN + " = ? ",
                arrayOf(book.isbn)
            )
            for (author in book.authors) {
                contentValues = ContentValues()
                contentValues.put(DbHelper.AUTHORS_COLUMN_NAME, author)
                database.insertWithOnConflict(DbHelper.AUTHORS_TABLE_NAME, null, contentValues,
                    SQLiteDatabase.CONFLICT_IGNORE)
            }
        }
    }
}

```



```

        database.setTransactionSuccessful()
        Log.d("MyTag", "wrote data to database")
    } catch (e: Exception) {
        Log.d("MyTag", "there was a problem inserting data: " + e.message)
    } finally {
        Log.d("MyTag", "finally")
        database.endTransaction()
    }
}

fun insertOrUpdate(database: SQLiteDatabase, table: String,
                  values: ContentValues, where: String,
                  args: Array<String>): Boolean {
    val rowId = database.insertWithOnConflict(table, null, values,
        SQLiteDatabase.CONFLICT_IGNORE)
    if (rowId < 0) {
        database.update(table, values, where, args)
        return false
    }
    return true
}

fun readAuthors(database: SQLiteDatabase, book: Book) {
    val cursor = database.query(
        DbHelper.AUTHORS_TABLE_NAME
            + " join "
            + DbHelper.BRIDGE_TABLE_NAME
            + " on "
            + DbHelper.AUTHORS_TABLE_NAME + "." + DbHelper.AUTHORS_COLUMN_ID
            + " = "
            // Строка, имя запрашиваемой таблицы
            + DbHelper.BRIDGE_TABLE_NAME + "." + DbHelper.BRIDGE_COLUMN_AUTHOR_ID,
        arrayOf(DbHelper.AUTHORS_COLUMN_NAME), // Строка, массив столбцов
        DbHelper.BRIDGE_COLUMN_AUTHOR_ID + " = ?", // Строка, предложение WHERE
        arrayOf(book.isbn), null, null, null
    )
    var i = 0
    while (!cursor.isAfterLast) {
        cursor.moveToNext()
        book.authors[i++] = cursor.getString(0)
    }
    cursor.close()
}
}
}

```

Теперь, имея вспомогательные методы чтения/записи данных `Book`, добавим в наш класс `DbHelper` чтение содержимого файла `catalog.json` в методе `onCreate` и его запись в базу данных!

Java

```

public class DbHelper extends SQLiteOpenHelper {
    private Context mContext;

```

```

public DBHelper(Context context, String name,
                SQLiteDatabase.CursorFactory factory,
                int version) {
    super(context, name, factory, version);
    mContext = context;
}

@Override
public void onCreate(SQLiteDatabase database) {
    database.execSQL(BOOKS_CREATE_TABLE);
    database.execSQL(AUTHORS_CREATE_TABLE);
    database.execSQL(BRIDGE_CREATE_TABLE);
    InputStream stream = null;
    try {
        stream = mContext.getAssets().open("catalog.json");
        String json = Files.getStringFromStream(stream);
        Books books = new Gson().fromJson(json, Books.class);
        for (Book book : books) {
            BookTransactions.write(database, book);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (stream != null) {
            try {
                stream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
Log.d("MyApp", BookTransactions.read(database, "978-0439136365").getTitle());
}

@Override
public void onUpgrade(SQLiteDatabase database, int i, int i1) {
    // ничего не делает
}

public static final String BOOKS_TABLE_NAME = "BOOKS";
public static final String BOOKS_COLUMN_TITLE = "TITLE";
public static final String BOOKS_COLUMN_ISBN = "ISBN";
public static final String BOOKS_COLUMN_PAGECOUNT = "PAGECOUNT";
public static final String BOOKS_COLUMN_ISFICTION = "ISFICTION";

public static final String AUTHORS_TABLE_NAME = "AUTHORS";
public static final String AUTHORS_COLUMN_ID = "ID";
public static final String AUTHORS_COLUMN_NAME = "NAME";

public static final String BRIDGE_TABLE_NAME = "BOOK_AUTHOR_BRIDGE";
public static final String BRIDGE_COLUMN_BOOK_ID = "BOOK_ID";
public static final String BRIDGE_COLUMN_AUTHOR_ID = "AUTHOR_ID";

public static final String BOOKS_CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " +
    BOOKS_TABLE_NAME + " (" +

```

```

    BOOKS_COLUMN_TITLE + " TEXT," +
    BOOKS_COLUMN_ISBN + " TEXT," +
    BOOKS_COLUMN_PAGECOUNT + " INT," +
    BOOKS_COLUMN_ISFICTION + " BOOL);";

public static final String AUTHORS_CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " +
    AUTHORS_TABLE_NAME + " (" +
    AUTHORS_COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
    AUTHORS_COLUMN_NAME + " TEXT)";

public static final String BRIDGE_CREATE_TABLE = "CREATE TABLE IF NOT EXISTS " +
    BRIDGE_TABLE_NAME +
    " (" +
    BRIDGE_COLUMN_BOOK_ID + " INTEGER REFERENCES BOOKS," +
    BRIDGE_COLUMN_AUTHOR_ID + " INTEGER REFERENCES AUTHORS)";
}

```

Kotlin

```

class DBHelper(private val context: Context, name: String,
    factory: SQLiteDatabase.CursorFactory,
    version: Int) :
    SQLiteOpenHelper(mContext, name, factory, version) {

    override fun onCreate(database: SQLiteDatabase) {
        database.execSQL(BOOKS_CREATE_TABLE)
        database.execSQL(AUTHORS_CREATE_TABLE)
        database.execSQL(BRIDGE_CREATE_TABLE)
        val stream = context.assets.open("catalog.json")
        stream.use { s ->
            val json = Files.getStringFromStream(s)
            val books = Gson().fromJson(json, Books::class.java)
            for (book in books) {
                BookTransactions.write(database, book)
            }
        }
    }

    override fun onUpgrade(database: SQLiteDatabase, i: Int, i1: Int) {
        // ничего не делает
    }

    companion object {

        const val BOOKS_TABLE_NAME = "BOOKS"
        const val BOOKS_COLUMN_TITLE = "TITLE"
        const val BOOKS_COLUMN_ISBN = "ISBN"
        const val BOOKS_COLUMN_PAGECOUNT = "PAGECOUNT"
        const val BOOKS_COLUMN_ISFICTION = "ISFICTION"

        const val AUTHORS_TABLE_NAME = "AUTHORS"
        const val AUTHORS_COLUMN_ID = "ID"
        const val AUTHORS_COLUMN_NAME = "NAME"

        const val BRIDGE_TABLE_NAME = "BOOK_AUTHOR_BRIDGE"
        const val BRIDGE_COLUMN_BOOK_ID = "BOOK_ID"
    }
}

```

```

const val BRIDGE_COLUMN_AUTHOR_ID = "AUTHOR_ID"

const val BOOKS_CREATE_TABLE =
    "CREATE TABLE IF NOT EXISTS " + BOOKS_TABLE_NAME +
    " (" +
    BOOKS_COLUMN_TITLE + " TEXT," +
    BOOKS_COLUMN_ISBN + " TEXT," +
    BOOKS_COLUMN_PAGECOUNT + " INT," +
    BOOKS_COLUMN_ISFICTION + " BOOL);"

const val AUTHORS_CREATE_TABLE =
    "CREATE TABLE IF NOT EXISTS " + AUTHORS_TABLE_NAME +
    " (" +
    AUTHORS_COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
    AUTHORS_COLUMN_NAME + " TEXT)"

const val BRIDGE_CREATE_TABLE =
    "CREATE TABLE IF NOT EXISTS " + BRIDGE_TABLE_NAME +
    " (" +
    BRIDGE_COLUMN_BOOK_ID + " INTEGER REFERENCES BOOKS," +
    BRIDGE_COLUMN_AUTHOR_ID + " INTEGER REFERENCES AUTHORS)"
}
}

```

Теперь при первой попытке получить экземпляр `SqliteDatabase` в классе `DbHelper` запустится метод `onCreate` и заполнит базу данных содержимым файла `catalog.json`, который устанавливается вместе с приложением, в специальном каталоге `assets`! Неплохо!

Мы потратили много времени на создание слоя хранения данных. Зато теперь готовы показать на экране сохраненные книги, но прежде чем двинуться дальше, реализуем все то же самое в iOS.

iOS

В Android мы использовали базу данных, к которой можно выполнять запросы SQL для получения данных. В iOS есть возможность работать с хранилищем непосредственно из кода на Swift, без посредничества SQL: механизм Core Data.

Core Data

Давайте сразу проясним: Core Data не является базой данных – это *граф объектов*, который *использует* базу данных в роли резервного хранилища. Этот механизм доступен в iOS (и на других платформах Apple), и его целью является упрощение операций со слоем модели данных и обеспечение прозрачного сохранения данных в постоянном хранилище.

Механизм Core Data способен удовлетворить самые сложные требования и является отличным выбором в 90 % случаев. Он идеально подходит для нашего приложения, поэтому давайте используем его в нашем проекте.

Чтобы добавить поддержку Core Data в проект, необходимо добавить:

- 1) файл модели данных и определить все сущности;
- 2) контроллер данных, выполняющий настройку стека Core Data;
- 3) инициализацию в логике запуска приложения.

Файл модели данных

Первым в нашем списке упоминается файл модели данных. Чтобы создать его, выберите в меню пункт **File** ⇒ **New** ⇒ **File** (Файл ⇒ Создать ⇒ Файл) и найдите в открывшемся диалоге раздел **Core Data**. Выберите **Data Model** (Модель данных) и щелкните на кнопке **Next** (Далее). Давайте, не мудрствуя лукаво, дадим нашей модели имя `LibraryModel`, но вообще вы можете назвать ее как угодно. Среди файлов проекта должен появиться файл с именем `LibraryModel.xcdatamodel`.

Core Data использует сущности с атрибутами, соответствующими свойствам хранимых объектов. Если открыть файл `LibraryModel.xcdatamodel`, вы увидите пустой список сущностей.

Теперь мы должны принять важное решение.

Можно продолжить использовать существующую структуру `Book` и определить отдельную сущность для хранения книг, которая содержит лишь некоторый идентификатор (например, уникальную строку). Это позволит сохранить стек Core Data максимально легковесным. Однако при таком подходе Core Data будет действовать подобно базе данных, и нам будут недоступны все его возможности в управлении слоем модели данных.

Вместо этого мы продолжим использовать файл `catalog.json` в качестве источника данных, но только при первом запуске приложения. Это позволит нам применять Core Data в виде своеобразного кеша и избавит от необходимости читать файл в память целиком всякий раз, когда потребуется просмотреть каталог библиотеки.

Чтобы добавить сущность в Core Data, щелкните на кнопке **Add Entity** (Добавить сущность) в нижней части экрана. В результате в список сущностей будет добавлена новая сущность с типовым именем `Entity`. Выберите ее, чтобы отобразить инспектор сущностей в правой части экрана. Измените имя сущности с `Entity` на `Book` и в поле **Class** измените имя класса на `BookManagedObject`, чтобы отделить фактическое имя файла управляемого объекта от имени сущности. Это поможет нам понять, что мы имеем дело с объектом, управляемым механизмом Core Data, при работе с объектами типа `BookManagedObject`.

Свойства в файле `Book.swift` необходимо воссоздать как атрибуты нашей сущности `Book`. Для этого щелкните на значке «плюс» в разделе **Attributes** (Атрибуты) и добавьте свойства и связанные с ним типы (например, `String`, `Int`, `Bool` и т. д.), соответствующие свойствам в имеющейся структуре `Book`.



Для свойства `authors` укажите тип **Transformable** (Преобразуемый). Затем выберите этот атрибут в инспекторе модели данных и установите в поле **Custom Class** тип `[String]`, чтобы показать, что это массив строк.

Теперь, определив сущность `Book`, заставим Xcode сгенерировать файл с исходным кодом. Выберите сущность `Book` в редакторе. Откройте инспектор модели данных и в **Codegen** замените значение **Class Definition** на **Category/Extension**. Выберите в меню пункт **Editor** ⇒ **Create NSManagedObject Subclass** (Редактор ⇒ Создать подкласс NSManagedObject) и в открывшемся мастере щелкайте на кнопке **Next** (Далее) до его завершения, оставляя все настройки со значениями по умолчанию.

Когда мастер закончит работу, вы увидите два новых файла: *BookManaged-Object+CoreDataClass.swift* и *BookManagedObject+CoreDataProperties.swift*. Первый файл нам не понадобится, поэтому просто отправьте его в корзину. Второй файл содержит все атрибуты Core Data, которые мы добавили как свойства Swift, снабженные аннотацией `@NSManaged`. Это синтаксический сахар, сообщающий компилятору, что этим конкретным свойством управляет Core Data.

Потом мы используем это расширение для добавления некоторых функций в нашу модель, а пока закончим настройку остальной части стека Core Data.

Инициализация стека

Стек Core Data состоит из трех частей:

- 1) файла модели данных, который мы только что создали. Он хранит описание сущности;
- 2) контейнера хранилища, связывающего приложение с самим хранилищем. В качестве хранилища мы будем использовать базу данных SQLite, но вообще хранилищем может быть XML-файл или хранилище в памяти;
- 3) контекста управляемого объекта – своего рода «кеша», в котором хранятся все экземпляры *BookManagedObject*, пока они активны; они в точности соответствуют объектам, хранимым в базе данных.

Мы выполнили пункт 1, создав файл модели данных *LibraryModel.xcdatamodel*. Но нам еще нужно выполнить пункты 2 и 3. Обычно инициализацию стека Core Data выполняют на запуске приложения. Для простоты мы заключим все необходимое в объект *DataController*.

В Xcode создайте новый файл Swift с именем *DataController* и добавьте его в проект. Введите в него следующий код:

```
import Foundation
import CoreData

class DataController {
    var persistentContainer: NSPersistentContainer

    init(completion: @escaping () -> ()) {
        persistentContainer = NSPersistentContainer(name: "LibraryModel")
        persistentContainer.loadPersistentStores { (description, error) in
            if let error = error {
                fatalError("Core Data stack could not be loaded. \(error)")
            }

            // Вызывается после инициализации стека Core Data
            DispatchQueue.main.async {
                completion()
            }
        }
    }
}
```

Рассмотрим поближе, что здесь происходит. Во-первых, в методе `init(completion:)` класса *DataController* мы инициализировали обработчик завершения. Этот обработчик будет вызван после настройки Core Data и позволит продолжить запуск приложения.

i Хранилище будет загружаться последовательно в любом потоке, выполнившем вызов. Мы можем загрузить хранилище асинхронно, добавив в `description` свойство `shouldAddStoreAsynchronously` со значением `true`. Однако в этом примере мы не используем эту возможность, чтобы упростить загрузку стека Core Data.

Внутри инициализатора мы сначала присваиваем переменной экземпляра `persistentContainer` ссылку на новый экземпляр `NSPersistentContainer`. Этот контейнер выполняет большую часть настроек и связывает файл модели данных с координатором хранилища. Для этого мы передаем его конструктору имя файла модели данных без расширения. После создания объекта `NSPersistentContainer` мы вызываем его метод `loadPersistentStores(_:)` и передаем ему замыкание, выполняющее проверку ошибок. Если база данных недоступна, файл модели не может быть прочитан или возникла какая-то другая ошибка, мы вызываем `fatalError`, чтобы завершить приложение. Если же хранилище было загружено правильно, мы вызываем `completion()` – замыкание, которое передали в вызов инициализатора `DataController`.

Теперь хранилище настроено, и в процессе его настройки мы настроили контекст управляемого объекта, доступный как свойство контейнера. Позже мы подробнее расскажем, как его использовать, а пока закончим настройку стека и используем только что созданный объект `DataController`.

Откройте файл `AppDelegate.swift`. Добавьте новое свойство `dataController` с типом `DataController!`. Вставьте в метод `application(_:didFinishLaunchingWithOptions:)` следующие строки перед инструкцией `return true` в самом конце:

```
// Инициализировать стек Core Data
dataController = DataController() {
    // Точка переопределения для обновления пользовательского
    // интерфейса после завершения инициализации
}
```

Теперь класс `AppDelegate` должен выглядеть так:

```
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?
    var dataController: DataController!

    func application(_ application: UIApplication,
                    didFinishLaunchingWithOptions launchOptions:
                    [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // Инициализировать стек Core Data
        dataController = DataController() {
            print("Core Data stack has been initialized.")
        }

        return true
    }

    ...
}
```

Если теперь собрать и запустить приложение, вы увидите в консоли Xcode сообщение: «Core Data stack has been initialized».

Да! Наш стек Core Data запущен и работает, однако в данный момент он ничем не управляет. Все данные до сих пор извлекаются из *catalog.json*. Давайте переключим наш `ListDataSource` с использования JSON-файла на Core Data.

Переключение с файла JSON на Core Data

С этой целью сначала определим новое расширение для `BookManagedObject`. Для этого можно создать новый файл. В Swift считается обычной практикой, когда расширения определяются в файлах с именами вида: *BookManagedObject+Extensions.swift*. Внутри этого файла определим новый контроллер результатов, который по сути является объектом, управляющим результатами выборки из Core Data. Вот наше расширение:

```
import Foundation
import CoreData

extension BookManagedObject {
    class func newCatalogResultsController(with dataController: DataController,
        delegate: NSFetchedResultsControllerDelegate?) ->
        NSFetchedResultsController<BookManagedObject> {
        let request = NSFetchedRequest<BookManagedObject>(entityName: "Book")

        // Добавить в запрос сортировку по названию
        let titleSort = NSSortDescriptor(key: "title", ascending: true)
        request.sortDescriptors = [titleSort]

        let context = dataController.persistentContainer.viewContext
        let fetchedResultsController =
            NSFetchedResultsController(fetchRequest: request,
                managedObjectContext: context,
                sectionNameKeyPath: nil,
                cacheName: "catalog.cache")

        // Назначить делегата для обработки обновлений
        fetchedResultsController.delegate = delegate

        do {
            try fetchedResultsController.performFetch()
        } catch {
            fatalError("Catalog fetch could not be completed. \(error)")
        }

        return fetchedResultsController
    }
}
```

Здесь выполняется довольно много разных операций, но наибольший интерес представляют лишь некоторые из них. Сначала мы создаем новый запрос на извлечение данных. Запросы на извлечение – это способ получения объектов модели из Core Data. Мы указываем, что результаты должны сортироваться по названию – это, пожалуй, наиболее подходящий способ сортировки для списка книг. Затем мы записываем в переменную `viewContext` контекст управляемого объекта, который собираемся использовать для выборки данных. В данном случае `viewContext` – это контекст, в котором выполняется главный поток. Счи-

тается допустимым и даже рекомендуется использовать управляемые объекты в главном потоке. Однако если бы мы сами писали объекты, то выполняли бы эти действия в фоновом потоке.

После получения контекста мы создаем объект контроллера результатов типа `NSFetchedResultsController`. Это специальный объект, используемый в `Core Data` для управления извлеченными объектами и передачи их в источники данных, такие как источник данных табличного представления или источник данных представления коллекции. Позже мы реализуем в `ListDataSource` поддержку протокола делегирования `NSFetchedResultsControllerDelegate`, который назначается здесь, чтобы дать ему возможность получать обновления и заполнять табличное представление, но сейчас мы просто записываем в свойство `fetchedResultsController` то, что было передано.

Наконец, вызовом `executeFetch()` выполняется фактическая выборка из `Core Data`. Этот метод может сгенерировать исключение, поэтому мы заключили его в блок `try`, в котором перехватываем любые ошибки. В этом примере мы просто вызываем `fatalError`, но в действующем приложении важно перехватить и обработать ошибку соответствующим образом или показать сообщение пользователю.

Теперь воспользуемся этим расширением и подключим наш источник данных.

Вернитесь к определению класса `ListDataSource`. Нам нужно внести в него несколько изменений. Во-первых, нужно удалить свойство `data`, которое мы использовали для отложенной загрузки файла `JSON`. Замените это новым свойством `fetchedResultsController`:

```
var fetchedResultsController: NSFetchedResultsController<BookManagedObject>?
```

Во-вторых, нужно добавить новый метод с именем `fetchCatalogResults`, который создаст контроллер результатов, определенный нами только что, и сохранит его в новом свойстве, добавленном выше:

```
func fetchCatalogResults(with dataController: DataController) {
    fetchedResultsController =
        BookManagedObject.newCatalogResultsController(
            with: dataController, delegate: nil)
}
```

В-третьих, добавим новый метод с именем `fetchCatalogResults`, который инициализирует экземпляр контроллера результатов, созданный перед этим. Затем сохраним этот экземпляр в новом свойстве:

```
import UIKit
import CoreData

class ListDataSource: NSObject {
    var fetchedResultsController: NSFetchedResultsController<BookManagedObject>?

    func fetchCatalogResults(with dataController: DataController,
        delegate: NSFetchedResultsControllerDelegate?) {
        fetchedResultsController =
            BookManagedObject.newCatalogResultsController(
                with: dataController, delegate: delegate)
    }
}
```

```

    }

    func book(for indexPath: IndexPath) -> Book? {
        return fetchedResultsController?.object(at: indexPath).book
    }
}

extension ListDataSource: UITableViewDataSource {
    func tableView(_ tableView: UITableView,
                   numberOfRowsInSection section: Int) -> Int {
        return fetchedResultsController?.sections?[section].numberOfObjects ?? 0
    }

    func tableView(_ tableView: UITableView,
                   cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        // Получить ячейку из пула свободных ячеек
        let cell =
            tableView.dequeueReusableCell(withIdentifier: "CatalogTableViewCell",
                                         for: indexPath)

        // Найти книгу, соответствующую заполняемой ячейке
        guard let book = fetchedResultsController?.object(at: indexPath) else {
            fatalError("Could not retrieve book instance.")
        }

        // Заполнить метку с заголовком ячейки названием книги
        cell.textLabel?.text = book.title

        return cell
    }
}
}

```

Мы использовали `fetchedResultsController` для передачи результатов везде, где это возможно. Единственное, на что нужно обратить внимание, – в методе `book(for:)` мы получаем `BookManagedObject`, а затем преобразуем его с помощью метода расширения, созданного ранее, в экземпляр `Book`. Мы продолжим использовать экземпляры `Book` повсюду в контроллере представления и в слое представлений нашего приложения вместо передачи экземпляров управляемых объектов.

Последнее, что осталось сделать, – переключить контроллер `CatalogViewController` на использование новых методов источника данных. Это делается вызовом `dataSource.fetchCatalogResults(with:delegate:)` в свойстве `dataSource`:

```

lazy var dataSource: ListDataSource = {
    let listDataSource = ListDataSource()
    let dataController = (UIApplication.shared.delegate as?
        AppDelegate)?.dataController!
    listDataSource.fetchCatalogResults(with: dataController, delegate: nil)
    return listDataSource
}()

```

Соберите и запустите приложение, и... вы увидите, что каталог пуст!

Все дело в том, что мы создали объекты `Core Data`, но *не* заполнили их данными. С этой целью приложения в iOS часто связывают либо с предварительно заполненной базой данных, либо с какими-то данными, хранящимися в ин-

тернете или внутри пакета приложения. В данном случае мы будем использовать файл *catalog.json* и заполним базу данных его содержимым, но только если перед этим базы данных не существовало.

Заполнение базы данных начальными данными

Откройте класс *DataController*, созданный ранее. Добавьте следующий код в начало метода *init(with:)* и новую переменную *shouldSeedDatabase* в определение класса:

```
var shouldSeedDatabase: Bool = false

init(completion: @escaping () -> ()) {
    // Проверить наличие базы данных
    do {
        let databaseUrl =
            try FileManager.default.url(for: .applicationSupportDirectory,
                                       in: .userDomainMask,
                                       appropriateFor: nil,
                                       create: false).appendingPathComponent("LibraryModel.sqlite")
        shouldSeedDatabase = !FileManager.default.fileExists(
            atPath: databaseUrl.path)
    } catch {
        shouldSeedDatabase = true
    }
    ...
}
```

Сначала мы получаем путь к базе данных там. Затем проверяем наличие файла базы данных и присваиваем переменной *shouldSeedDatabase* значение *true*, если его нет. Это простая, но полезная логика. Теперь создадим метод, который фактически заполнит базу данных. Назовем это *seedData():*

```
private func seedData() {
    do {
        guard let rawCatalogData =
            try? Data(contentsOf:
                Bundle.main.bundleURL.appendingPathComponent("catalog.json"))
            else {
                return
            }
        let books = try JSONDecoder().decode([Book].self, from: rawCatalogData)

        persistentContainer.performBackgroundTask { (managedObjectContext) in
            // Обойти все книги в файле JSON и добавить в базу данных
            books.forEach { (book) in
                let bookManagedObject =
                    BookManagedObject(context: managedObjectContext)
                bookManagedObject.title = book.title
                bookManagedObject.authors = book.authors
                bookManagedObject.isbn = book.isbn
                bookManagedObject.pageCount = Int16(book.pageCount)
                bookManagedObject.fiction = book.fiction
            }
        }
    }
}
```

```

        do {
            try managedObjectContext.save()
        } catch {
            print("Could not save managed object context. \(error)")
        }
    }
} catch {
    print("Catalog.json was not found or is not decodable.")
}
}
}

```

Рассмотрим подробнее, что делает этот код.

Сначала мы открываем файл *catalog.json*, находящийся в пакете приложения. Затем преобразуем его содержимое из формата JSON в массив объектов *Book*. В следующей строке, начинающейся с `persistentContainer.performBackgroundTask`, используем контейнер *NSPersistentContainer*, который мы создали в ходе инициализации стека *Core Data*, для выполнения задачи в фоновом режиме. Это важно, потому что эта операция в конечном итоге выполняет запись данных в постоянное хранилище и могла бы заблокировать основной поток до ее завершения. В результате вызова этого метода мы получаем *NSManagedObjectContext*, который затем используется для создания нового *BookManagedObject* в цикле, перебирающем массив книг из JSON.

Все экземпляры *BookManagedObject* создаются в фоновом контексте управляемого объекта. Важно отметить, он существует в памяти только в данный момент. И лишь когда мы вызовем метод `save()` контекста управляемого объекта, объекты *BookManagedObject* и сохранятся в координаторе постоянного хранилища и, если необходимо, в базе данных.

Однако если теперь запустить приложение, оно не заполнит базу данных. Причина в том, что база данных уже существует, и этот код проверяет просто наличие файла, но не проверяет, была ли она заполнена. Как одно из решений: удалите приложение из *iOS Simulator*, а затем снова соберите и запустите его. Если все пойдет как надо, вы увидите в каталоге список тех же книг, что и раньше, отсортированный по названиям.

Уф! Мы сделали это. Теперь мы используем базу данных, так же как в приложении для *Android*. Однако мы пошли немного дальше, чтобы воспользоваться всеми преимуществами *Core Data*, которые недоступны с простой базой данных *SQLite*.

Но мы еще не закончили! Теперь нам нужно использовать эту базу данных для первоначальной цели: сохранения книг в закладках. Вернемся к этой части проекта!

СОХРАНЕНИЕ КНИГ В ЗАКЛАДКАХ

Итак, наш слой хранения данных сохраняет книги в базе данных. Если позже нам предложат воспользоваться удаленной службой для добавления в базу данных большего количества книг из библиотеки, у нас уже будет иметься вся необходимая для этого инфраструктура. Давайте вспомним, что нам осталось

сделать. Помните ту кнопку, которую мы добавили для сохранения понравившейся книги в закладках? Давайте подключим ее.

Android

Как уже не раз отмечалось, сделать это можно несколькими способами, и кто-то из вас мог бы предложить добавить в таблицу BOOKS столбец типа INTEGER и сохранять в нем 0 (нет) или 1 (да), в зависимости от включения книги в закладки (в отличие от большинства баз данных SQL, SQLite не имеет логического типа данных – вместо них обычно используются целые числа, как только что было описано). На практике лично я так бы и поступил, но в этом примере мы используем альтернативный подход, с одной стороны, для простоты, а с другой – для демонстрации еще одной распространенной и удобной особенности платформы Android, обсуждавшейся в первой части «Задачи и операции», в главе 11.

Как уже упоминалось, класс SharedPreferences принимает множество (Set) строк (String). Он кажется идеальной структурой данных для нашей цели – нам нужна уникальная неупорядоченная коллекция идентификаторов книг (номеров ISBN), по которым легко можно определить, находится ли книга в закладках. Если включить флажок, мы добавим ISBN в множество, если выключить – удалим его. Все просто! В какой-то момент у вас может появиться желание добавить в пользовательский интерфейс некоторый визуальный признак, например звезду, которая меняет свой цвет с золотого на серый, или специальный стиль отображения в списке, но сейчас мы просто обновим текст кнопки, чтобы он отражал текущее состояние.

Итак, сначала создадим метод, добавляющий книгу в Set<String> в SharedPreferences. Определим эту функцию в классе BookDetailActivity, потому что в этом случае мы получим локальный доступ к объекту Context (самому экземпляру Activity) и всем его дочерним компонентам:

Java

```
private void toggleFavorite(Button button, String isbn) {
    SharedPreferences preferences = getSharedPreferences("prefs", Context.MODE_PRIVATE);
    Set<String> favorites = preferences.getStringSet("favorites", new HashSet<>());
    if (favorites.contains(isbn)) {
        favorites.remove(isbn);
        button.setText("Mark as Favorite");
    } else {
        favorites.add(isbn);
        button.setText("Remove from favorites");
    }
    SharedPreferences.Editor editor = preferences.edit();
    editor.putStringSet("favorites", favorites);
    editor.apply();
}
```

Kotlin

```
private fun toggleFavorite(button: Button, isbn: String) {
    val preferences = getSharedPreferences("prefs", Context.MODE_PRIVATE)
```

```

val favorites = preferences.getStringSet("favorites", HashSet())
favorites?.let {
    if (it.contains(isbn))
        it.remove(isbn)
        button.text = "Mark as Favorite"
    } else {
        it.add(isbn)
        button.text = "Remove from favorites"
    }
}
val editor = preferences.edit()
editor.putStringSet("favorites", favorites)
editor.apply()
}

```

Этот метод можно использовать в качестве обработчика щелчка на кнопке, как показано ниже:

Java

```

findViewById(R.id.button_save).setOnClickListener(
    view -> toggleFavorite((Button) view, book.getIsbn()));

```

Kotlin

```

button_save.setOnClickListener{view -> toggleFavorite(view as Button, book.isbn)}

```

Этого должно быть достаточно! Запустите приложение, выберите книгу из списка и на странице с подробной информацией нажмите кнопку, отвечающую за сохранение книги в закладках. Поскольку это значение находится в хранимой структуре, оно будет сохраняться между запусками приложения.

Что мы узнали

Поздравляем! Теперь ваше приложение обладает разнообразными возможностями. Приложение уже можно использовать, и оно совместимо с Android и iOS. В этой главе мы узнали об особенностях хранения данных в постоянных хранилищах и насколько они отличаются между Android и iOS. В Android используется гораздо более простой подход к использованию баз данных для хранения информации, в то время как в iOS применяется более абстрактное решение, предлагающее больше возможностей, хотя и за счет увеличения сложности.

Прежде чем мы отправимся в закат с нашим приложением, рассмотрим еще один аспект: выполнение сетевых операций. В следующей главе мы посмотрим, какие подводные камни нас поджидают при создании сетевых подключений.

Глава 20

Сетевые операции в приложении

Мы прошли долгий путь, разрабатывая наше приложение. Предыдущая глава, посвященная базам данных, была одной из самых насыщенных. Теперь у нас есть отличное приложение для просмотра каталога книг. Мы можем даже сохранить понравившиеся книги в закладках, чтобы вернуться к ним позже. Отлично! Пожалуй, что пришло время отложить это приложение и начать работу над каким-нибудь другим проектом.

Но подождите! Вот к нам подходит наш старый библиотекарь и открывает секрет, от которого волосы встают дыбом! Как оказывается, это не единственная такая библиотека. Во многих странах существуют точно такие же библиотеки, и ваше приложение должно отображать информацию обо всех этих библиотеках, чтобы помочь читателям, таким же, как мы сами, открыть для себя знания, содержащиеся в них.

Похоже, что мы еще не завершили наш проект.

Теперь нам нужно реализовать эту новую возможность. Но как это сделать? Можно ли использовать тот же подход, что использовался для реализации каталога: добавить в приложение поддержку списка библиотек? Такой подход выглядит чересчур статичным. А что, если одна из библиотек закроется и наш пользователь будет тщетно искать необходимые ему знания в ней? Может быть, лучше воспользоваться удобным случаем и, наконец, сломать барьеры и позволить приложению общаться с внешним миром через, как вы уже догадались, интернет?! Это позволит приложению динамически обновлять данные, изменяющиеся с течением времени, как и окружающий нас мир.

Давайте попробуем! Пристегните ремни – и начнем!

Поиск в сети

Взаимодействия с конечной точкой поисковой системы в Android и в iOS реализуются практически одинаково. Для этого нужно выполнить следующие шаги:

- 1) ввести искомую фразу в поле ввода;
- 2) отправить введенную строку сетевой службе поиска;
- 3) получить ответ, содержащий результаты поиска;

- 4) вывести результаты на экран, например в представление списка или таблицы.

Итак, сначала мы должны добавить в пользовательский интерфейс что-то, что запустит процесс поиска. В Android, поскольку отказались от использования панели ActionBar, добавим в верхнюю часть BrowseContentActivity компонент SearchView. После ввода строки и активизации компонента (щелчком на кнопке или нажатием клавиши **Enter**) отправим введенную строку веб-службе, а затем, в случае успеха, откроем новый экран (Activity) и отобразим в нем полученные результаты.

Давайте изменим макет *res/layout/activity_browse.xml* и добавим в него SearchView:

```
<?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"
    android:orientation="vertical">

    <SearchView
        android:id="@+id/search_locations"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#FFFFFF" />

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/browse_content_recyclerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#FFFFFF" />

</LinearLayout>
```

Теперь добавим новые возможности в существующий Activity:

- 1) добавим новое представление SearchView в манифест;
- 2) отобразим представление SearchView;
- 3) добавим инструкции журналирования в метод поиска в локальном проекте;
- 4) исправим значки.

Java

```
public class BrowseContentActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_browse);
        SearchView searchView = findViewById(R.id.search_locations);
        searchView.setOnQueryTextListener(new SearchView.OnQueryTextListener() {
            @Override
            public boolean onQueryTextSubmit(String query) {
                Log.d("MyApp", "searching on " + query);
                return false;
            }
        });
    }
}
```



```

    }
    @Override
    public boolean onQueryTextChanged(String newText) {
        return false;
    }
});
RecyclerView recyclerView = findViewById(R.id.browse_content_recyclerview);
try {
    InputStream stream = getAssets().open("catalog.json");
    String json = Files.getStringFromStream(stream);
    Books source = new Gson().fromJson(json, Books.class);
    BrowseBooksAdapter adapter = new BrowseBooksAdapter(source);
    Log.d("MyApp", "books = " + adapter.getItemCount());
    recyclerView.setAdapter(adapter);
    recyclerView.setLayoutManager(new LinearLayoutManager(this));
} catch (IOException e) {
    Log.d("MyApp", "Oops! Something went wrong trying to read our catalog json");
    e.printStackTrace();
}
}
}

```

Kotlin

```

class BrowseContentActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_browse)
        search_locations.setOnQueryTextListener(object : SearchView.OnQueryTextListener {
            override fun onQueryTextSubmit(query: String): Boolean {
                Log.d("MyApp", "searching on $query")
                return false
            }
        })
        override fun onQueryTextChanged(newText: String): Boolean {
            return false
        }
    })
    try {
        val stream = assets.open("catalog.json")
        val json = Files.getStringFromStream(stream)
        val books = Gson().fromJson(json, Books::class.java)
        Log.d("MyApp", "${books.size}, ${books[0].title}")
        browse_content_recyclerview.adapter = BrowseBooksAdapter(books)
        browse_content_recyclerview.layoutManager = LinearLayoutManager(this)
    } catch (e: IOException) {
        Log.d("MyApp", "Oops! Something went wrong trying to read our catalog json")
        e.printStackTrace()
    }
}
}
}

```

В iOS мы добавим кнопку поиска в верхнюю часть экрана отображения каталога и реализуем обработчик события щелчка на кнопке, который отображает `UISearchController` с `UISearchBar` для ввода искомой строки, которую отправим конечной точке службы поиска.

А теперь перейдем к пользовательскому интерфейсу для отображения результатов!

Android

Подготовим простой макет и `Activity` для отображения результатов поиска:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#FFFFFF">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/search_results_recyclerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:visibility="gone"
        android:background="#FFFFFF" />

    <ProgressBar
        android:id="@+id/search_progress"
        android:indeterminate="true"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center" />

</FrameLayout>
```

Этот интерфейс обеспечит вывод вращающегося индикатора, чтобы показать пользователю, что приложение занято и ожидает получения результатов от сервера.

Наш экземпляр `Activity` ожидает получить строку из `SearchView` для отправки веб-службе поиска и действует в зависимости от ситуации.

Давайте сделаем некоторые предположения. Допустим, что URL веб-службы имеет вид `magic://my.app/search` (мы используем схему `magic://`, чтобы не забыть потом изменить URL, – большинство клиентов Java HTTP отвергают эту схему и возвращают дружелюбное описание ошибки) и включает параметр запроса GET с именем `query`. В ответ веб-служба возвращает документ JSON с массивом объектов `Location` или пустой массив, если поиск не дал результатов.

Допустим также, что документ JSON выглядит примерно так:

```
[
  {
    "street_address": "123 Eiffel Tower Street",
    "city": "Paris",
    "country": "France",
```

```
    "emoji": "🇫🇷",  
    "hours": "8am-7pm"  
  },  
  {  
    "street_address": "86 Libery Boulevard",  
    "city": "New York",  
    "country": "America",  
    "emoji": "🇺🇸",  
    "hours": "6am-10pm"  
  }  
]
```

Соответствующая структура данных выглядит так:

Java

```
public class Location {  
    @SerializedName("street_address")  
    private String mStreetAddress;  
    @SerializedName("city")  
    private String mCity;  
    @SerializedName("country")  
    private String mCountry;  
    @SerializedName("emoji")  
    private String mEmoji;  
    @SerializedName("hours")  
    private String mHours;  
  
    public String getStreetAddress() {  
        return mStreetAddress;  
    }  
  
    public void setStreetAddress(String streetAddress) {  
        mStreetAddress = streetAddress;  
    }  
  
    public String getCity() {  
        return mCity;  
    }  
  
    public void setCity(String city) {  
        mCity = city;  
    }  
  
    public String getCountry() {  
        return mCountry;  
    }  
  
    public void setCountry(String country) {  
        mCountry = country;  
    }  
  
    public String getEmoji() {  
        return mEmoji;  
    }  
}
```

```

public void setEmoji(String emoji) {
    mEmoji = emoji;
}

public String getHours() {
    return mHours;
}

public void setHours(String hours) {
    mHours = hours;
}
}

```

Kotlin

```

data class Location(
    var streetAddress: String,
    var city: String,
    var country: String,
    var emoji: String,
    var hours: String
)

```

Теперь у нас есть макет, модель и общая стратегия. Осталось только реализовать логику в коде. Этим мы займемся в следующем разделе.

iOS

В iOS мы используем немного иной, но очень похожий подход. Обычно для запуска поиска в iOS добавляют кнопку на панель навигации. Откройте *Main.storyboard* и выберите сцену приветствия. Из панели **Library** (Библиотека) перетащите элемент **Bar Button Item** на панель навигации и поместите его справа. Во время перетаскивания на панели появится контур, куда можно поместить кнопку. В инспекторе атрибутов в поле **System Item** выберите значение **Search**. При этом внешний вид кнопки на экране изменится – вместо простой надписи **Item** на кнопке появится изображение лупы.

Теперь экран с приветствием послужит нам отправной точкой. Мы добавим переход к другому контроллеру представления, который будет отображать панель с полем для ввода искомой строки и результаты поиска. Для простоты перетащите новый объект **Table View Controller** из панели **Library** (Библиотека) на холст рядом со сценой приветствия. Затем в структуре документа распахни-те табличное представление в сцене **Table View** и выберите объект ячейки. В инспекторе атрибутов справа в поле **Reuse Identifier** установите значение **LocationCell**. Также, если у вас появится такое желание, в инспекторе идентичности введите в поле **Title** значение **Locations**, чтобы этот заголовок отображался на этом экране.

Наконец, вернитесь к экрану приветствия и, удерживая нажатой клавишу **Ctrl**, перетащите кнопку поиска, которую мы добавили выше, в новую сцену **Locations** и выберите в поле **Kind** значение **Show**. После этого при каждом нажатии кнопки с лупой на экране с приветствием будет отображаться сцена с табличным представлением.

Если теперь собрать и запустить приложение, вы увидите новую кнопку поиска в правом верхнем углу экрана. Если щелкнуть на ней, должен появиться экран с пустой таблицей без результатов.

К сожалению, на этом наша работа в редакторе раскадровки заканчивается. Дальше, чтобы начать поиск, нам придется определить свой класс контроллера представления результатов. Добавьте в проект новый класс Cocoa Touch, наследующий `UITableViewController`, и дайте ему имя `LocationsTableViewController`, для этого выберите в меню пункт **File** ⇒ **New** ⇒ **File** (Файл ⇒ Создать ⇒ Файл). Чтобы переключить контроллер табличного представления на использование этого класса, выберите сцену **Locations** и в инспекторе идентичности введите в поле **Custom Class** значение `LocationsTableViewController`.

Управление поиском

В iOS есть удобный класс `UISearchController`, упрощающий разработку пользовательского интерфейса поиска. Однако этот класс нельзя использовать в редакторе раскадровки или в Interface Builder, поэтому нам придется создать его вручную.

Откройте файл `LocationsTableViewController.swift`. Сначала удалим весь шаблонный код, который Xcode добавляет автоматически, когда мы наследуем класс `UITableViewController`. В результате должен получиться такой контроллер представления:

```
import UIKit

class LocationsTableViewController: UITableViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    // Источник данных для табличного представления

    override func numberOfSections(in tableView: UITableView) ->
    Int {
        // #warning Неполная реализация, должна вернуть число разделов
        return 0
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection section:
    Int) -> Int {
        // #warning Неполная реализация, должна вернуть число записей
        return 0
    }
}
```

Теперь мы должны создать и сохранить контроллер поиска для последующего использования в начальном контроллере представления `LocationsTableViewController`. Как рассказывалось в обсуждении жизненного цикла контроллера представления в главе 2, это делается путем переопределения метода `viewDidLoad` контроллера представления. Наш контроллер наследует класс `UITableViewController`, который, в свою очередь, наследует `UIViewController`, а значит, он обладает тем же набором методов.

Добавьте следующий код в тело метода `viewDidLoad`:

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

    let searchController = UISearchController(searchResultsController: nil)
    searchController.searchBar.delegate = self
    searchController.searchBar.placeholder = "Search Locations by Country"
    searchController.obscuresBackgroundDuringPresentation = false
    searchController.definesPresentationContext = true

    navigationItem.searchController = searchController
    navigationItem.hidesSearchBarWhenScrolling = false
}
```

Этот код создает новый объект `UISearchController` и сохраняет его в переменной `searchController`. Затем мы устанавливаем свойство `delegate` строки поиска так, чтобы она вызывала контроллер каждый раз, когда пользователь пытается выполнить поиск. Продолжим придерживаться этого протокола и пока проигнорируем ошибку компилятора. Далее мы задаем текст-заполнитель для панели поиска. В данном случае это просто строка `Search Locations by Country` (Поиск местоположений по странам), но это может быть любой другой текст, который вы решите отобразить, чтобы подсказать пользователям, что они могут искать.

Следующие две строки необходимы для отображения результатов поиска. В iOS для поддержки поиска используется `UISearchController`. Он имеет несколько свойств, которые нужно настроить, чтобы сделать поиск максимально удобным. Первое свойство, `obscuresBackgroundDuringPresentation`, предотвращает перекрытие контроллера поиска текущим контроллером представления, который используется для отображения результатов. Это важно, потому что при инициализации контроллера поиска мы не предоставляем новый контроллер `searchResultsController` непосредственно, поэтому результаты будут отображаться в этом же контроллере представления. Если этот контроллер представления окажется перекрыт представлением с результатами, пользователь не сможет воспользоваться им повторно, а это очень неудобно.

В связи с этим мы также должны присвоить свойству `definePresentationContext` значение `true`, чтобы когда `UISearchController` будет отображать представление с результатами, контроллер представления, в котором мы находимся, предоставлял контекст для отображения результатов и панели поиска. По сути, это означает, что этот контроллер представления будет скрывать панель поиска после перехода к другому представлению.

Наконец, мы записываем ссылку на только что созданный контроллер поиска в свойство `searchController` элемента навигации `navigationItem`. Это позволяет родительскому контроллеру навигации, который управляет этим представлением, отобразить панель поиска в панели навигации. Мы также устанавливаем свойство `hidesSearchBarWhenScrolling`, потребовав постоянно отображать панель поиска.

Теперь нам нужно заставить наш контроллер представления придерживаться протокола `UISearchBarDelegate`. Этот протокол необходим для обновления табличного представления, отображающего результаты поиска. При каждом

нажатии кнопки **Search** (или клавиши **Enter**) будет вызываться этот метод. Пока можно добавить неполную версию данного метода в `LocationsTableViewController` в виде расширения:

```
extension LocationsTableViewController: UISearchBarDelegate {
    func searchBarTextDidEndEditing(_ searchBar: UISearchBar) {
        // TODO: Отобразить результаты поиска в табличном представлении
    }
}
```

Если теперь собрать и запустить проект, вы сможете щелкнуть на кнопке с изображением лупы и увидеть табличное представление.

Теперь нам нужно заполнить это табличное представление, а значит, пришла пора заняться сетью!

СОЗДАНИЕ СЛУЖБЫ ПОИСКА

Как вы наверняка понимаете, мы не предполагали рассматривать в этой книге тему разработки веб-служб. Это ставит нас в тупик, потому что для связи с веб-службой нам нужна... веб-служба. Есть несколько способов решить эту проблему. Если хотите, можете просто следовать за примерами кода, не обращаясь к веб-службе. К счастью, приложение будет работать как обычно, просто не будет отображать результатов. Однако чуть ниже мы покажем файл JSON с местоположениями библиотек, который будет использовать приложение. Вы сможете поместить его в локальную папку, и приложение получит этот файл.

! Если вы решите использовать локальный файл, вам нужно будет указать тип содержимого `Content-Type` как `application/json`. Некоторые службы, такие как Google Drive, поддерживают такую возможность, но обсуждение этого вопроса выходит за рамки данной книги.

Однако предпочтительнее на скорую руку создать службу `node.js`, которая обеспечит передачу содержимого файла. На самом деле мы, авторы книги, написали такую службу с использованием технологии `Express`. Если вы знакомы с `Node`, то легко справитесь с созданием подобной службы.

Установка Node и Express

Ниже приводится реализация самой простой веб-службы, которую можно встретить. Если вы уже знакомы с `Node`, то можете пропустить этот раздел. Если вам неинтересно знать, как устанавливать и использовать `Node`, то тоже можете спокойно пропустить этот раздел. Инструкции, что приводятся здесь, в основной своей массе взяты с сайтов `Node` и `Express`. Поехали!

1. Откройте страницу <https://nodejs.org> и установите последнюю версию `Node`.
2. Откройте терминал и создайте каталог для нового проекта `library-node-service`.

```
$ mkdir library-node-service
$ cd library-node-service
```

3. Инициализируйте новый проект Node, выполнив в терминале команду `npm init`. На выбор будет предложено несколько вариантов, но вы без всякой опаски можете просто нажать клавишу **Enter**, чтобы принять значения по умолчанию.
4. Выполните команду `npm install express --save`, чтобы установить Express в проект. Express – это облегченный фреймворк для быстрого создания служб. Он идеально подходит для нашего случая.
5. Создайте файл `index.js`, если он еще не был создан, и добавьте в него следующий код:

```
const express = require('express')
const PORT = process.env.PORT || 3000

express()
  .get('/catalog', (req, res) => res.json(catalog(req)))
  .get('/locations', (req, res) => res.json(locations(req)))
  .listen(PORT, () => console.log(`Listening on ${ PORT }`))

catalog = (req) => {
  var fs = require('fs');
  var json = JSON.parse(fs.readFileSync('catalog.json', 'utf8'));
  const query = (req.query.q || "").toLowerCase()
  return json.filter(book => book.title.toLowerCase().startsWith(query))
}

locations = (req) => {
  var fs = require('fs');
  var json = JSON.parse(fs.readFileSync('locations.json', 'utf8'));
  const query = (req.query.country || "").toLowerCase()
  return json.filter(loc => loc.country.toLowerCase().startsWith(query))
}
```

6. Скопируйте версию файла `catalog.json` из проекта в Xcode или Android Studio в каталог, где находится `index.js`. (Также скопируйте файл `location.json`, который мы создадим чуть ниже.)
7. Чтобы запустить приложение локально, введите в терминале команду `node index.js`. Теперь эта служба должна быть доступна по адресу `http://localhost:3000`. Это легко проверить, если открыть в браузере страницу `http://localhost:3000/catalog` – вы должны увидеть список всех книг из файла `catalog.json`.

Файл JSON с местоположениями библиотек

Мы снова будем использовать формат JSON в качестве транспорта для передачи данных. Вот как должен выглядеть файл `location.json`:

```
[
  {
    "street_address": "123 Eiffel Tower Street",
    "city": "Paris",
    "country": "France",
    "emoji": "🇫🇷",
    "hours": "8am-7pm"
```



```

    },
    {
        "street_address": "86 Libery Boulevard",
        "city": "New York",
        "country": "America",
        "emoji": "🇺🇸",
        "hours": "6am-10pm"
    },
    {
        "street_address": "49 Lombard Street",
        "city": "San Francisco",
        "country": "America",
        "emoji": "🇺🇸",
        "hours": "8am-8pm"
    },
    {
        "street_address": "1901 Aussie Way",
        "city": "Melbourne",
        "country": "Australia",
        "emoji": "🇦🇺",
        "hours": "7am-8pm"
    },
    {
        "street_address": "302 Deutsch Avenue",
        "city": "Berlin",
        "country": "Germany",
        "emoji": "🇩🇪",
        "hours": "9am-6pm"
    }
}
]

```

Как вы могли заметить, здесь перечислены местонахождения нескольких библиотек. Это всего лишь примеры (а не реальные библиотеки), которые используются для иллюстрации структуры данных. Вы можете добавить сюда свои библиотеки – не стесняйтесь проявлять творческую сметку!

ВЫЗОВ СЛУЖБЫ

Чтобы отправить запрос этой службе, в Android и iOS рекомендуется создать отдельный объект, отвечающий за сетевые взаимодействия, вместо непосредственного использования API на уровне представления. Поэтому далее создадим объект `LocationsController`, который будет играть роль посредника между пользовательским интерфейсом поиска и сетевой службой. Закончив создание объекта, мы включим его в работу.

Сначала реализуем объект в Android.

Android

В Android имеется ряд служб, готовых помочь в организации взаимодействий с веб-службами RESTful, но мы, как и прежде, вновь обратимся к стандартным

библиотекам и будем выполнять сетевые запросы с их помощью, как было показано в главе 9. Поскольку мы знаем, что результат будет иметь вид строки в формате JSON, то снова обратимся к библиотеке Gson, которая обсуждалась в главе 12. Так как мы уже видели многое из того, что потребуется в нашем приложении, например десериализацию JSON, взаимодействие RecyclerView с Adapter и передачу данных через Intent, перейдем прямо к коду. Далее представлена реализация Activity, включающая весь необходимый код поиска и пользовательский интерфейс. Если бы это приложение было больше или требовало большей гибкости, мы, вероятно, начали бы с выделения логических блоков в соответствующие классы и интерфейсы. Но, так как все это мы уже видели раньше, просто возьмем этот файл с кодом, написанным на скорую руку и включающим полный набор функций, и передадим его нашему заказчику (который, как мы знаем, никогда не отличался ни своим терпением, ни стремлением к элегантному коду):

Java

```
public class SearchResultsActivity extends Activity {
    public static final String QUERY = "QUERY";

    private ProgressBar mProgressBar;
    private RecyclerView mRecyclerView;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_search);
        mProgressBar = findViewById(R.id.search_progress);
        mRecyclerView = findViewById(R.id.search_results_recyclerview);
        mRecyclerView.setLayoutManager(new LinearLayoutManager(this));
        String query = getIntent().getStringExtra(QUERY);
        new Thread(() -> fetchResults("magic://my.app/search?query=" + query)).start();
    }

    private void onResultsReceived(List<Location> locations) {
        mProgressBar.setVisibility(View.GONE);
        mRecyclerView.setVisibility(View.VISIBLE);
        Adapter adapter = new Adapter(locations);
        mRecyclerView.setAdapter(adapter);
    }

    private void fetchResults(String urlWithQuery) {
        try {
            StringBuilder builder = new StringBuilder();
            URL url = new URL(urlWithQuery);
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            int data;
            while ((data = connection.getInputStream().read()) != -1) {
                builder.append((char) data);
            }
            String json = builder.toString();
            List<Location> locations = new Gson().fromJson(json,
                new TypeToken<List<Location>>().getType());
        }
    }
}
```

```

        onResultsReceived(locations);
    } catch (IOException e) {
        Log.d("MyApp",
            "Something went wrong when fetching results. Probably the fake URL! " +
            e.getMessage());
    }
}

private static class Adapter extends RecyclerView.Adapter<Adapter.ViewHolder> {
    private List<Location> mLocations;

    public Adapter(List<Location> locations) {
        mLocations = locations;
    }

    @Override
    public ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        return new ViewHolder(new TextView(parent.getContext()));
    }

    @Override
    public void onBindViewHolder(@NonNull ViewHolder holder, int position) {
        Location location = mLocations.get(position);
        holder.mTextView.setText(location.getStreetAddress() + ", " +
            location.getCity() + ", " + location.getCountry());
    }

    @Override
    public int getItemCount() {
        return mLocations.size();
    }

    private static class ViewHolder extends RecyclerView.ViewHolder {
        private TextView mTextView;
        public ViewHolder(View itemView) {
            super(itemView);
            mTextView = (TextView) itemView;
        }
    }
}
}
}

```

Kotlin

```

class SearchResultsActivity : Activity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_search)
        search_results_recyclerview.layoutManager = LinearLayoutManager(this)
        val query = intent.getStringExtra(QUERY)
        Thread { fetchResults("magic://my.app/search?query=$query") }.start()
    }

    private fun onResultsReceived(locations: List<Location>) {
        search_progress.visibility = View.GONE
    }
}

```

```

        search_results_recyclerview.visibility = View.VISIBLE
        search_results_recyclerview.adapter = Adapter(locations)
    }

    private fun fetchResults(urlWithQuery: String) {
        try {
            val builder = StringBuilder()
            val url = URL(urlWithQuery)
            val connection = url.openConnection() as HttpURLConnection
            var data: Int = connection.inputStream.read()
            while (data != -1) {
                builder.append(data.toChar())
                data = connection.inputStream.read()
            }
            val json = builder.toString()
            val locations = Gson().fromJson<List<Location>>(json,
                object : TypeToken<List<Location>>() {}.type)
                .onResultsReceived(locations)
        } catch (e: IOException) {
            Log.d("MyApp",
                "Something went wrong when fetching results. Probably the fake URL!")
        }
    }

    private class Adapter(private val locations: List<Location>) :
        RecyclerView.Adapter<Adapter.ViewHolder>() {

        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
            return ViewHolder<TextView>(parent.context)
        }

        override fun onBindViewHolder(holder: ViewHolder, position: Int) {
            val (streetAddress, city, country) = locations[position]
            holder.textView.text = "$streetAddress, $city, $country"
        }

        override fun getItemCount(): Int {
            return locations.size
        }

        private class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
            val textView: TextView = itemView as TextView
        }
    }

    companion object {
        val QUERY = "QUERY"
    }
}

```

(Не забудьте добавить этот объект Activity в *AndroidManifest.xml*!)

Довольно много кода! Но что он делает? Ранее мы детально разбирали листинги и подробно разясняли, что делает каждая инструкция или выражение. Но вы постепенно набираетесь опыта и становитесь настоящими разработчиками, поэтому мы отступим от прежнего правила и попробуем объяснить

происходящее здесь так же, как объяснили опытному коллеге, который может просмотреть код или протестировать его.

Здесь определяется класс, наследующий Activity, а значит, он представляет отдельный экран с информацией. Наш класс ожидает получить строку с поисковым запросом из SearchView в BrowseContentActivity.

Согласно определению макета, первоначально Activity отображает вращающийся индикатор ProgressBar в центре экрана, чтобы показать, что приложение выполняет некоторую работу в фоновом режиме. RecyclerView не отображается, содержимое не будет благополучно загружено.

В процессе создания мы берем строку с поисковым запросом и добавляем ее в конец известного URL веб-службы. Для отправки запроса на сервер мы используем стандартные библиотечные классы, такие как URL и HttpURLConnection, а полученную в ответ последовательность байтов преобразуем в строку.

Чтобы не блокировать пользовательский интерфейс, сетевой запрос выполняется в фоновом потоке – мы уже видели, что для получения даже простого ответа иногда требуется несколько секунд (или больше), и было бы нехорошо, если бы пользовательский интерфейс зависал на это время.

Мы знаем, что ответ возвращается в формате JSON, поэтому используем библиотеку Gson для его парсинга и преобразования в коллекцию объектов Location. Если все прошло успешно, мы скрываем ProgressBar и отображаем RecyclerView, после чего передаем экземпляры Location в адаптер и подключаем его. Если сетевой запрос или операция десериализации потерпели неудачу, мы просто выводим ошибку в журнал – в готовом приложении, вероятно, следовало бы вывести сообщение в пользовательском интерфейсе или просто повторить операцию.

Теперь, после заполнения RecyclerView и адаптера, мы должны увидеть список всех объектов Location, полученных в ответ на поисковый запрос, в форме простых строк, содержащих адреса библиотек.

Даже не самый лучший код, написанный на скорую руку, может выглядеть довольно простым, если знать, как используются имеющиеся инструменты.

А теперь посмотрим, как то же самое сделать в iOS.

iOS

В Xcode добавьте в проект новый файл с исходным кодом на Swift и с именем LocationsController. Как вы помните, главная цель нашего пользовательского интерфейса – получить список местоположений библиотек для данной страны. Созданная нами служба имеет конечную точку, выполняющую именно эту функцию. Если вы откроете страницу http://localhost:3000/location?country=<название_страны>, то увидите список библиотек, находящихся в этой стране. Итак, у нас есть параметр country, в котором можно передать название страны.

Контроллер LocationsController можно представить как класс, имеющий единственный метод, выбирающий библиотеки для данной страны. Выразить этот метод можно следующим образом:

```
func fetchLocations(for country: String) -> [Location] {
}
```

Нам также нужно определить структуру `Location`. Мы знаем, что будем получать ответы в формате JSON, поэтому используем структуру и протокол `Codable`, подобно тому, как мы это сделали в отношении структуры `Book`. Используя файл `location.json` в качестве руководства, получаем следующий объект `Location`:

```
struct Location: Codable {
    let streetAddress: String
    let city: String
    let country: String
    let emoji: String
    let hours: String

    private enum CodingKeys: String, CodingKey {
        case streetAddress = "the_address"
        case city
        case country
        case emoji
        case hours
    }
}
```

i Обратите внимание на свойство `streetAddress`. В документе JSON это свойство называется `the_address`. Поскольку имена свойств в структуре `Location` и в файле JSON не совпадают, было добавлено приватное перечисление `CodingKeys`, которое обеспечивает правильное сопоставление имен свойств в JSON и в структуре `Location`. К сожалению, даже если не совпадает имя всего одного свойства, необходимо перечислить имена всех свойств.

При этом `JSONDecoder` может помочь в некоторых случаях, таких как преобразование имен в нотации с символами подчеркивания, используемых в JSON, в имена свойств в верблюжьей нотации структуры, например:

```
let decoder = JSONDecoder()
decoder.keyDecodingStrategy = .convertFromSnakeCase
...
```

Если теперь использовать только что объявленный метод `fetchLocations(for:)`, он прекрасно будет работать с быстрым соединением. Но он действует синхронно, что может стать проблемой при получении большого объема данных или с низкоскоростным соединением. То есть если вызвать его из основного потока, пользователь будет вынужден сидеть и ждать завершения процесса, прежде чем что-то произойдет; наше приложение будет выглядеть зависшим.

Это не самое лучшее решение.

К счастью, эту проблему легко исправить с помощью замыканий. Типичный шаблон выполнения сетевых операций в Swift заключается в использовании обработчика, который выполняется по завершении операции. Давайте изменим сигнатуру метода так, чтобы он передавал возвращаемый массив `[Location]` в обработчик события завершения операции. Также добавим обработчик ошибок на случай, если операция завершится с ошибкой. Вот как теперь должен выглядеть метод:

```
func fetchLocations(for country: String,
                   completionHandler: @escaping ([Location]) -> (),
```

```

        errorHandler: @escaping (Error?) -> () {
    }

```

А после добавления структуры Location содержимое файла должно выглядеть так:

```

import Foundation

class LocationsController {
    func fetchLocations(for country: String,
                       completionHandler: @escaping ([[Location]]) -> (),
                       errorHandler: @escaping (Error?) ->
                       ()) {
    }
}

struct Location: Codable {
    let streetAddress: String
    let city: String
    let country: String
    let emoji: String
    let hours: String

    private enum CodingKeys: String, CodingKey {
        case streetAddress = "street_address"
        case city
        case country
        case emoji
        case hours
    }
}

```

Библиотека *URLSession*

Вернемся к нашему методу `fetchLocations(for:completionHandler:errorHandler:)` и добавим его реализацию. При этом мы используем библиотеку `URLSession`, а конкретнее – класс `URLSessionDataTask` из нее.

Организовать сетевые взаимодействия можно несколькими способами. Библиотека `URLSession`, в частности, разбивает взаимодействия на три типа с тремя разными реализациями интерфейса `URLSessionTask` (фактически представляющего единичный запрос), которые упрощают работу с ними: `URLSessionDataTask`, `URLSessionDownloadTask` и `URLSessionUploadTask`.

Каждый из этих реализаций обеспечивает уникальную функциональность для своей цели. Тип `URLSessionDataTask`, который мы будем использовать, предназначен для извлечения данных из URL – это именно то, что нам нужно. Более подробно об этих объектах рассказывается в главе 9.

Посмотрим, как можно использовать `URLSessionDataTask` в нашем коде:

```

let url = URL(string: "http://localhost:3000/locations?country=\(country)")!
let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
}
task.resume()

```

Здесь сначала создается URL с адресом нашей службы поиска. В настоящее время служба доступна по адресу `http://localhost:3000/location`, если вы следовали за примером создания службы Node выше в этой главе, в противном случае используйте любой другой URL, уместный здесь. Инициализатор `URL(string:)` генерирует объект URL, поддерживающий пустое значение, то есть экземпляр типа `URL?`. Мы принудительно распаковываем с использованием оператора `!`, потому что точно знаем, что он не будет содержать пустого значения. (Но даже если это произойдет, для нас предпочтительнее узнать об этом, получив аварийное завершение приложения!)

Далее используется свойство `shared` класса `URLSession`, чтобы получить доступ к разделяемому сеансу и сгенерировать экземпляр `URLSessionDataTask` вызовом метода `dataTask(with:completionHandler:)` – мы нигде не создаем экземпляры `URLSessionDataTask` непосредственно. Полученный экземпляр сохраняется в переменной `task`.

Обработчик завершения имеет три параметра: `data`, `response` и `error`. Параметр `data` имеет тип `Any?` и содержит данные, полученные в ответе. Параметр `response` содержит экземпляр ответа типа `URLResponse`, а параметр `error`, содержащий экземпляр `Error?`, используется для проверки успешного выполнения запроса.

Прямо сейчас наш метод `fetchLocations(for:completionHandler:errorHandler)` уже сможет послать запрос службе, но, поскольку замыкание с обработчиком завершения имеет пустое тело, ничего не произойдет. Давайте исправим это. Вот как должно выглядеть полное тело нашего метода:

```
func fetchLocations(for country: String,
                  completionHandler: @escaping ([Location]) -> (),
                  errorHandler: @escaping (Error?) -> ()) {
    let url = URL(string: "http://localhost:3000/locations?country=\(country)")!
    let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
        if let error = error {
            // Возникла ошибка на стороне сервера
            errorHandler(error)
            return
        }

        guard let response = response as? HTTPURLResponse,
              response.statusCode < 300 else {
            // Возникла ошибка на стороне клиента
            errorHandler(nil)
            return
        }

        guard let data = data else {
            // Получены недействительные данные
            errorHandler(nil)
            return
        }

        // Преобразовать данные в массив [Location]
    }
    task.resume()
}
```


В теле `dataTask(with:completionHandler:)` мы передаем замыкание, которое:

- 1) проверяет наличие ошибки на стороне сервера, анализируя параметр `error`. Если произошла ошибка, вызывается замыкание `errorHandler`, и затем управление возвращается приложению;
- 2) если на стороне сервера никаких ошибок не возникло (то есть `error` содержит значение `nil`), проверяется код состояния HTTP-ответа – он должен быть меньше 300. Если это не так, снова вызывается `errorHandler`, но этот раз, так как `error` содержит значение `nil`, мы передаем обработчику `nil`;
- 3) если сетевая операция завершилась успешно, проверяются полученные данные. Объект `data` должен быть непустым. Если данные отсутствуют, снова вызывается `errorHandler` с параметром `nil`.

Теперь возьмем этот объект с данными и преобразуем содержащийся в нем документ JSON в массив объектов `Location`. После добавления необходимого кода получаем следующий файл `LocationsController.swift`:

```
import Foundation

class LocationsController {
    func fetchLocations(for country: String,
                      completionHandler: @escaping ([Location]) -> (),
                      errorHandler: @escaping (Error?) -> ()) {
        let url = URL(string: "http://localhost:3000/locations?country=\(country)")!
        let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
            if let error = error {
                errorHandler(error)
                return
            }

            guard let response = response as? HTTPURLResponse,
                  response.statusCode < 300 else {
                errorHandler(nil)
                return
            }

            guard let data = data, let locations =
                try? JSONDecoder().decode([Location].self, from: data) else {
                errorHandler(nil)
                return
            }

            // Вызвать обработчик успешного завершения операции
            completionHandler(locations)
        }
        task.resume()
    }
}

struct Location: Codable {
    let streetAddress: String
    let city: String
    let country: String
    let emoji: String
}
```

```

    let hours: String
}

```

Отлично! Теперь у нас есть действующий сетевой клиент, который можно использовать для поиска библиотек. Мы *почти* закончили, осталось только задействовать клиента в пользовательском интерфейсе. Вернемся к `LocationsTableViewController` и добавим следующий код в наш метод `searchBarTextDidEndEditing(_:)` в классе `UISearchBarDelegate`:

```

func searchBarTextDidEndEditing(_ searchBar: UISearchBar) {
    let country = searchBar.text ?? ""
    locationsController.fetchLocations(for: country, completionHandler:
    { (locations) in
        DispatchQueue.main.async {
            self.locations = locations
            self.tableView.reloadData()
        }
    }) { (error) in
        DispatchQueue.main.async {
            self.locations = []
            self.tableView.reloadData()
        }
    }
}

```

Рассмотрим этот код подробнее.

Сначала сохраняем в переменной `country` строку из свойства `text` поля ввода. Свойство `text` имеет тип `String`, а значит, может содержать `nil`; мы исправим эту проблему с помощью оператора `??`, который означает следующее: «если операнд слева от оператора равен `nil`, использовать операнд справа». В данном случае мы присваиваем пустую строку. Затем вызываем метод `fetchLocations` нового свойства, добавленного в класс, которое содержит объект `LocationsController`. Мы передаем методу обработчик `completionHandler` с замыканием `errorHandler`, который сохранит полученные результаты в локальном свойстве и обновит табличное представление, содержащееся в этом представлении.

Обратите внимание, что для обновления представления мы посылаем из замыканий запрос в основной поток. Мы не знаем, в каком потоке будет выполняться сетевой вызов, но все обновления пользовательского интерфейса должны производиться только в основном потоке.

Нам осталось урегулировать еще одну проблему в этом классе, прежде чем мы сможем его протестировать. Нам нужно, чтобы табличное представление получило данные из массива `locations`, который в данный момент хранится в классе. К счастью, если вы помните, этот класс доступен в табличном представлении в виде свойства `dataSource`, поэтому, чтобы задействовать массив `locations`, достаточно внести небольшие изменения в методы протокола `UITableViewDataSource`, как показано ниже:

```

override func tableView(_ tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
    return locations.count
}

```

```

override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // Получить ячейку из пула свободных ячеек
    let cell = tableView.dequeueReusableCell(withIdentifier: "LocationCell",
                                           for: indexPath)

    // Найти местоположение, соответствующее заполняемой ячейке
    let location = locations[indexPath.row]

    // Добавить оформление ячейки
    cell.textLabel?.text = location.emoji
    cell.detailTextLabel?.text =
"\(location.streetAddress)\n\((location.city), \((location.country))\nHours: \((location.
hours)"
    return cell
}

```

Окончательная версия класса `LocationsViewController` должна выглядеть так:

```

import UIKit

class LocationsTableViewController: UITableViewController {

    let locationsController = LocationsController()
    var locations: [Location] = []

    override func viewDidLoad() {
        super.viewDidLoad()

        let searchController = UISearchController(searchResultsController: nil)
        searchController.searchBar.delegate = self
        searchController.searchBar.placeholder = "Search Locations by Country"
        searchController.obscuritiesBackgroundDuringPresentation = false
        definesPresentationContext = true

        navigationItem.searchController = searchController
        navigationItem.hidesSearchBarWhenScrolling = false
    }

    // Источник данных для табличного представления
    override func tableView(_ tableView: UITableView,
                            numberOfRowsInSection section: Int) -> Int {
        return locations.count
    }

    override func tableView(_ tableView: UITableView,
                            cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        // Получить ячейку из пула свободных ячеек
        let cell = tableView.dequeueReusableCell(withIdentifier: "LocationCell",
                                               for: indexPath)

        // Найти местоположение, соответствующее заполняемой ячейке
        let location = locations[indexPath.row]

        // Добавить оформление ячейки
        cell.textLabel?.text =
"\(location.streetAddress)\n\((location.city), \((location.country))\nHours: \((location.
hours)"

```

```

        cell.detailTextLabel?.text = location.emoji
    }
    return cell
}
}

extension LocationsTableViewController: UISearchBarDelegate {
    func searchBarTextDidEndEditing(_ searchBar: UISearchBar) {
        let country = searchBar.text ?? ""
        guard country != "" else {
            self.locations = []
            self.tableView.reloadData()
            return
        }
        locationsController.fetchLocations(for: country, completionHandler:
        { (locations) in
            DispatchQueue.main.async {
                self.locations = locations
                self.tableView.reloadData()
            }
        }) { (error) in
            DispatchQueue.main.async {
                self.locations = []
                self.tableView.reloadData()
            }
        }
    }

    func searchBarCancelButtonClicked(_ searchBar: UISearchBar) {
        self.locations = []
        self.tableView.reloadData()
    }
}
}

```

Соберите и запустите приложение, щелкните на кнопке с лупой, выполните поиск по названию страны, и... на экране ничего не появится. Но почему?

Взгляните на панель консоли, там вы увидите сообщение об ошибке:

```

App Transport Security(("App Transport Security")) has blocked a cleartext HTTP (http://)
resource load since it is insecure. Temporary exceptions can be configured via your
app's(("Info.plist file")) Info.plist file.

```

Причина в том, что политика безопасности, устанавливаемая компанией Apple, требует использовать URL со схемой `https://`, тогда как мы использовали схему `http://`. Эту проблему легко исправить, и фактически решение приводится в самом сообщении об ошибке: нужно добавить исключение в файл *Info.plist* приложения. В этом файле находятся конфигурационные параметры приложения.

Чтобы добавить исключение для своего домена, откройте *Info.plist* и в последней строке щелкните на кнопке + для добавления нового свойства. В открывающемся списке выберите **App Transport Security Settings** (Настройки безопасности транспорта приложения). Затем добавьте новое дочернее свойство **Allow Arbitrary Loads** (Разрешить загрузку из произвольного места)

и установите его в значение **YES**. Это позволит приложению использовать любые незащищенные URL. Если теперь запустить приложение и выполнить поиск по названию страны France, вы должны увидеть в симуляторе результаты, как показано на рис. 20.1.

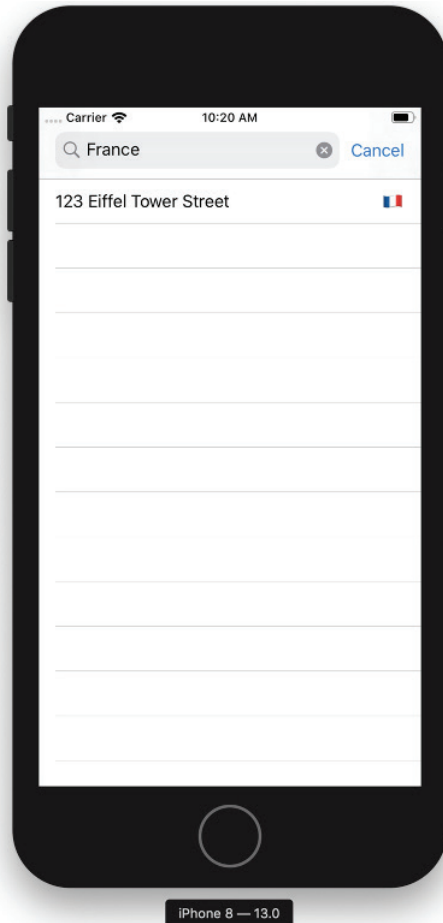


Рис. 20.1 ❖ Да здравствует симулятор!

ⓘ Имейте в виду, что, выполнив настройку в *Info.plist*, разрешающую загрузку данных из произвольных URL, мы сделали приложение менее безопасным. В данном примере такой прием вполне оправдан. Но вообще старайтесь для связи со службами использовать безопасный протокол HTTPS. Не выпускайте такие приложения в дикую природу, если не уверены в своих действиях. Другой возможный вариант решения проблемы в нашем примере – присвоить **YES** параметру **NSAllowsLocalNetworking** в файле *Info.plist*, чтобы разрешить загрузку локальных файлов.

Если вы решитесь начать распространять приложение с включенным разрешением загрузки из произвольных URL, будьте готовы к тому, что вам придется обосновать это решение перед специалистами из App Review!

Мы можем сделать еще кое-что, чтобы сообщить пользователю, что выполняется сетевой вызов: присвоить свойству `UIApplication.shared.isNetworkActivityIndicatorVisible` значение `true` и `false`, чтобы изменить отображение индикатора сетевой активности в строке состояния и отметить начало и конец сетевой операции. Включить индикатор можно перед запуском `URLSessionDataTask`, примерно так:

```
func fetchLocations(for country: String,
                  completionHandler: @escaping ([[Location]] -> ()),
                  errorHandler: @escaping (Error?) -> ()) {
    ...
    DispatchQueue.main.async {
        UIApplication.shared.isNetworkActivityIndicatorVisible = true
    }
    task.resume()
}
```

Затем внутри обработчика события завершения операции можно присвоить свойству `UIApplication.shared.isNetworkActivityIndicatorVisible` значение `false`.

На этом мы завершаем обсуждение поддержки сетевых операций в iOS.

Что мы узнали

Мы увидели, как организовать взаимодействие с веб-службами из Android и iOS. Подходы, используемые в обеих системах, имеют много общего, но в каждой имеются свои уникальные особенности. В этой главе мы узнали, как:

- 1) добавить новый экран, обслуживающий поиск;
- 2) создать сетевого клиента для взаимодействия с удаленной веб-службой;
- 3) связать пользовательский интерфейс с сетевым клиентом.

Android и iOS обладают намного более широкими возможностями, и здесь мы увидели лишь самую верхушку айсберга. Загляните в главу 9, где приводится более подробная информация и демонстрируются примеры, как добавить поддержку сетевых взаимодействий в ваше приложение.

Предметный указатель

Символы

- .actionSheet, 177
- .alert, 177
- .compact, размерный класс, 31
- .regular, размерный класс, 36

А

Activity, класс

- и Fragment, 25
- начальный контроллер
- пользовательского интерфейса, 21
- передача информации в, 23
- регистрация в ApplicationManifest.xml, 21
- рекомендованный стиль навигации, 20
- этапы жизненного цикла, 27

Android

- версии, 239
- данные приложений
 - динамические данные
 - в представлениях списков, 291
 - заполнение представления списка, 278
 - кнопки, 256
 - организация, 274
 - переключение слоя данных на
 - использование JSON, 301
 - списки, 272
- добавление кнопки в приложение, 319
- конкурентное (многопоточное)
- выполнение, 140
 - завершение потока выполнения, 145
 - запуск заданий в фоновом потоке, 141
 - передача результатов из фонового
 - потока в главный, 144
 - создание потока выполнения, 140
- настройка окружения разработки, 229
- обратная связь
 - изменение строки состояния, 175
 - с использованием системных
 - инструментов, 172
 - сообщения Dialog, 174
 - сообщения Snackbar, 173
 - сообщения Toast, 173
- оформление представлений, 255
- передача сообщений
 - LocalBroadcastManager, 90
 - обратные вызовы, 91
 - подписчикам, 95

- получение и обработка, 96
- реализация своей шины
- сообщений, 98

пользовательские компоненты

- использование, 66, 67
- обзор, 60
- создание, 61

пользовательский ввод

- касания, 74
- клавиатура, 78
- сложные жесты, 81
- щелчки, 74

предпочтения пользователя

- SharedPreferences, 182
- в многопользовательских
- приложениях, 184
- сохранение, 183
- чтение, 184

представления

- View, базовый класс, 43
- вложение друг в друга, 49
- доступ к макетам, 47
- изменение состояния, 50
- развертывание макетов, 48
- создание новых, 44

расширения, 207

сериализация и транспорты

- XML, JSON и Protocol Buffers, 192
- сериализация и десериализация
- экземпляров объектов, 192

сетевые взаимодействия

- взаимодействия с конечными
- точками, 343
- выполнение запросов, 353
- дополнительные ресурсы, 160
- загрузка двоичных файлов, 159
- загрузка текстовых файлов с
- удаленного сервера, 156
- многопоточность, 156
- отображение результатов поиска, 345
- создание запроса HTTP POST, 157
- сторонние библиотеки, 156

создание экрана, 247

тестирование

- интеграционные тесты, 220
- модульные тесты, 217
- типы тестов, 213

- файлы
 - java.io, 112
 - копирование файлов, 118
 - обзор, 111
 - определение характеристик файлов, 112
 - получение ссылки на файл, 112
 - создание каталогов, 112
 - создание файлов, 112
 - чтение и запись в файлы, 113
 - хранение данных
 - SQLite, 126
 - выбор базы данных, 132
 - добавление вывода подробных данных в приложение, 309
 - добавление кнопки в приложение, 319
 - запись данных, 129
 - запись книг в хранилище, 322
 - соединение с базой данных, 127
 - создание таблицы или хранимого объекта, 128
 - сохранение книг в закладках, 340
 - чтение данных, 130
 - экран запуска, 247
 - эмуляторы, 239
 - Android, команды и методы
 - afterTextChanged, 79
 - android:imeOptions, 80
 - beforeTextChanged, 79
 - compileStatement, 132
 - file.createNewFile(), 112
 - file.mkdir(), 112
 - file.mkdirs(), 112
 - getBytes(), 113
 - getFilesDir(), 112
 - getReadableDatabase, 127
 - getSharedPreferences(String fileName, Context.MODE_PRIVATE);, 183
 - getWritableDatabase, 127
 - InputStream.read, 119
 - onCreate, 20, 27, 128
 - onDestroy, 29
 - onDown, 82
 - onFling, 82
 - onFocusChanged(boolean, int, Rect), 84
 - onKeyDown(int, KeyEvent), 83
 - onKeyUp(int, KeyEvent), 83
 - onLayout, 63
 - onLongPress, 82
 - onMeasure, 63
 - onPause, 29
 - onResume, 29
 - onShowPress, 82
 - onSingleTapUp, 82
 - onStart, 29
 - onStop, 29
 - onTextChanged, 79
 - onTouchEvent, 81
 - onTouchEvent(MotionEvent), 84
 - onTrackballEvent(MotionEvent), 83
 - onUpgrade, 127, 128
 - registerReceiver, 96
 - removeTextChangedListener, 79
 - sendBroadcast, 96
 - TextView.setImeOptions, 80
 - Android Bundles, 248
 - Android Native Development Kit (Android NDK), 227
 - Android Open Source Project (AOSP), 227
 - Android Studio
 - выбор темы оформления, 232
 - выбор типа установки, 231
 - диалог приветствия, 230
 - загрузка, 229
 - импорт настроек, 230
 - и расширения, 208
 - поддержка рефакторинга, 306
 - подтверждение параметров установки, 233
 - преимущества, 229
 - создание нового проекта, 238
 - AndroidX, 30, 216
 - Apache Commons, библиотека для Java, 119
 - Apple Developer Program, 235
 - Apple File System (APFS), 124
 - Application, класс, 20
 - Auto Layout, 56, 317
- ## В
- bounds, свойство, 51
- ## С
- Cocoa Touch
 - делегаты, 103
 - добавление контроллера представления, 251
 - добавление нового класса, 348
 - создание нового класса, 285
 - уведомления, 104
 - Codable, протокол, 201, 205, 306
 - Core Data
 - NSAllowsArbitraryLoads, 170
 - NSAppTransportSecurity, 170
 - NSCoding, протокол, 186
 - NSExceptionDomains, 170
 - NSFetchedResultsController, 138

NSKeyedUnarchiver, 187
 @NSManaged, атрибуты, 136
 NSManagedObject, 134
 NSManagedObjectContext, 134
 NSManagedObjectModel, 133
 NSPersistentStoreCoordinator, 133
 NSPersistentStoreDescription, 134
 добавление поддержки, 331
 добавление сущностей, 332
 инициализация стека, 333
 настройка стека, 133
 определение объектов в существующем проекте, 135
 переключение с использованием файла JSON, 335
 преимущества, 133, 138
 сравнение с низкоуровневым интерфейсом SQLite, 133

D

Dart, язык программирования, 228
 Dialog, класс, 173
 DispatchQueue, 150

E

Espresso, библиотека тестирования пользовательского интерфейса, 216
 EventBus, 98
 Express, 350

F

FileManager, класс (iOS), 120
 Flutter, 228
 Fragment, класс
 и Activity, 25
 этапы жизненного цикла, 27

G

Grand Central Dispatch (GCD), 53, 151
 Gson, 302
 Gson, библиотека, 194

I

IME (Input Method Editor – редактор методов ввода), 80
 Info.plist, файл, 31, 33, 161, 170
 Interface Builder, 56
 iOS
 Auto Layout, 317
 данные приложений
 динамические данные
 в представлениях списков, 295

 заполнение представления списка, 285
 кнопки, 265, 270
 организация, 277
 переключение слоя данных на использование JSON, 306
 списки, 273
 добавление кнопки в приложение, 320
 касания, 84
 конкурентное (многопоточное) выполнение
 доступные варианты, 151
 запуск задачи в фоновом потоке, 151
 передача результатов из фонового потока в главный, 152
 модульные тесты, 222
 настройка окружения разработки, 235
 обратная связь
 изменение строки состояния, 179
 с использованием системных инструментов, 177
 тактильная, 180
 текстовые поля
 в предупреждениях, 180
 оформление представлений, 265
 передача сообщений
 Key-Value Observation (KVO), 109
 замыкания вместо селекторов, 107
 обратные вызовы, 98
 отмена подписки, 108
 подписчикам, 104
 получение и обработка, 106
 получение уведомлений от конкретного издателя, 108
 пользовательские компоненты
 использование, 70
 обзор, 68
 создание, 68
 пользовательский ввод
 клавиатура, 85
 программный интерфейс сенсорных событий, 89
 сложные жесты, 87
 предпочтения пользователя
 UserDefaults, 185
 в многопользовательских приложениях, 189
 сохранение, 185
 чтение, 188
 представления
 Auto Layout, 317
 Interface Builder, 56
 автоматическое размещение, 57
 вложение друг в друга, 53

- в многопоточном окружении, 53
- изменение позиции, 58
- изменение состояния, 57
- ограничения, 54
- прозрачность, 57
- создание новых, 51
- сокрытие, 57
- цвет фона, 57
- экземпляры UIView, 51
- программирование Keychain, 189
- расширения, 209
- резервное копирование в iCloud и iTunes, 124
- сериализация и транспорты
 - XML, JSON и списки свойств, 200
 - сериализация и десериализация экземпляров объектов, 200
- сетевые взаимодействия
 - безопасность передаваемых данных, 170
 - взаимодействия с конечными точками, 347
 - выполнение запросов, 356
 - загрузка двоичных файлов, 166
 - загрузка текстовых файлов с удаленного сервера, 161
 - отображение результатов поиска, 347
 - приостановка и возобновление загрузки, 169
 - создание запроса HTTP POST, 162
 - фоновые потоки и обновление пользовательского интерфейса, 170
- создание начального экрана, 249
- файлы
 - URL и строки, 124
 - атрибуты файлов, 121
 - доступ к каталогам, 120
 - копирование файлов, 123
 - обзор, 119
 - определение характеристик файла, 119
 - чтение и запись в файлы, 122
- хранение данных
 - Core Data, 133
 - добавление вывода подробных данных в приложение, 314
 - добавление кнопки в приложение, 320
 - запись данных, 136
 - запись книг в хранилище, 331
 - создание таблиц и объектов, 135
 - чтение данных, 137
- экран запуска, 249

- iOS, методы
 - application(_:didFinishLaunchingWithOptions:), 134
 - didReceiveMemoryWarning, 40
 - handleTap(_:), 85
 - init(coder:), 38
 - makeKeyAndVisible(), 33
 - performSegue(withIdentifier:sender:), 36
 - push(_:animated:completion:), 35
 - require(toFail:), 88
 - setUpView, 69
 - shouldPerformSegue(withIdentifier:sender:), 37
 - showDetail(_:sender:), 41
 - show(_:sender:), 34, 51
 - textFieldDidChange(_:), 86
 - urlSession(_:downloadTask:didWriteData:), 168
 - urlSession(_:task:didCompleteWithError:), 168
 - viewDidAppear, 39
 - viewDidDisappear, 39
 - viewDidLoad, 39
 - viewDidLoad(), 84
 - viewWillAppear, 39
 - viewWillDisappear, 39
 - write(to:atomically:encoding:), 123

J

- Java
 - Apache Commons, библиотека, 119
 - данные приложений
 - динамические данные
 - в представлениях списков, 291
 - заполнение представления списка, 278
 - переключение слоя данных на использование JSON, 301
 - конкурентное (многопоточное) выполнение
 - завершение потока выполнения, 145
 - запуск заданий в фоновом потоке, 141
 - передача результатов из фонового потока в главный, 144
 - создание потока выполнения, 140
 - обратная связь
 - изменение строки состояния, 175
 - сообщения Dialog, 174
 - сообщения Snackbar, 173
 - сообщения Toast, 173
 - оформление представлений, 260
 - передача сообщений
 - обратные вызовы, 91
 - подписчикам, 96

пользовательские компоненты
 использование, 67
 создание, 61
 пользовательский ввод, 74
 клавиатура, 79
 сложные жесты, 82
 предпочтения пользователя
 в многопользовательских приложениях, 184
 сохранение, 183
 чтение, 184
 представления
 доступ к макетам, 47
 развертывание макетов, 48
 создание нового представления, 44
 сериализация, 196
 сериализация и транспорты
 сериализация и десериализация экземпляров объектов, 192
 сетевые взаимодействия
 взаимодействия с конечными точками, 346
 выполнение запросов, 353
 загрузка двоичных файлов, 159
 загрузка текстовых файлов с удаленного сервера, 156
 создание запроса HTTP POST, 157
 тестирование, 214
 файлы
 копирование файлов, 118
 определение характеристик файлов, 112
 чтение и запись в файлы, 113
 хранение данных
 добавление вывода подробных данных в приложение, 311
 запись данных, 129
 запись книг в хранилище, 323
 соединение с базой данных, 127
 создание таблицы или хранимого объекта, 128
 сохранение книг в закладках, 340
 чтение данных, 130
 Java, сериализация, 197
 Java Development Kit (JDK), 229
 java.nio.file, пакет, 111
 Jetpack, библиотека, за и против, 30
 JSON, сериализация
 Android, 193
 iOS, 200
 JSON, файл с местоположениями библиотек, 351
 jsonplaceholder.typicode.com, служба, 157

К

Keychain (iOS), 185
 KitKat, 239
 Kotlin
 данные приложений
 динамические данные
 в представлениях списков, 291
 заполнение представления списка, 278
 переключение слоя данных на использование JSON, 301
 конкурентное (многопоточное) выполнение
 завершение потока выполнения, 145
 запуск заданий в фоновом потоке, 141
 создание потока выполнения, 140
 обратная связь
 изменение строки состояния, 175
 сообщения Dialog, 174
 сообщения Snackbar, 173
 сообщения Toast, 173
 оформление представлений, 260
 передача сообщений
 обратные вызовы, 91
 подписчикам, 96
 пользовательские компоненты
 использование, 67
 создание, 61
 предпочтения пользователя
 в многопользовательских приложениях, 184
 сохранение, 183
 чтение, 184
 представления
 доступ к макетам, 47
 развертывание макетов, 48
 создание нового представления, 44
 расширения, 208
 сериализация и транспорты
 сериализация и десериализация экземпляров объектов, 192
 сетевые взаимодействия
 взаимодействия с конечными точками, 346
 выполнение запросов, 354
 загрузка двоичных файлов, 159
 загрузка текстовых файлов с удаленного сервера, 156
 создание запроса HTTP POST, 157
 тестирование, 214
 файлы
 копирование файлов, 118
 определение характеристик файлов, 112

- чтение и запись в файлы, 113
- хранение данных
 - добавление вывода подробных данных в приложение, 311
 - запись данных, 129
 - запись книг в хранилище, 324
 - соединение с базой данных, 127
 - создание таблицы или хранимого объекта, 128
 - чтение данных, 130

L

- LocalBroadcastManager, 90
- Lollipop, 239

M

- modalTransitionStyle, 35
- Multi-Touch, 84

N

- Node, 350
- Notification, класс, 175

O

- Objective C, 107, 210
- Objective-C, 100
- Open Step, 205
- org.json, пакет, 193
- org.xmlpull, пакет, 195
- Otto, 98

P

- PhoneGap, 228

R

- React Native, 228
- Robolectric, библиотека, 216
- Room, библиотека, 132
- rootViewController, свойство, 31

S

- SharedPreferences API, 182
- Snackbar, класс, 173
- SQLite, 126, 136
- SwiftUI, библиотека, 59

T

- TextWatcher, интерфейс, 79
- Toast, класс, 172

U

- UIKit
 - UIAlertController, 178

- UIAlertController, класс, 177
- UIMainStoryboardFile, 31
- UINavigationController, 41
- UINavigationController, класс, 266
- UISplitViewController, 41
- UITabBarController, 41
- UITapGestureRecognizer, 85
- UITextField, 85
- UITextView, 85
- UIView, 51, 68
- UIViewController, 30, 32, 51, 178
- UIWindow, 31, 51
 - последователь, 59
- URLSession, класс
 - URLSessionDataDelegate, 170
 - URLSessionDownloadDelegate, 167, 170
 - URLSessionTaskDelegate, 170
 - делегаты и методы делегатов, 170
 - основные возможности, 161

X**Xcode**

- Apple Developer Program, 235
- инспектор атрибутов, 250
- редактор раскладки, 249
- симулятор, 244
- создание нового проекта, 242
- установка и настройка, 235
- XML, сериализация, 201
- XML Interface Builder (XIB), 52

B

- Венгерская нотация, 194

G

- Графические ресурсы и разрешение экрана, 248

D**Данные приложений****Android**

- заполнение представления списка, 278
- кнопки, 256
- организация, 274
- списки, 272

iOS

- заполнение представления списка, 285
- кнопки, 265, 270
- организация, 277
- списки, 273

- динамические данные в представлениях списков

- Android, 291

- iOS, 295
- модель слоя JSON, пример, 298
- обзор задач, 255
- переключение слоя данных на использование JSON
 - Android, 301
 - iOS, 306
- Двоичные файлы, загрузка
 - Android, 159
 - iOS, 166
- Делегаты, 103

Е

- Единая база кода, 228

Ж

- Жесты
 - Android, 81
 - iOS, 87

З

- Замыкания, 98
 - вместо селекторов, 107

И

- Изменение строки состояния
 - Android, 175
 - iOS, 179
- Инспектор атрибутов, 250
- Инструментированное тестирование, 216
- Интеграционные тесты, обзор, 214

К

- Касания
 - Android, 74
 - iOS, 84
- Качество обслуживания (QoS), 151
- Клавиатура
 - Android, 78
 - iOS, 85
- Конкурентное (многопоточное) выполнение
 - Android
 - завершение потока выполнения, 145
 - запуск заданий в фоновом потоке, 141
 - передача результатов из фонового потока в главный, 144
 - создание потока выполнения, 140
 - iOS
 - доступные варианты, 151
 - запуск задачи в фоновом потоке, 151
 - передача результатов из фонового потока в главный, 152

- обзор задач, 140
- определение, 140
- Контейнер данных приложения, 119
- Контейнер пакета приложения, 119
- Контроллеры пользовательского интерфейса
 - Android, этапы жизненного цикла, 27
 - getIntent, метод, 24
 - iOS
 - UIKit, 30
 - создание, 31
 - этапы жизненного цикла, 38
 - изменение активного, 23, 34
 - обновление Fragment, 26
 - рекомендованный стиль навигации, 20
 - создание, 21, 31
 - этапы жизненного цикла, 27, 38
- Кросс-платформенная разработка, 227

М

- Межпроцессные взаимодействия (IPC), 143
- Механизмы обратной связи, 177
- Модель–представление–контроллер (Model-View-Controller, MVC), 246
- Модульные тесты Android, 213

Н

- Наблюдение за изменением значений свойств (Key-Value Observation, KVO), 109
- Настройка окружения разработки
 - Android, 229
 - iOS, 235
 - сравнение нативных и кросс-платформенных инструментов, 227
- Нативность, определение, 227
- Необязательные значения (Swift), 268

О

- Обратная связь
 - Android
 - изменение строки состояния, 175
 - с использованием системных инструментов, 172
 - сообщения Dialog, 174
 - сообщения Snackbar, 173
 - сообщения Toast, 173
 - iOS
 - изменение строки состояния, 179
 - с использованием системных инструментов, 177
 - тактильная, 180

- текстовые поля
 - в предупреждениях, 180
- обзор задач, 172
- Обратные вызовы
 - Android, 91
 - iOS
 - делегаты, 103
 - закрывания, 98
 - экранированные и неэкранированные
 - закрывания, 99
- Объектно-реляционные отображения, 132
- Ограничения, 54
- Оформление представлений
 - Android, 255
 - iOS, 265
- П**
- Передача сообщений
 - Android
 - обратные вызовы, 91
 - подписчикам, 95
 - получение и обработка, 96
 - LocalBroadcastManager, 90
 - iOS
 - Key-Value Observation (KVO), 109
 - закрывания вместо селекторов, 107
 - обратные вызовы, 98
 - отмена подписки, 108
 - подписчикам, 104
 - получение и обработка, 106
 - получение уведомлений от конкретного издателя, 108
 - обзор задач, 90
 - шаблоны и системы, 90
- Переходы, 36, 273
- Подписчики, передача сообщений
 - Android, 95
 - iOS, 104
- Пользовательские компоненты
 - Android
 - использование, 66, 67
 - обзор, 60
 - создание, 61
 - iOS
 - использование, 70
 - обзор, 68
 - создание, 68
 - применение, 60
- Пользовательский ввод
 - iOS
 - касания, 84
 - клавиатура, 85
 - программный интерфейс сенсорных событий, 89
 - сложные жесты, 87
 - клавиатура, 78
 - обзор, 73
 - сложные жесты, 81
 - формы, 73
- Предпочтения пользователя
 - Android
 - SharedPreferences, 182
 - в многопользовательских приложениях, 184
 - сохранение, 183
 - чтение, 184
 - iOS
 - UserDefaults, 185
 - в многопользовательских приложениях, 189
 - сохранение, 185
 - чтение, 188
 - обзор задач, 182
- Представления
 - Android
 - вложение друг в друга, 49
 - доступ к макетам, 47
 - изменение состояния, 50
 - развертывание макетов, 48
 - создание новых, 44
 - View, базовый класс, 43
 - iOS
 - Interface Builder, 56
 - автоматическое размещение, 57
 - вложение друг в друга, 53
 - в многопоточном окружении, 53
 - изменение позиции, 58
 - изменение состояния, 57
 - ограничения, 54
 - прозрачность, 57
 - создание новых, 51
 - сокрытие, 57
 - цвет фона, 57
 - экземпляры UIView, 51
 - назначение, 43
 - обзор задач, 43
- Простейшая анимация, 59
- Протокол доступа к объектам (Simple Object Access Protocol, SOAP), 202
- Р**
- Раскадровки
 - выбор главной раскадровки, 251
 - и класс UIViewController, 32
 - контроллеры навигации, 266

- переходы, 36, 273
- создание контроллеров
- пользовательского интерфейса, 38
- создание новых представлений, 52
- создание списков данных, 274

Расширения

- Android, 207
- iOS, 209
- обзор задач, 207

С

Сериализация и транспорты

- Android
 - XML, JSON и Protocol Buffers, 192
 - сериализация и десериализация экземпляров объектов, 192
- iOS

- XML, JSON и списки свойств, 200
- сериализация и десериализация экземпляров объектов, 200

Сетевые взаимодействия

- Android
 - взаимодействия с конечными точками, 343
 - выполнение запросов, 353
 - дополнительные ресурсы, 160
 - загрузка двоичных файлов, 159
 - загрузка текстовых файлов с удаленного сервера, 156
 - многопоточность, 156
 - отображение результатов поиска, 345
 - создание запроса HTTP POST, 157
 - сторонние библиотеки, 156

- iOS
 - безопасность передаваемых данных, 170
 - взаимодействия с конечными точками, 347
 - выполнение запросов, 356
 - загрузка двоичных файлов, 166
 - загрузка текстовых файлов с удаленного сервера, 161
 - отображение результатов поиска, 347
 - приостановка и возобновление загрузки, 169
 - создание запроса HTTP POST, 162
 - фоновые потоки и обновление пользовательского интерфейса, 170
 - запросы и ответы HTTP, 155
 - обзор задач, 155
 - создание службы поиска, подходы, 350
 - установка Node и Express, 350

- Симуляторы, 244

- Системы управления базами данных, 126

- Сквозные тесты, 214

Сложные жесты

- Android, 81
- iOS, 87

Создание приложений

- Android Studio, 238
- Xcode, 242
- архитектура, 246

Списки

- Android, 272
- iOS, 273
- динамические данные в представлениях списков
 - Android, 291
 - iOS, 295

- Списки свойств, 205

Т

- Тактильная обратная связь (iOS), 180

Текстовые файлы

- Android, 156
- iOS, 161

Тестирование

- Android
 - интеграционные тесты, 220
 - модульные тесты, 217
 - типы тестов, 213
- iOS, модульные тесты, 222
- обзор задач, 213
- преимущества, 213

- Типы с поддержкой null, 268

У

- Уведомления, 104, 109

Ф

Файлы

- Android
 - java.io, 112
 - загрузка, 159
 - копирование файлов, 118
 - обзор, 111
 - определение характеристик файлов, 112
 - получение ссылки на файл, 112
 - создание каталогов, 112
 - создание файлов, 112
 - чтение и запись в файлы, 113

iOS

- URL и строки, 124
- атрибуты файлов, 121
- доступ к каталогам, 120

- загрузка, 166
- копирование файлов, 123
- обзор, 119
- определение характеристик файла, 119
- резервное копирование в iCloud и iTunes, 124
- чтение и запись в файлы, 122
- задачи, 111
- Фоновые потоки
 - Android, 141
 - iOS, 150
- Х**
- Хранение данных
 - Android
 - SQLite, 126
 - выбор базы данных, 132
 - добавление вывода подробных данных в приложение, 309
 - добавление кнопки в приложение, 319
 - запись данных, 129
 - запись книг в хранилище, 322, 324
 - соединение с базой данных, 127
 - создание таблицы или хранимого объекта, 128
 - сохранение книг в закладках, 340
 - чтение данных, 130
- iOS
 - Core Data, 133
 - добавление вывода подробных данных в приложение, 314
 - добавление кнопки в приложение, 320
 - запись данных, 136
 - запись книг в хранилище, 331
 - создание таблиц и объектов, 135
 - чтение данных, 137
- обзор задач, 126
- реляционные базы данных, 126
- Э**
- Эмуляторы, 239

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
тел.: (499) 782-38-89, электронная почта: books@aliants-kniga.ru.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: www.a-planeta.ru.

Шон Льюис и Майк Данн

Нативная разработка мобильных приложений

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70 × 100 1/16.

Гарнитура PT Serif. Печать офсетная.

Усл. печ. л. 30,55. Тираж 200 экз.

Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5

Веб-сайт издательства: www.dmkpress.com