

КНИГА 2

ВАСИЛИЙ УСОВ

SWIFT

РАЗРАБОТКА ПРИЛОЖЕНИЙ
под iOS на основе фреймворка UIKit



КНИГА 2

ВАСИЛИЙ УСОВ

SWIFT

———— РАЗРАБОТКА ПРИЛОЖЕНИЙ ————
под iOS на основе фреймворка UIKit

swiftme.ru

Москва

2021

ББК
УДК
У76

Усов В.

У76 Swift. Разработка приложений под iOS на основе фреймворка UIKit. — Москва, 2021. — 492 с.

ISBN

Swift — один из самых современных языков программирования, вобравший в себя все лучшее из C, Objective-C, Java, Python и многих других. Сегодня он входит в топ-7 по популярности среди всех языков и возглавляет рейтинг языков для мобильной разработки. Создание на Swift приложения под iOS и macOS — это очень творческий процесс, который позволит вам проявить себя. В каждой написанной строчке вы ощутите его лёгкость, гибкость и широкие возможности.

В данной книге вы продолжите свое знакомство с языком Swift и средой разработки Xcode.

Узнаете, что такое архитектура проекта, какой она бывает и из чего состоит; как приложение работает в операционной системе, каков его жизненный цикл и жизненный цикл его отдельных элементов; как происходит обмен данными и как передаются события. И все это вы попробуете на практике, рассматривая возможности фреймворка UIKit.

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

ББК
УДК

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN

КРАТКОЕ СОДЕРЖАНИЕ

https://t.me/it_boooks

Введение..... 10

Часть I. Архитектура iOS-приложения.

Проект «Right on target»..... 15

Глава 1. Игра «Right on target» 19

Глава 2. Введение в шаблон проектирования MVC 48

Глава 3. Введение в жизненный цикл View Controller..... 57

Глава 4. Рефакторинг программного кода 92

Глава 5. Структура и запуск. iOS-приложения..... 105

Часть II. Введение в табличные представления.

Проект «Contacts»..... 125

Глава 6. Шаблоны «Делегирование» и «Источник данных»..... 127

Глава 7. Табличные представления. Класс UITableView. 138

Глава 8. Долговременное хранение данных. User Defaults..... 185

Часть III. Продвинутое табличные представления.

Проект «To-Do Manager»..... 202

Глава 9. Навигационный контроллер.
Класс UINavigationController. 204

Глава 10. Передача данных между контроллерами 227

Глава 11. Контроллер табличного представления.
Класс UITableViewController. 256

Глава 12. Табличные представления на основе прототипов
ячеек 264

Глава 13. Изменение элементов табличного представления 296

Глава 14. Создание и изменение задач..... 313

Глава 15. Завершение разработки приложения 344

Часть IV. Графический интерфейс. Проект «Cards» ... 352

Глава 16. Классы UIView и CALayer	354
Глава 17. Кривые Безье.....	397
Глава 18. Создание игровой карточки. Разработка кастомных представлений и слоев	419
Глава 19. События и анимации в iOS	442
Глава 20. Разработка приложения	461
Заключение.....	490

ОГЛАВЛЕНИЕ

Введение.....	10
---------------	----

Часть I. Архитектура iOS-приложения.

Проект «Right on target».....	15
-------------------------------	----

Глава 1. Игра «Right on target».....	19
--------------------------------------	----

1.1 Создание и подготовка проекта	20
1.2 Разработка графического интерфейса	28
1.3 Программирование бизнес-логики	41

Глава 2. Введение в шаблон проектирования MVC.....	48
--	----

2.1 Архитектурные шаблоны проектирования	48
2.2 Шаблон проектирования MVC	50
2.3 Шаблон MVC в приложении «Right on target»	53

Глава 3. Введение в жизненный цикл View Controller.....	57
---	----

3.1 Понятие жизненного цикла.....	57
3.2 Жизненный цикл View Controller	58
3.3 «Right on target», версия 1.1	60
3.4 Введение в отображение графических элементов.....	65
3.5 Схема жизненного цикла View Controller	69

Глава 4. Рефакторинг программного кода	92
--	----

4.1 Рефакторинг программного кода	93
4.2 «Right on target», версия 1.2.....	93
4.3 «Right on target», версия 1.3. Самостоятельная работа	101
4.4 «Right on target», версия 1.4. Самостоятельная работа	103

Глава 5. Структура и запуск iOS-приложения	105
--	-----

5.1 Класс UIApplication	105
5.2 Паттерн Делегирование и класс AppDelegate	107
5.3 Классы UIWindowScene и SceneDelegate	111
5.4 Класс UIWindow.....	118

Часть II. Введение в табличные представления.

Проект «Contacts».....	125
------------------------	-----

Глава 6. Шаблоны «Делегирование» и «Источник данных» ..	127
6.1 Шаблон «Делегирование»	127
6.2 Шаблон «Источник данных»	135
Глава 7. Табличные представления. Класс UITableView.....	138
7.1 Введение в табличные представления.....	138
7.2 Использование табличного представления.....	145
7.3 Создание и конфигурирование ячеек.....	154
7.4 Разработка модели	164
7.5 Удаление контактов	168
7.6 Создание контактов.....	174
Глава 8. Долговременное хранение данных. User Defaults.	185
8.1 Варианты долговременного хранения данных.....	185
8.2 User Defaults.....	189
8.3 Хранение контактов в User Defaults	196
8.4 Распределение элементов проекта по папкам.....	200
Часть III. Продвинутое табличные представления.	
Проект «To-Do Manager».....	202
Глава 9. Навигационный контроллер. Класс UINavigationController.	204
9.1 Навигационный контроллер.....	205
9.2 Создание навигационного контроллера.....	207
9.3 Навигационный стек.....	212
9.4 Навигация с помощью программного кода	216
9.5 Визуальное оформление Navigation Controller	223
Глава 10. Передача данных между контроллерами.....	227
10.1 Создание проекта	228
10.2 Передача данных от А к Б с помощью свойств.....	232
10.3 Передача данных от Б к А с помощью свойств.....	235
10.3 Передача данных от А к Б с помощью segue.....	238
10.4 Передача данных от Б к А с помощью unwind segue.....	241
10.5 Передача данных от Б к А с помощью делегирования.....	244
10.6 Передача данных от Б к А с помощью замыкания.....	248
10.7 Другие способы передачи данных	251

Глава 11. Контроллер табличного представления.	
Класс UITableViewController.	256
11.1 Создание проекта на основе Table View Controller	257
11.2 Класс UITableViewController	258
11.3 Разработка прототипа Модели.....	260
Глава 12. Табличные представления на основе прототипов	
ячеек	264
12.1 Прототипы ячеек	264
12.2 Создание прототипов ячеек.....	265
12.3 Создание прототипа с использованием констрейнтов и тегов	268
12.4 Создание прототипа с использованием Horizontal Stack View и кастомного класса.....	273
12.5 Наполнение таблицы тестовыми данными.....	281
Глава 13. Изменение элементов табличного представления	296
13.1 Изменение статуса задач.....	296
13.2 Режим редактирования.....	300
13.3 Удаление задач с помощью режима редактирования.....	306
13.4 Сортировка задач с помощью режима редактирования ...	309
Глава 14. Создание и изменение задач	313
14.1 Экран создания и изменения задачи.....	314
14.2 Таблица на основе статических ячеек	316
14.3 Ячейка для названия задачи.....	320
14.4 Ячейка для типа задачи.....	321
14.5 Создание экрана выбора типа задачи	323
14.6 Передача данных между сценами	335
14.7 Ячейка изменения статуса задачи.....	337
14.8 Сохранение задачи	340
Глава 15. Завершение разработки приложения.....	344
15.1 Доработка хранилища задач	344
15.2 Недостатки приложения To-Do Manager	349
Часть IV. Графический интерфейс. Проект «Cards» ..	352
Глава 16. Классы UIView и CALayer.....	354

16.1 Фреймворки UIKit, Core Animation и Core Graphics	355
16.2 Точки и пиксели	357
16.3 Позиционирование представлений и системы координат	359
16.4 Создание кастомных представлений	362
16.5 Класс CALayer	375
16.6 Свойство transform.....	384
16.7 Свойство bounds	390
Глава 17. Кривые Безье	397
17.1 Что такое кривые Безье	397
17.2 Создание кривых Безье.....	399
Глава 18. Создание игровой карточки.	
Разработка кастомных представлений и слоев	419
18.1 Требования к игровой карточке	419
18.2 Создание кастомных слоев для фигур	421
18.3 Создание кастомного представления для игровой карточки	430
18.4 Как представления появляются на экране	437
Глава 19. События и анимации в iOS	442
19.1 События	442
19.2 События касания.....	444
19.3 Responder Chain	447
19.4 Пример обработки событий. Перемещение игровых карточек.....	451
19.5 Анимации графических элементов	452
19.6 Анимированные переходы.....	455
19.7 Доработка игровой карточки	457
Глава 20. Разработка приложения	461
20.1 Распределение типов по проекту	462
20.2 Разработка Модели	464
20.3 Разработка Представления. Связь кнопки и метода	474
20.4 Шаблон проектирования «Фабрика». Фабрика фигур	478
20.5 Размещение игровых карточек на игровом поле	481
20.6 «Cards», версия 1.1. Самостоятельная работа.....	486
Заключение.....	490

Самое ценное, что есть в моей жизни – это родные и любимые. Только благодаря вашей поддержке два года активной «работы после работы» смогли явить себя миру в виде этой книги. Посвящается вам.

Особое спасибо за вычитку, техническую редактуру и тестирование кода:

Александру Воробьеву (@mosariot)
Руслану Уразбахтину (@iruspro11)

Спасибо за прекрасную корректуру и редактуру:

Максиму Житову (@KingOfChidori)

Спасибо за помощь мне с администрированием чата в Telegram:

Тимуру Фатуллаеву (@tima5959)
Алмазу Рахматуллину (@almazof_102)
Денису Роеенко (@g01dt00th)

Без вас этот труд был бы неподъемным.

Введение

«Регулярно инвестируйте в свои знания»

Дэвид Томас, книга «Программист-прагматик»

С момента, как я написал первые строки этой книги до момента ее опубликования, прошло более двух лет. Подумать только, сколько времени было потрачено, сколько труда вложено! И сейчас, оглядываясь назад, я вижу, что все было не зря. Работа над книгой была прекрасным и очень увлекательным занятием, в ходе которого я не просто перекладывал на бумагу свои мысли и знания, но и продолжал учиться сам. Продолжаю учиться и сейчас.

Все эти годы я был в поисках новых подходов к обучению, а также внимательно наблюдал за тем, как развивается Swift. Ситуация с созданием книги немного осложнялась высокой занятостью на основной работе, необходимостью регулярного обновления первой книги и работой с зарождающимся в русскоговорящем IT-сегменте сообществом Swift-разработчиков. После написания новой главы мне зачастую приходилось пересматривать и переписывать написанный ранее материал, ведь знания в IT-сфере быстро устаревают, особенно в таком интенсивно развивающемся направлении, как разработка под iOS и macOS. При этом в результате мне не хотелось получить книгу, состоящую из переводов англоязычных учебников и статей. Я хотел создать что-то свое, что-то действительно оригинальное и уникальное.

И, кажется, мне это удалось.

На сегодняшний день, когда Swift и Xcode достигли высокого уровня стабильности, я готов представить вам результаты своих трудов. Эта книга стала итогом долгого и плодотворного общения с вами, мои дорогие читатели. Каждый из вас, кто участвовал в дискуссиях, выражал свои пожелания и мнение, внес неоценимый вклад в развитие проекта. Сейчас я уверен, что пока вы читаете книгу, пока находитесь в путешествии между первой и последней страницами, материал, изложенный в ней, не потеряет своей актуальности.

Эта книга – мои инвестиции в будущее русскоговорящего сегмента разработчиков. Дальше дело за вами. Чтобы быть на острие индустрии, теперь инвестировать в свои знания должны вы. Знания – это лучшая инвестиция, которую только можно придумать.

Читая книгу, вы узнаете много интересного материала, который, безусловно, потребуется вам в дальнейшей работе в качестве разработчика. Мы создадим несколько несложных проектов, параллельно рассматривая возможности фреймворка UIKit и постепенно углубляясь в архитектуру приложений. Материал книги — это не просто «набивание» кода и размещение UI-элементов «вслед за автором». Я старался, чтобы вы достигли полного понимания каждого элемента и механизма, который используется в процессе разработки приложений.

Приятного и полезного чтения!

Присоединяйтесь к нам

Самые важные ссылки я собрал в одном месте. Обязательно посетите каждую из них — это поможет в изучении материала книги.

САЙТ СООБЩЕСТВА



<https://swiftme.ru>

Swiftme.ru — это развивающееся сообщество программистов на Swift. Здесь вы найдете ответы на вопросы, возникающие в ходе обучения и разработки, а также сможете пройти уроки и курсы, которые помогут вам глубже и более детально изучить тему разработки приложений.



МЫ В TELEGRAM

<https://swiftme.ru/telegramchat> или @usovswift

Если по ходу чтения книги у вас появились вопросы, вы можете задать их в нашем чате в Telegram.



ОПЕЧАТКИ КНИГИ

<https://swiftme.ru/typo21>

Здесь вы можете посмотреть перечень всех опечаток, а также оставить информацию о найденных вами и еще не отмеченных. Документ создан в Google Docs, поэтому для доступа к нему вам потребуется Google-аккаунт.



ЛИСТИНГИ

<https://swiftme.ru/listings21>

Здесь вы можете найти большую часть листингов и проектов, создаваемых в книге.

Для кого написана книга

Книга, которую вы держите в руках, предназначена для начинающих разработчиков, имеющих базовые знания синтаксиса и возможностей Swift. Если вы еще не знакомы с данным языком, но хотите его освоить, начните свое обучение с предыдущей книги серии «Swift. Основы разработки приложений под iOS и macOS».

О том, как работать с книгой

В ходе чтения книги вы встретите не только теоретические сведения, но и большое количество практических примеров и заданий, выполняя которые вы углубите свои знания в области разработки iOS-приложений на языке Swift. Вам предстоит пройти большой путь, и поверьте, он будет очень интересным.

Книга предназначена, в первую очередь, для изучения практической стороны и принципов разработки полноценных приложений.

Код и проекты в книге написаны с использованием Swift 5.3, iOS 14, Xcode 12 и операционной системы macOS 11 Big Sur. Если у вас установлены более свежие версии программного обеспечения, вполне возможно, что показанный на скриншотах интерфейс будет немного отличаться от вашего. При этом весь приведенный программный код с большой долей вероятности будет работать без каких-либо правок. Тем не менее, если вы встретитесь с ситуацией, при которой интерфейс или код необходимо исправить, прошу вас сообщить мне об этом одним из способов:

- отметить в специальном электронном файле (<https://swiftme.ru/typo21>);
- написать в Telegram-канале (<https://swiftme.ru/telegramchat> или @usovswift).

Очень важно, чтобы вы не давали своим рукам «простаивать». Тестируйте весь предлагаемый код и выполняйте все задания, так как учиться программировать, просто читая текст — не лучший способ. Если в процессе изучения нового материала у вас появится желание «поиграть» с кодом из листингов, делайте это, не откладывая. Постигайте Swift!

Не бойтесь ошибаться: пока вы учитесь, ошибки — ваши друзья. Помните, что с каждой совершенной ошибкой, к вам приходит бесценный опыт. А исправлять их и избегать в будущем вам помогут среда разработки Xcode и мои книги.

Помните: чтобы стать великим программистом, потребуется время. Много времени! Будьте терпеливы и внимательно изучайте материал. Желаю увлекательного путешествия!

Очень часто во время написания кода начинающие разработчики пользуются нехитрым приемом «копировать/вставить». Они копируют все, начиная от

решения домашних заданий и заканчивая найденными в Сети готовыми участками кода, решающими определенную проблему. Недостаток такого подхода в том, что чаще всего человек не разбирается в том, что копирует. Решение задачи проходит мимо и не оседает в его голове.

Конечно, в некоторых случаях такой подход может ускорить достижение цели получения работающего кода. Но в действительности ваша цель, заключающаяся в получении глубоких знаний для повышения собственного уровня развития, так и не будет достигнута.

Я настоятельно советую разбирать каждый пример или рецепт и не гнаться за готовыми решениями. Каждое нажатие на клавиатуру, каждое написание символа должно быть осознанным.

Если у вас возникают проблемы с решением задания, смело обращайтесь к нашему сообществу в Telegram.

Старайтесь решать задания самостоятельно, используя при этом помощь сообщества, книгу и другие справочные материалы. Но не ориентируйтесь на то, чтобы посмотреть (или правильнее — подсмотреть?) готовое решение.

Экспериментируйте, пробуйте, тестируйте — среда разработки выдержит даже самый некрасивый и неправильный код!

Структура книги

Весь последующий материал книги разделен на четыре части:

- **Часть I. Архитектура iOS-приложения. Проект «Right on target».** Первый проект, который будет реализован — это небольшая игра на угадывание числа с помощью перемещения слайдера. В этой главе вы вспомните базовые знания по работе со средой Xcode, а также начнете изучать архитектурный шаблон проектирования MVC. В этой главе закладываются базовые и самые важные знания для изучения всего последующего материала.
- **Часть II. Введение в табличные представления. Проект «Contacts».** Табличные представления (Table View) — это один из наиболее часто используемых элементов, используемый для удобной компоновки данных в табличном виде. В этой главе вы начнете знакомство с тем, как создаются таблицы в iOS, и на их основе разработаете приложения для хранения контактов «Contacts».
- **Часть III. Продвинутое табличные представления. Проект «To-Do Manager».** В процессе создания менеджера задач «To-Do Manager» мы продолжим изучение табличных представлений и их возможностей. Также в этой главе будут рассмотрены способы передачи данных между сценами проекта, а также рассмотрен вариант перехода от одной сцены к другой с помощью навигационного контроллера.

- **Часть IV. Графический интерфейс. Проект «Cards».** В этой части мы рассмотрим принципы создания кастомных графических элементов, обработки событий и работы с анимацией. Все это будет сделано в процессе разработки игры на поиск пар одинаковых карточек «Cards».

Условные обозначения

ПРИМЕЧАНИЕ

В данном блоке приводятся примечания и замечания.

Задание

В данном блоке приводятся задания для самостоятельного решения.

ЛИСТИНГ

А это примеры кода (листинги).

СИНТАКСИС

В таких блоках приводятся синтаксические конструкции с объяснением вариантов их использования.



Такие блоки содержат указания, которые вам необходимо выполнить.

Часть I

АРХИТЕКТУРА iOS-ПРИЛОЖЕНИЯ

ПРОЕКТ «RIGHT ON TARGET»

Архитектура – это совокупность соглашений, описывающих состав элементов программы и порядок их взаимодействия

или, другими словами,

Архитектура описывает блоки, из которых состоит программа и то, как эти блоки взаимодействуют между собой.

Архитектура есть у любой программы. В некоторых случаях, когда разработчик не задумывается о реализуемых сущностях, порядке их взаимодействия и не видит картину в целом, архитектура проекта может представлять из себя достаточно неустойчивую структуру (рис.1), в которой элементы опираются друг на друга, а их взаимодействие хаотично и происходит по не определенным правилам. При таком подходе изменение или удаление любого из блоков может привести к тому, что весь проект развалится и перестанет функционировать.

Существует множество архитектурных подходов, позволяющих привнести в структуру проекта ясность и логичность, а также сделать ее гибкой и безопасной. В этом случае каждый отдельный блок не зависит от остальных и может быть изменен или удален без каких-либо последствий (для других блоков). Такие подходы и решения позволяют создать *хорошую архитектуру*.

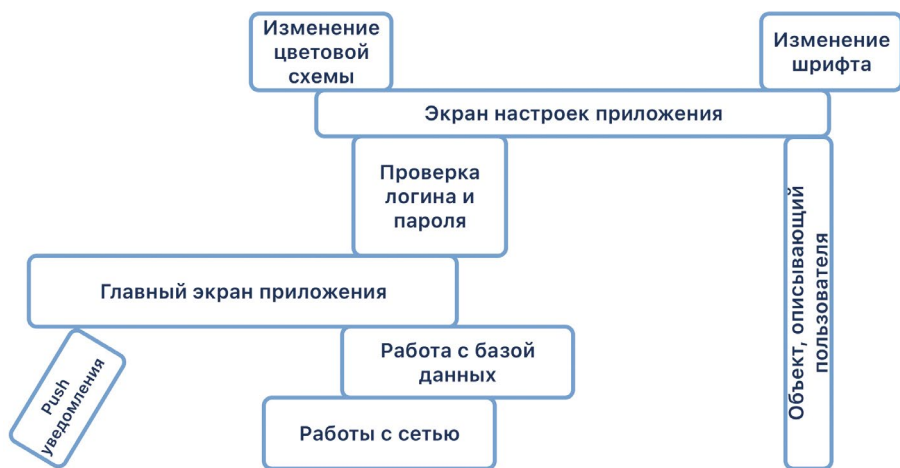


Рис. 1. Пример плохой архитектуры

Но что такое **хорошая архитектура**? Это понятие несколько абстрактно и субъективно, так как каждый разработчик может интерпретировать его с учетом своего собственного опыта и взглядов. Тем не менее, в общем случае архитектура приложения может называться хорошей, если она соответствует двум критериям:

- 1. Гибкость.** В проекте должна быть обеспечена возможность легкого внесения изменений в любой элемент программы с целью модификации или расширения его функциональных возможностей.
- 2. Безопасность.** Внесение изменений во внутреннюю структуру одного элемента программы не должно влиять на другие элементы. Т.е. каждый элемент программы должен быть максимально изолированным от других, а взаимодействие между ними должно вестись по заранее определенным правилам и соглашениям.

Каждый из указанных критериев может достигаться десятками различных способов. Со временем, когда вы прочитаете больше профессиональных статей и книг (в том числе, и в процессе работы с этой книгой), вы познакомитесь с различными паттернами и принципами разработки, способными улучшить внутреннюю структуру ваших проектов. С их помощью вы будете добиваться соблюдения критериев гибкости и безопасности.

Для того, чтобы лучше понять, что же такое «гибкость» и «безопасность», рассмотрим следующий пример. Вы создали новостное приложение, которое загружает статьи с удаленного сервера с помощью API¹, описанного в документации. В определенный момент владелец сервера производит доработку и обновляет API доступа к информационным ресурсам, вследствие чего перед

вами возникает задача обновления элемента приложения, отвечающего за работу с сетью. Так как ваше приложение имеет *хорошую архитектуру*:

1. вы с легкостью находите элемент, в который необходимо внести изменения;
2. внесенные изменения никак не повлияли на работу других элементов, и вам не потребовалось дополнительно модифицировать что-то еще (например, экран, отображающий статьи).

Несмотря на то, что API сервера изменилось, это не привело к тому, что весь код вашей программы стал требовать доработки. Это достигается благодаря разделению зон ответственности внутри проекта, когда каждый элемент программы решает только одну конкретную задачу и при этом взаимодействует с другими элементами только в строгом соответствии с описанными правилами.

ПРИМЕЧАНИЕ Серверная часть любого программного продукта обычно называется backend («бэкенд», «бэк»). Возможно, вам уже приходилось слышать это понятие. Обычно в качестве бэка выступает сервер в сети Интернет (или локальной сети), который принимает запросы от приложения, обрабатывает их и возвращает результат.

Чаще всего бэкенд используется в том случае, когда необходимо удаленно обновлять данные в приложении (новости, статьи, данные о погоде, результаты спортивных матчей, курсы валют) или синхронизировать данные между пользователями (онлайн-игры), а также во многих других случаях.

Архитектуру iOS-приложения условно можно разделить на 2 уровня:

1. **Архитектура верхнего уровня**, определяющая, как ваше приложение функционирует в среде операционной системы, какие в нем рождаются элементы и как они взаимодействуют между собой. Данный уровень реализуется непосредственно операционной системой iOS (iPadOS), а также фреймворком **UIKit** – основой любого мобильного приложения (в том числе, написанного с использованием **SwiftUI**).

2. **Архитектура нижнего уровня**, которая определяет то, каким образом организуется внутренний программный код приложения (классы, структуры, функции и т.д.).

В этой книге мы будем преследовать три основные цели:

1. рассмотреть новые для вас возможности Xcode и Swift, необходимые для создания приложений;
2. рассмотреть архитектуру верхнего уровня;
3. научиться создавать хорошую архитектуру нижнего уровня.

¹ API – Application Programming Interface, программный интерфейс, описывающий способы и правила взаимодействия с сервером, сервисом, модулем, программой или любым другим программным объектом. Например, для web-сервера API обычно описывает HTTP-запросы, которые необходимо отправить для получения информации, размещенной на нем.

Каждая глава будет приближать вас к ответу на вопросы «как работает iOS-приложение» и «как создать хорошую архитектуру приложения». Помните, что совсем не сложно написать *калькулятор* – сложнее создать для него такую архитектуру, при которой дальнейшее развитие и поддержка проекта будут требовать минимальных затрат сил и времени.

Навык создания хорошей архитектуры – один из важнейших для профессионального iOS-разработчика. К сожалению, довольно часто понимание этого приходит лишь после того, как за плечами уже имеется несколько проектов, и когда программист начинает закапываться в собственном коде, не понимая, за что отвечает тот или иной блок его программы. Внесение изменений и наращивание функциональности в своих же проектах превращаются в довольно тяжелую и совсем не быструю задачу. Я считаю, что вам необходимо задумываться о том, что такое архитектура iOS-проекта уже сейчас, в самом начале своей карьеры.

В этой главе мы будем изучать новые для вас возможности среды разработки Xcode и языка Swift и параллельно говорить о том, как сделать код своих проектов удобным или, другими словами, как создать *хорошую архитектуру*. Все это будет происходить параллельно разработке игры «**Right on target**». На ее основе мы рассмотрим основные процессы в жизни iOS-приложения.

Глава 1. Игра «**Right on target**»

Глава 2. Введение в шаблон проектирования MVC

Глава 3. Введение в жизненный цикл View Controller

Глава 4. Рефакторинг программного кода

Глава 5. Структура и запуск iOS-приложения

Глава 1.

Игра «Right on target»

В этой главе вы:

- научитесь работать с новым для вас графическим элементом фреймворка **UIKit** – слайдером (класс **UISlider**);
- узнаете, как обеспечить поддержку различных ориентаций экрана в приложении;
- начнете изучать вопросы унифицированного позиционирования графических элементов на сцене;
- узнаете о том, что такое бизнес-логика приложения.



Весь программный код главы доступен по следующей ссылке:

<https://swiftme.ru/listings21>

Мы начинаем работу над довольно интересным и простым проектом – игрой «**Right on target**» (рис. 1.1)¹. Ее основная цель будет состоять в том, чтобы набрать максимальное количество очков путем установки бегунка на слайдере на определенную, загаданную приложением, позицию. Интерфейс будущего приложения будет состоять всего из трех графических элементов: слайдера (класс **UISlider**), кнопки (класс **UIButton**) и текстовой метки (класс **UILabel**).

ПРИМЕЧАНИЕ Терминология, используемая разработчиками, порой может вводить в ступор, так как мы очень любим использовать англоязычные термины наравне с переведенными. Для того, чтобы вы уже сейчас начинали привыкать к такой форме общения, в книге я буду использовать как оригинальные (в том числе написанные кириллицей), так и переведенные названия элементов.

Так, кнопка также будет обозначаться как **Button**, текстовая метка – **Label**, переход – **segue**, раскладка (так никто не говорит, ее называют сторибордом) – **storyboard**, контроллер представления – **вью контроллер** и **View Controller** и т.д.

¹ Идея приложения взята у коллектива разработчиков Ray Wenderlich. Они создают прекрасные учебные материалы на английском языке для Swift-разработчиков.

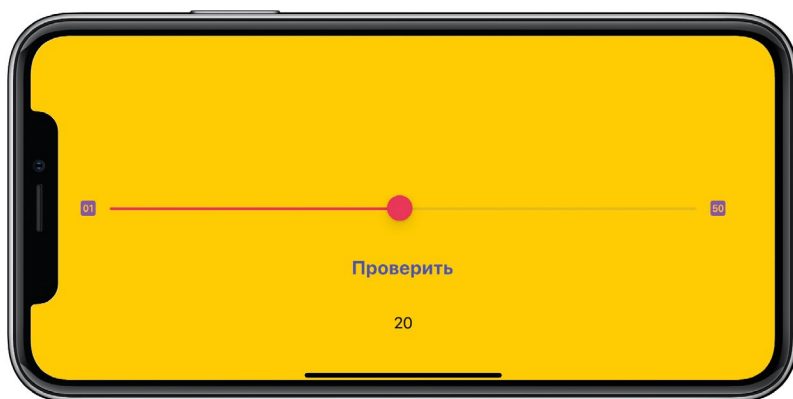


Рис. 1.1. Интерфейс игры «Right on target»

Игра «**Right on target**» будет проходить по следующим правилам:

1. Приложение случайным образом выбирает целое число от 1 до 50 и сообщает его игроку, выводя в текстовой метке.
2. Игрок устанавливает бегунок на слайдере в положение, которое, по его мнению, соответствует данному числу. При этом крайнее левое положение слайдера соответствует 1, крайнее правое – 50. Сложность состоит в том, что ориентироваться игроку приходится на глаз.
3. Игрок подтверждает выбранную позицию слайдера нажатием кнопки.
4. Приложение определяет, насколько близко к загаданному числу оказалось выбранное на слайдере значение, и вычисляет количество заработанных очков.

Одна игра длится пять раундов. Цель – набрать максимальное суммарное количество очков.

1.1 Создание и подготовка проекта

Создание нового приложения всегда начинается с создания проекта в Xcode.

- ▶ Запустите Xcode.
- ▶ В стартовом окне нажмите **Create a new Xcode project**.

Приложение будет использовать всего один экран (одну сцену). По этой причине для него прекрасно подойдет шаблон **App**.

ПРИМЕЧАНИЕ Проект, созданный по шаблону **App**, изначально содержит одну сцену (один выю контроллер).

- В качестве операционной системы выберите **iOS**, а шаблона – **App** (рис. 1.2), и нажмите **Next**.

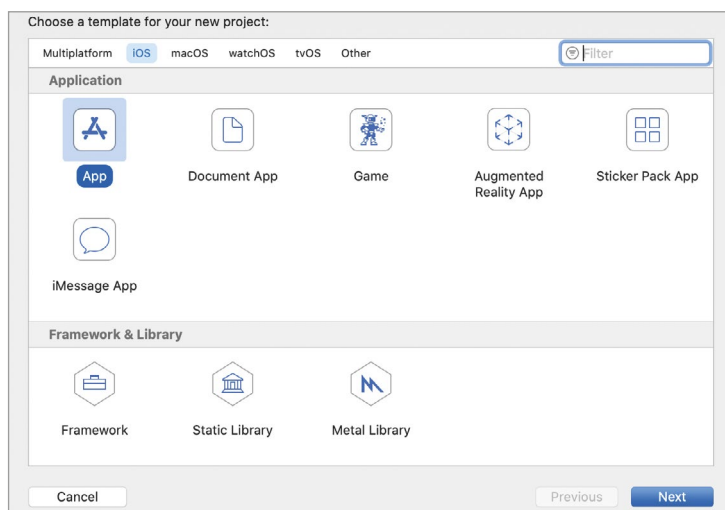


Рис. 1.2. Выбор шаблона приложения

- В окне настроек заполните требуемые поля (рис. 1.3):
 - **Product Name** определяет название проекта.
Укажите «**Right on target**».
 - **Team** – команда разработки.
Если у вас есть аккаунт разработчика, выберите его. В ином случае, можете оставить это поле пустым.
 - **Organization Identifier** – идентификатор организации, который используется для создания уникального идентификатора приложения. Обычно указывается доменное имя, написанное в обратном порядке, например, «**ru.swiftme**».
Вы можете использовать собственный или вписать «**ru.swiftme**».
 - **Interface** определяет способ создания интерфейса приложения: с помощью **UIKit** или **SwiftUI**.
Выберите значение **Storyboard**.
 - **Life Cycle** определяет способ управления жизненным циклом приложения.

Выберите **UIKit App Delegate**.

- **Language** – язык программирования, на котором будет вестись разработка.

Мы учимся разрабатывать на Swift.

- Пункты **Use Core Data**, **Include Tests** должны быть отключены.

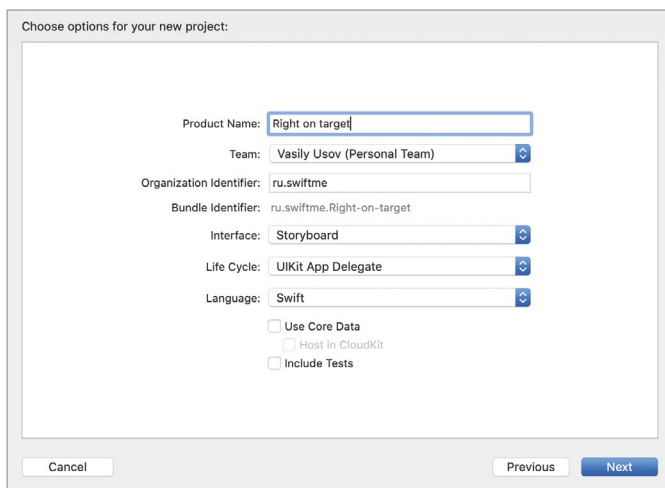


Рис. 1.3. Первичные настройки проекта

- Нажмите **Next** и сохраните проект.

После выполнения указанных действий перед вами откроется уже знакомая рабочая среда Xcode (рис. 1.4).

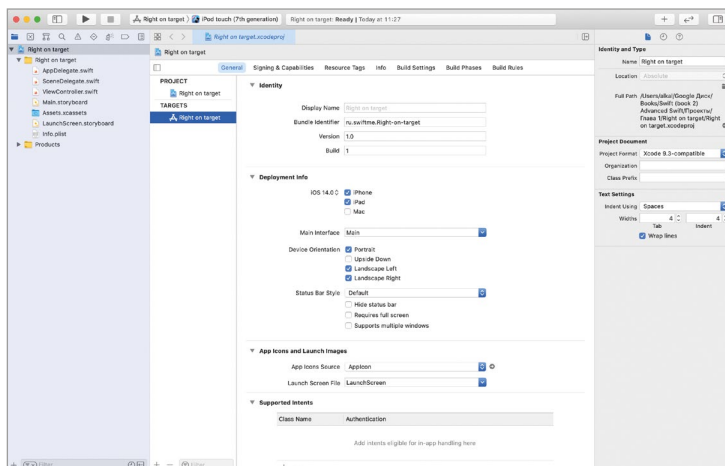


Рис. 1.4. Рабочая среда Xcode

Рабочая среда Xcode

Мы уже неоднократно использовали среду разработки Xcode в процессе обучения. Тем не менее, вспомним, из каких областей состоит ее интерфейс (рис. 1.5):

- **Toolbar** служит для запуска и остановки сборки приложения, выбора настроек, а также отображает текущий статус проекта.
- **Navigator** служит для навигации по проекту. Сейчас в этой панели открыт **Project Navigator**, отображающий все ресурсы, входящие в проект.
- **Inspectors** служит для просмотра и изменения свойств активного элемента.
- **Project Editor** является основной рабочей площадкой во время разработки приложения. Именно в этой панели производится настройка проекта, пишется исходный код, создается структура баз данных, подключаются внешние модули и многое другое.

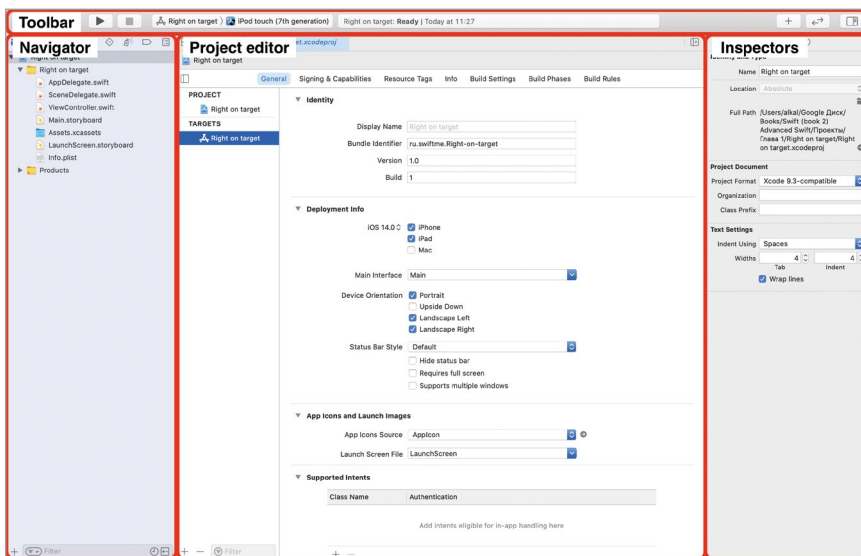


Рис. 1.5. Основные рабочие области Xcode

- Перейдите к файлу **Main.storyboard** в панели **Project Navigator**.

Внешний вид и структура **Project Editor** изменяется в соответствии с выбранным в **Project Navigator** ресурсом. Так как сейчас активным является файл **Main.storyboard**, в центральной части окна отображаются **Document Outline** и **Interface Builder** (рис. 1.6). Панель **Document Outline** предоставляет доступ к структуре проекта. В ней отображаются все доступные сцены (на данный момент там находится всего один элемент).

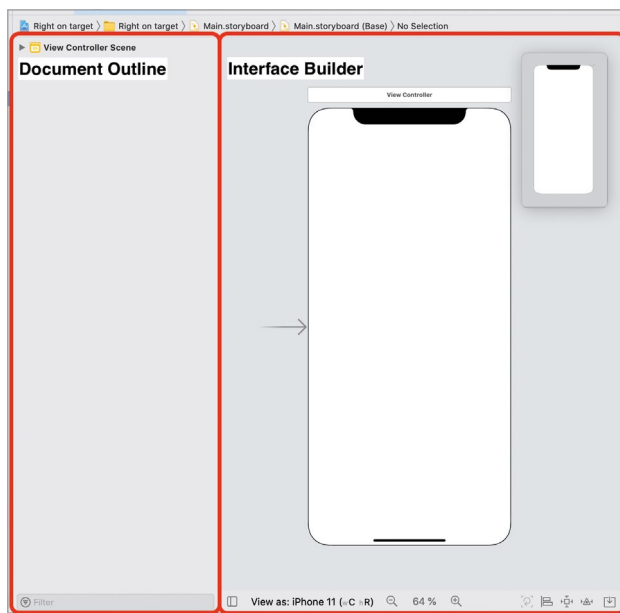


Рис. 1.6. Interface Builder и Document Outline

Если в **Project Navigator** выделить файл с исходным кодом (он имеет расширение swift), то вместо **Interface Builder** отобразится **Code Editor** (редактор кода) (рис. 1.7).

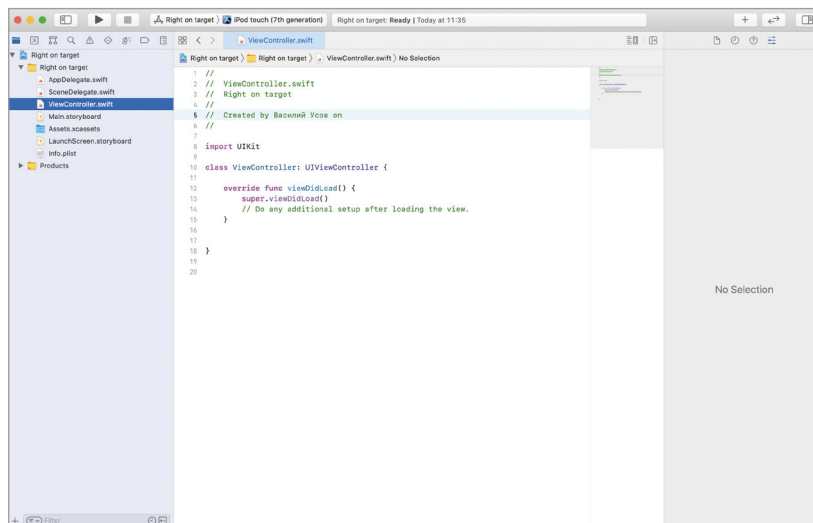


Рис. 1.7. Редактирование файла с исходным кодом

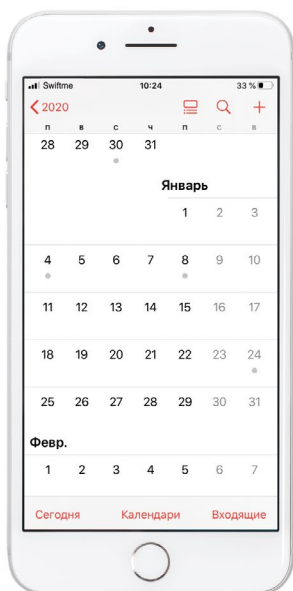
Ориентация экрана

Одной из возможностей операционной системы iOS является запуск приложений в различных ориентациях устройства: одни приложения удобно использовать в альбомной ориентации (например, медиа-плеер VLC при просмотре фильма), другие удобно в портретной (например, при навигации по файловой системе в Файлах), а третьи должны поддерживать сразу обе ориентации (например, Калькулятор). О поддержке необходимых ориентаций программист должен позаботиться на этапе разработки проекта.

Примечание Портретный режим (или портретная ориентация), также называемая книжной, на английском звучит, как *portrait*. Альбомный режим, он же пейзажный или горизонтальный, на английском – *landscape*. Обе ориентации продемонстрированы на рисунке 1.8.

Для изменения поддерживаемой ориентации используется меню настроек (рис. 1.9).

Портретная ориентация



Альбомная ориентация

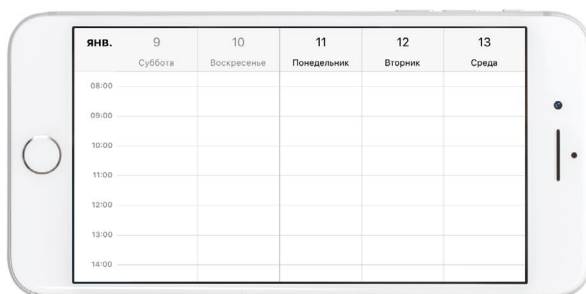


Рис. 1.8. Примеры ориентаций iPhone

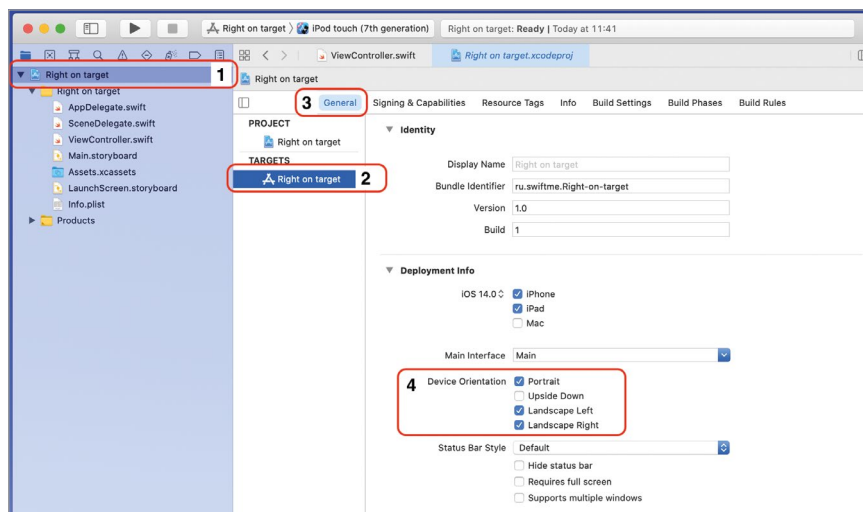


Рис. 1.9. Настройка ориентации приложения

- ▶ (1) Щелкните по файлу проекта в **Project Navigator** (синий, в самом верху).
- ▶ (2) В левой панели **Project Editor** выберите пункт с названием проекта в разделе **Targets**.
- ▶ (3) В **Project Editor** выберите вкладку **General**.

В разделе **Device orientation** (4) вы можете определить доступные ориентации. Всего на выбор доступны 4 варианта:

- **Portrait** – портретная ориентация;
- **Upside Down** – портретная перевернутая ориентация (когда порт зарядки находится сверху);
- **Landscape Left** и **Landscape Right** – альбомная ориентация, когда устройство повернуто влево и вправо соответственно.

Активация пунктов позволяет осуществлять поддержку соответствующих им ориентаций. Тем не менее, выбор настроек является необходимым, но недостаточным условием для того, чтобы интерфейс приложения начал автоматически подстраиваться под изменение положения устройства. Помимо этого, в процессе разработки вам потребуется определять положение графических элементов, указывая правила их размещения или математически высчитывая их позиции (на самом деле, это не так сложно, как звучит).

Поговорим о том, поддержку каких ориентаций необходимо включить в приложение. Основным элементом интерфейса, с которым будет взаимодействовать пользователь в разрабатываемой нами игре, станет Slider. Чем больше

ширина слайдера, тем выше точность выбора значения на нем. При использовании горизонтальной ориентации слайдер будет иметь большие размеры, поэтому логичным решением станет разработка интерфейса исключительно под эту ориентацию.

- (рис.1.9, пункт 4) В разделе **Device orientation** снимите галочки со всех пунктов, кроме **Landscape Left** и **Landscape Right**.

Теперь программа настроена для работы исключительно в горизонтальном режиме. При этом совершенно неважно, в какую сторону будет повернуто устройство (влево или вправо).

Примечание В продуктовой линейке Apple находится внушительное количество устройств, в которых используются различные экраны с отличающимся разрешением и соотношением сторон. Но Xcode позволяет вести унифицированную разработку интерфейса, создавая приложения сразу под iPhone, iPad и даже macOS. В процессе чтения книги вы будете постепенно узнавать о доступных вам возможностях, благодаря которым размещение графических элементов, несмотря на большой зоопарк устройств, становится совсем не сложным делом.

Посмотрим, изменилось ли что-нибудь в **Interface Builder**.

- В **Project Navigator** выберите файл **Main.storyboard**.

Несмотря на внесенные изменения, сцена на storyboard все еще находится в книжной ориентации. Дело в том, что настройки проекта изменяют то, как приложение будет работать на устройстве (или в симуляторе), а storyboard – это лишь инструмент для создания интерфейса. И если перед вами стоит цель создания интерфейса для альбомной ориентации, сцену потребуется повернуть непосредственно в **Interface Builder**. Для этого выполним следующие действия (рис. 1.10):

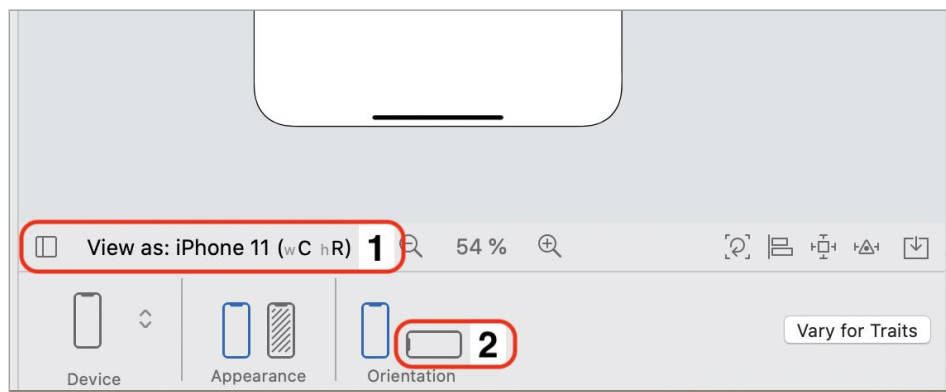


Рис. 1.10. Изменение ориентации сцены

- ▶ (1) В нижней части **Interface Builder** нажмите на **View as**, после чего откроется панель настройки сцены. В левой части доступен выбор устройства, а в правой – его ориентации.
- ▶ (2) В правой части в разделе **Orientation** щелкните по иконке, соответствующей альбомной ориентации.

Теперь сцена на сториборде имеет альбомную ориентацию, что позволит нормально вести дальнейшую работу над интерфейсом. Вы можете скрыть панель настройки, повторно щелкнув по **View as** (рис. 1.10, пункт 1).


1.2 Разработка графического интерфейса

Перейдем к разработке графического интерфейса приложения. На сцене, которая уже входит в состав storyboard, необходимо разместить следующие UI-элементы:

- Слайдер (Slider) – в центре сцены, растянут на всю ее ширину с небольшими отступами слева и справа.
- Кнопка (Button) – ниже слайдера.
- Текстовая метка (Label) – ниже кнопки.

Слайдер

Слайдер будет использоваться при выборе загаданного программой числа. Разместим этот элемент на сцене (рис. 1.11):

- ▶ (1) В **Project Navigator** выберите файл **Main.storyboard**.
- ▶ (2) Откройте панель **Library** (с помощью кнопки **Library**  на **Toolbar** или одновременного нажатия клавиш **⇧Shift + ⌘Command + L**).
- ▶ (3) В появившемся окне найдите UI-элемент Slider. Для удобства вы можете использовать строку поиска.
- ▶ (4) Перетащите элемент в центр сцены. При позиционировании слайдера на сцене будут отображаться вспомогательные синие линии, которые помогут вам установить элементы точно в требуемом месте.

Слайдер размещен, но перед нами возникли две проблемы:

1. Запуск приложения на устройстве, отличном от указанного в нижней части **Interface Builder** (на рисунке это iPhone 11), приведет к тому, что эле-

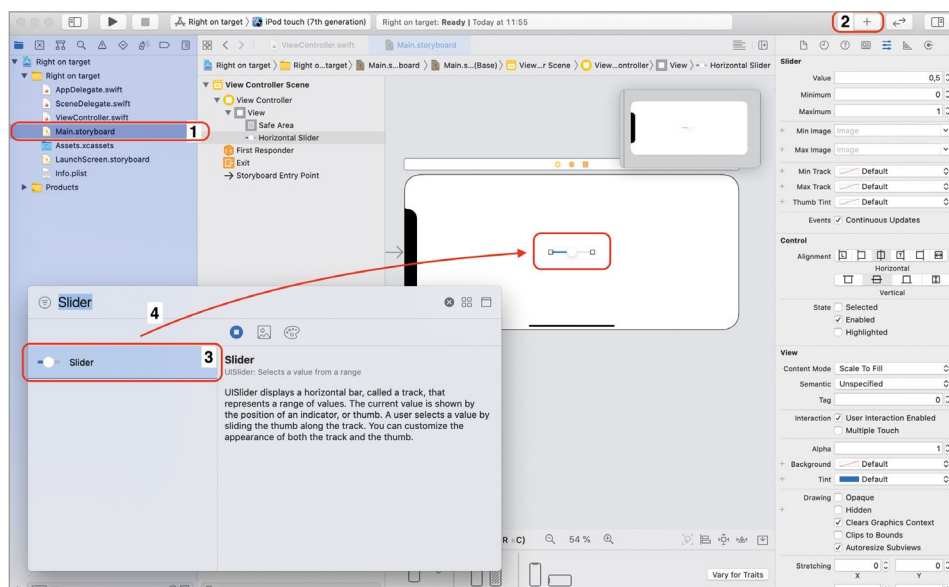


Рис. 1.11. Размещение UI-элементов на сцене

мент будет смещен, т.е. расположен не в центре. Ранее мы уже встречались с такой ситуацией в первой книге.

2. Так как по умолчанию слайдер очень маленький, необходимо увеличить его размер (растянуть в стороны). При этом слайдер должен оставаться растянутым на любом устройстве (речь идет только про различные модели iPhone).

Xcode и Swift предоставляют несколько механизмов, способных решать проблемы, связанные с созданием универсальных интерфейсов. Одним из них является Auto Layout – система позиционирования графических элементов, позволяющая создавать адаптивный интерфейс, подстраивающийся под используемое устройство и текущую ориентацию экрана. Основным элементом Auto Layout являются констрейнты (от англ. constraints, или ограничения), с помощью которых можно создавать правила позиционирования элементов, например:

- разместить элемент Button в 20 точках от верхнего края экрана;
- разместить элемент Label по центру экрана;
- разместить Slider на 10 точек правее Button.

Примечание Обычно, размеры и расстояния, связанные с экранами, принято измерять в пикселях в то время, как я говорю о каких-то непонятных точках. Пока что воспринимайте точки именно как пиксели, а далее я подробно расскажу, в чем состоит их отличие, и почему в iOS используется такая, на первый взгляд, странная система позиционирования.

Опираясь на созданные ограничения, Auto Layout подстраивает размеры и позиции элементов, обеспечивая при этом возможность унифицированной разработки интерфейса. Грубо говоря, вы размещаете элементы, задаете правила их позиционирования и видите примерно одинаковую картинку на всех устройствах. Auto Layout – это очень функциональный механизм с большими возможностями, часть из которых будут рассмотрены в этой книге.

Сейчас мы воспользуемся Auto Layout для создания констрейнтов, определяющих позицию и размер слайдера:

1. Элемент Slider должен размещаться по центру вертикальной оси сцены.

Примечание Для того, чтобы понять, как именно будет центрироваться элемент, мысленно проведите на сцене вертикальную ось (сверху вниз) и определите точку, являющуюся ее центром. Горизонтальная линия, идущая по центру слайдера и делящая его на две равные части, должна проходить ровно через данную точку. В результате чего расстояния выше и ниже слайдера будут равны (рис. 1.12). Далее в книге мы более подробно поговорим о центрировании по вертикальной и горизонтальной осям.

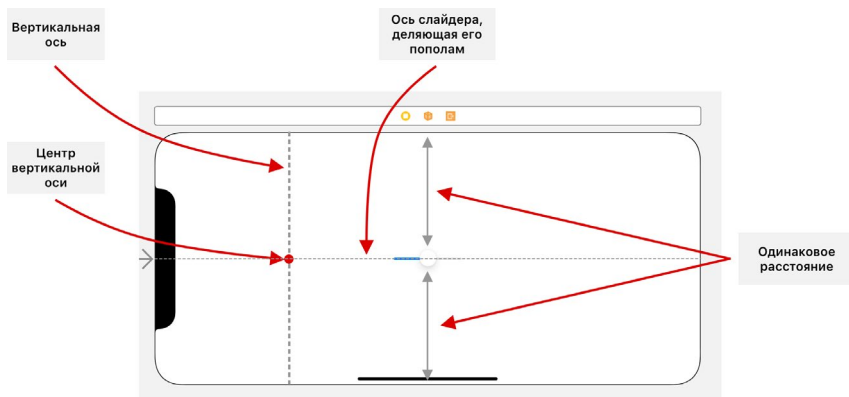


Рис. 1.12. Центрирование элемента по вертикальной оси

2. Слайдер должен иметь отступы в 20 точек слева и справа от боковых краев сцены.

Создадим констрейнты, реализующие указанные правила (рис. 1.13):

- ▶ Выделите слайдер в **Document Outline** или прямо на сцене.
- ▶ (1) В нижней части **Interface Builder** нажмите на кнопку **Align**.
- ▶ (2) В появившемся окне активируйте пункт **Vertically in Container**.
- ▶ (3) Нажмите на кнопку **Add 1 Constraint**.

Теперь, когда слайдер располагается в центре вертикальной оси сцены, разберемся, как задать отступы слева и справа.

В текущем состоянии сцена включает в себя два графических элемента: корневое представление (root View) и слайдер. Корневое представление – это подложка сцены, которая занимает всю доступную область независимо от того, на какой модели смартфона (или планшета) запущено приложение. Слайдер же входит в состав корневого представления (это видно по структуре сцены в **Document Outline**).

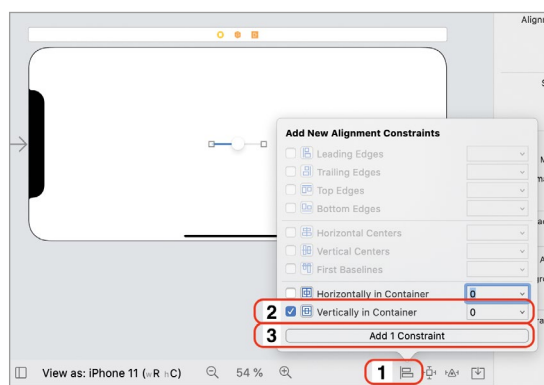


Рис. 1.13. Создание ограничения

Констрейнты позволяют задавать отступы между границами графических элементов. А раз root View, равно как и слайдер, является обычным графическим элементом, это позволяет создать констрейнт, задающий отступ между ними, даже несмотря на то, что слайдер входит в состав корневого представления. Создав констрейнты, мы зададим отступы от краев слайдера до краев вью, однако слайдер при этом будет находиться внутри представления.

- ▶ Выделите слайдер в **Document Outline** или прямо на сцене.
- ▶ В нижней части **Interface Builder** нажмите на кнопку **Add New Constraints**, расположенную рядом с **Align** (рис. 1.14).

Всплывающее окно позволяет создавать различные типы ограничений. В частности, четыре поля, расположенные в верхней части, определяют отступы с каждой из сторон (сверху, слева, справа и снизу). Отступ задается между текущим элементом и ближайшим к нему с указанной разработчиком стороны. Ближайшим к слайдеру с каждой из сторон является корневое представление, внутри которого он и расположен.

- ▶ Укажите значение 20 в левом и правом полях (рис. 1.15). При этом прерывистые линии слева и справа станут сплошными, указывая на создаваемые ограничения.

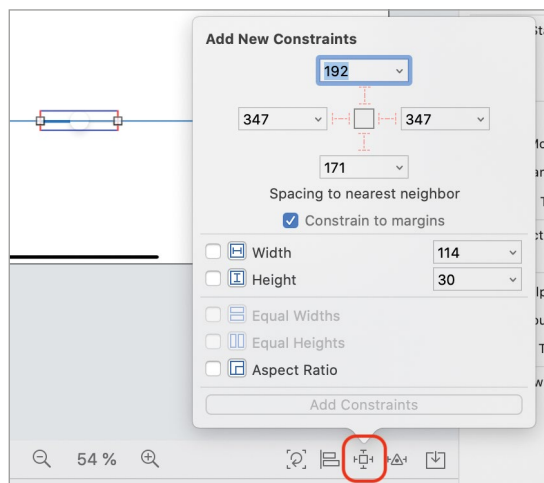


Рис. 1.14. Создание constraint

- Нажмите кнопку **Add 2 Constraints** или клавишу **Enter** на клавиатуре.

После проделанных действий слайдер растянулся по всей ширине сцены, оставив отступы в 20 точек слева и справа от границ корневого представления (рис. 1.16). Благодаря Auto Layout, слайдер сохранит свою позицию на любом устройстве и при любой ориентации.

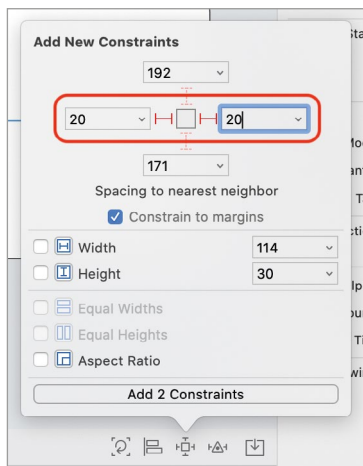


Рис. 1.15. Создание ограничений

Как вы могли заметить на рисунке 1.16, визуально расстояние слева и справа все же немного отличаются. Дело в том, что внутри корневого представления есть особая зона – **Safe Area**.

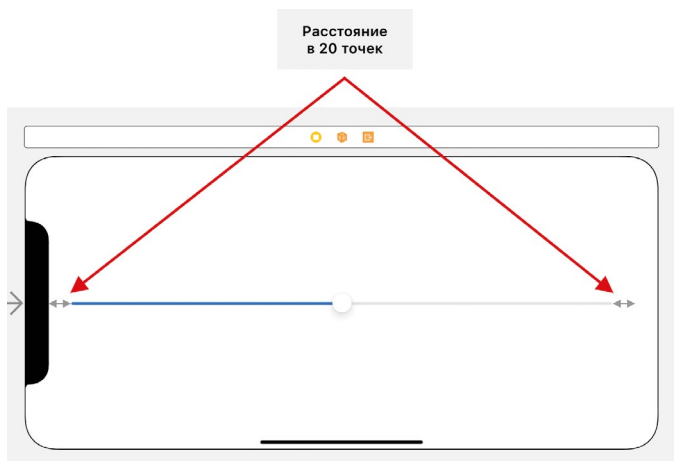


Рис. 1.16. Созданные ограничения

Safe Area – это так называемая «безопасная область», исключающая элементы корпуса и статус-бар устройства. Safe Area была добавлена в iOS 11 в связи с выходом iPhone X с его «нестандартным» экраном. Safe Area – это не отдельный графический элемент, а лишь специальная область внутри корневого представления, облегчающая позиционирование элементов на сцене без опасности их перекрытия с системными физическими и UI-элементами.

► Выделите Safe Area на панели **Document Outline**.

После того, как вы выделили Safe Area, соответствующая ей область выделилась прямо на сцене (рис. 1.17). При создании констрейнтов для слайдера расстояния определялись не от границ корневого представления, а от границ его Safe Area.

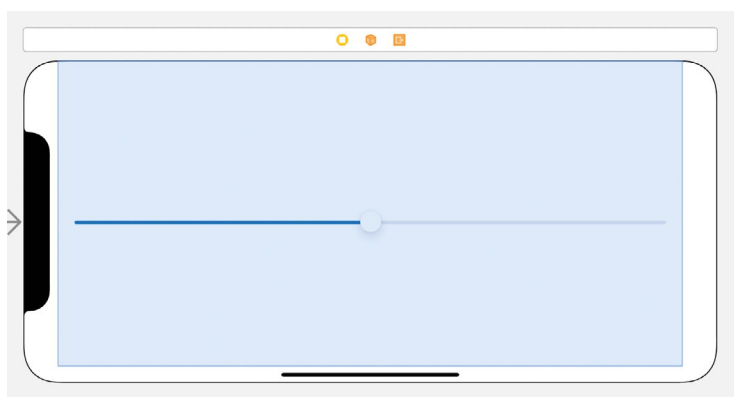


Рис. 1.17. Границы Safe Area на сцене

Кнопка и текстовая метка

Теперь перейдем к размещению на сцене оставшихся элементов.

- ▶ Убедитесь в том, что в **Project Navigator** выбран **Main.storyboard**.
- ▶ Откройте панель **Library** (с помощью кнопки на **Toolbar** или сочетанием клавиш **⇧Shift + ⌘Command + L**)
- ▶ Найдите в библиотеке объектов UI-элемент Button и разместите его ниже слайдера на сцене.
- ▶ Найдите в библиотеке объектов UI-элемент Label и разместите его ниже кнопки на сцене.

В конечном итоге на сцене должны разместиться слайдер, кнопка и текстовая метка (рис. 1.18).

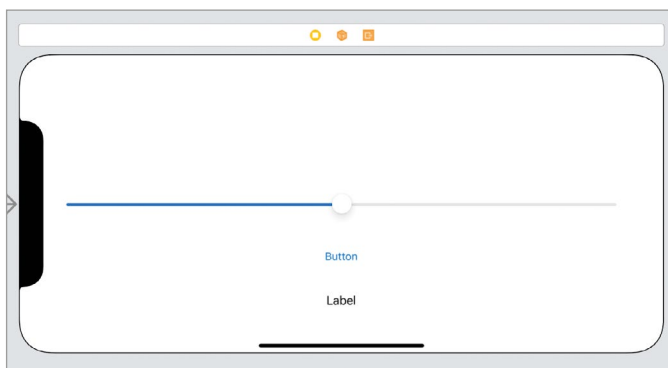


Рис. 1.18. Графические элементы на сцене

Для новых элементов также потребуется создать констрейнты (ограничения) для того, чтобы они сохраняли свои позиции при запуске на различных устройствах.

- ▶ Выделите кнопку на сцене.
- ▶ Нажмите кнопку **Align** в нижней части **Interface Builder**.
- ▶ Во всплывающем окне выберите пункт **Horizontally in Container** и нажмите кнопку **Add 1 Constraint**.

Теперь кнопка выровнена по центру горизонтальной оси.

Примечание Обратите внимание на то, что слайдер был выровнен по центру вертикальной оси, а кнопка – по центру горизонтальной. Вероятнее всего, на первых порах применение различных видов выравниваний могут вызывать у вас сложности. По этой причине я советую вам самостоятельно попробовать каждый из них, чтобы лучше разобраться в их отличиях.

Horizontally in Container – центрирование по горизонтали.

Вы можете мысленно провести горизонтальную линию от одного края сцены до другого. Центр графического элемента будет находиться точно посередине этой линии. Но при этом по высоте линия может находиться в любом месте.

При создании такого ограничения расстояния справа и слева от него всегда одинаковы.

Vertically in Container – центрирование по вертикали.

Вы можете мысленно провести вертикальную линию от верхнего края сцены к нижнему. Центр графического элемента будет находиться посередине этой линии. Но при этом по ширине линия может находиться в любом месте.

При создании такого ограничения расстояния сверху и снизу от него всегда одинаковы.

Для того, чтобы установить элемент по центру сцены по обеим осям, необходимо создать два ограничения, выбрав оба пункта.

Зафиксируем вертикальный отступ от кнопки до слайдера:

- ▶ Выделите кнопку на сцене.
- ▶ В нижней части **Interface Builder** нажмите на кнопку **Add New Constraints**, расположенную рядом с **Align**.
- ▶ Во всплывающем окне в верхнем текстовом поле укажите значение 30 (рис. 1.19).

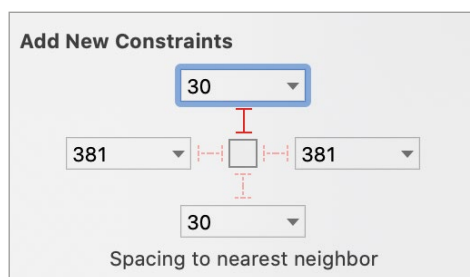


Рис. 1.19. Отступ от верхнего элемента

Таким образом, будет создан отступ от слайдера, который находится выше кнопки.

- ▶ Нажмите кнопку **Add 1 Constraint** или клавишу **Enter** на клавиатуре.

После создания ограничения кнопка на сцене автоматически сдвинется на требуемую позицию, а между ней и слайдером отобразится синяя линия, обозначающая данное ограничение. Благодаря ограничению, кнопка всегда будет находиться в 30 точках от слайдера, тем самым обозначив свою позицию для любого устройства.

Все созданные вами ограничения отображаются в составе сцены в **Document Outline** в разделе **Constraints** (рис. 1.20). Вы можете щелкнуть по любому из них и при необходимости отредактировать их с помощью панели **Inspectors** (рис. 1.21).

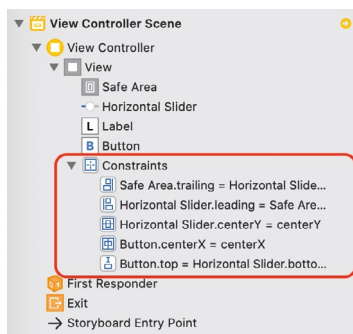


Рис. 1.20. Список созданных ограничений

Обратите внимание, что в правом верхнем углу панели **Document Outline** отображается желтый круг со стрелкой внутри ➡. Так Xcode сообщает вам, что в проекте есть не критические замечания к позиционированию элементов на сцене.

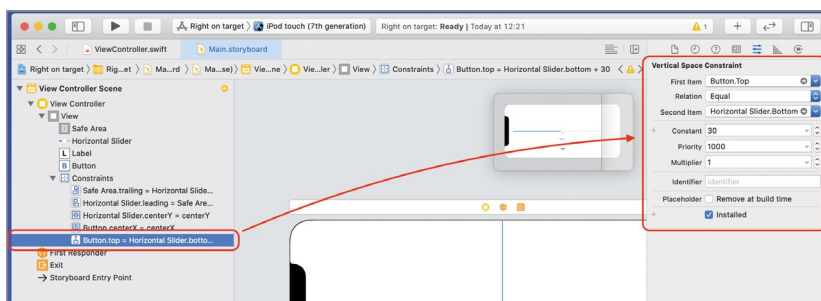


Рис. 1.21. Редактирование созданного ограничения

► Нажмите на желтый кружок в **Document Outline**.

Теперь в панели приведен список замечаний (рис. 1.22), состоящий всего из одного пункта, относящегося к текстовой метке. В нем говорится о том, что для метки не созданы ограничения (constraints) и это может плохо повлиять на внешний вид интерфейса.

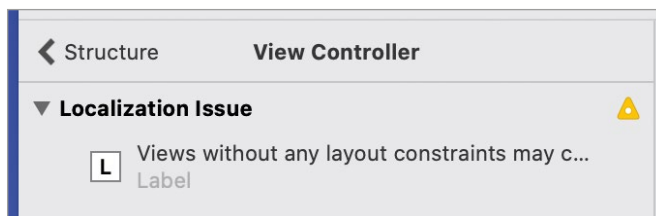



Рис. 1.22. Замечания к расположению элементов

Если нажать на желтый треугольник справа от текста ошибки, то Xcode предложит автоматически создать необходимые ограничения, но сейчас не будем делать этого. Исправим данный недостаток самостоятельно.

Создадим ограничение для центрирования текстовой метки по горизонтальной оси:

- ▶ В **Document Outline** вернитесь к списку элементов сцены с помощью кнопки «< **Structure**».
- ▶ Выделите текстовую метку на сцене или в **Document Outline**.
- ▶ Нажмите кнопку **Align** в нижней части **Interface Builder**.
- ▶ В появившемся окне выделите пункт **Horizontally in Container** и нажмите кнопку **Add 1 Constraint**.

Теперь в **Document Outline** отображается красный круг  вместо желтого, как это было ранее. Так Xcode сообщает нам, что в проекте есть критические ошибки позиционирования элементов.

- ▶ Нажмите на красный круг в **Document Outline**.

В появившемся списке указано, что для элемента Label необходимо создать дополнительное ограничение для позиционирования по оси Y (рис. 1.23). И Xcode готов сделать это в автоматическом режиме.

- ▶ Нажмите на красный круг, расположенный справа.
- ▶ В появившемся окне щелкните по кнопке **Add Missing Constraints** (рис. 1.24).

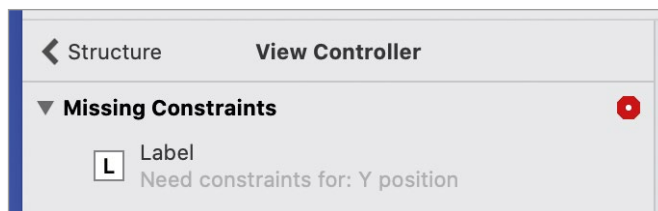


Рис. 1.23. Замечания к расположению элементов

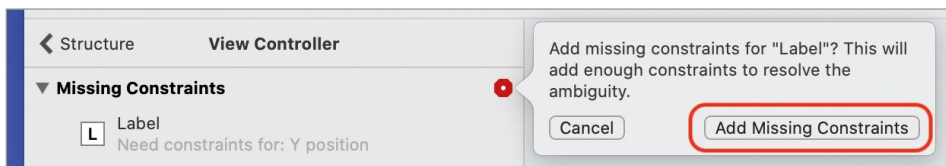


Рис. 1.24. Создание нового ограничения

Теперь все необходимые ограничения созданы, и интерфейс приложения будет автоматически подстраиваться под экран устройства, на котором запущено приложение. Но последнее ограничение для текстовой метки было создано в автоматическом режиме, и вполне вероятно, что отступ между кнопкой и слайдером отличается от отступа между меткой и кнопкой.

- ▶ Вернитесь к списку элементов сцены в **Document Outline**.
- ▶ Внутри элемента **Constraints** щелкните по ограничению, которое указывает на отступ текстовой метки от кнопки.

В моем случае этот элемент называется «**Label.top = Button.bottom + 49**», т.е. «**Верхняя координата метки - это нижняя координата кнопки + 49 пойнтов**» (рис. 1.25).

Примечание В вашем случае отступ может быть другим, так как значение констрейнта зависит от того, где вы разместили текстовую метку при перетаскивании ее из библиотеки на сцену.

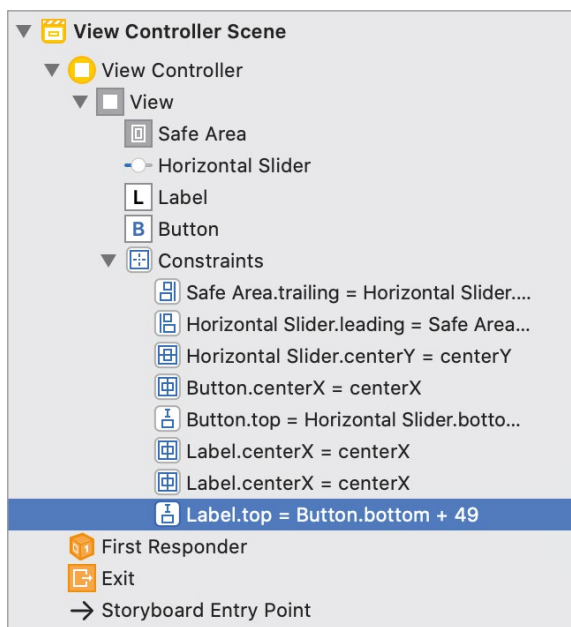


Рис. 1.25. Ограничение текстовой метки

Изменим отступ, сделав его равным 30 (рис. 1.26):

- ▶ На панели **Inspectors** измените значение поля **Constant** на 30.

После совершенных действий текстовая метка на сцене автоматически изменит свое расположение, встав ровно на 30 точек ниже кнопки.

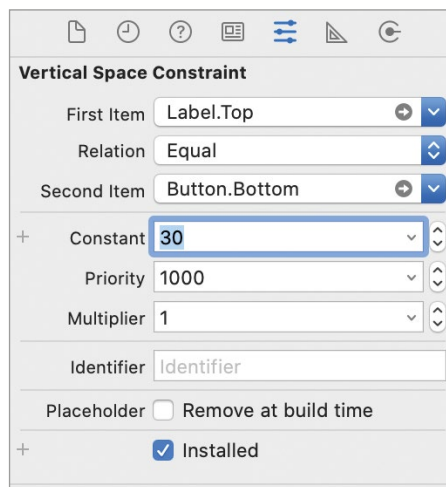


Рис. 1.26. Настройка отступа текстовой метки

Запуск симулятора

► Запустите сборку проекта и дождитесь загрузки симулятора (рис. 1.27).

Запущенное приложение выглядит не совсем так, как предполагалось, поскольку ориентация симулятора и ориентация, указанная в настройках проекта, не совпадают.

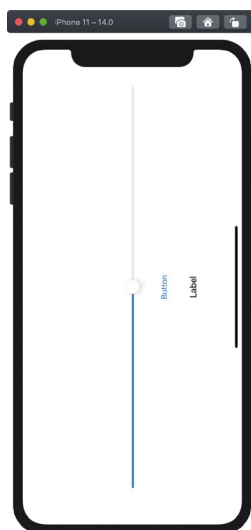


Рис. 1.27. Приложение, запущенное на симуляторе

- ▶ Измените ориентацию симулятора на альбомную. Для этого используйте сочетание клавиш **⌘ Command + стрелка влево**, **⌘ Command + стрелка вправо** (соответствует пунктам меню **Device > Rotate Left** и **Device > Rotate Right**).

Примечание Независимо от того, какую горизонтальную ориентацию (левую или правую) принимает устройство, сцена соответствующим образом переворачивается. Это происходит благодаря тому, что при настройке ориентации приложения были выбраны пункты **Landscape Left** и **Landscape Right**.

Финальные правки интерфейса

Осталось внести некоторые изменения во внешний вид графических элементов.

- ▶ Выделите root View и откройте панель **Attributes Inspector**.
- ▶ Измените цвет фона на **System Yellow Color** (свойство **Background**).
- ▶ Выделите кнопку и откройте панель **Attributes Inspector**.
- ▶ С помощью свойства **Title** измените текст элемента на «**Проверить**».
- ▶ Измените начертание текста на **Bold** и размер шрифта на 20 (свойство **Font**).
- ▶ Измените цвет текста на **System Indigo Color** (свойство **Text Color**).
- ▶ Выделите слайдер и откройте панель **Attributes Inspector**.
- ▶ Измените цвет полосы до бегунка на **System Pink Color** (свойство **Min Track**).
- ▶ Измените цвет бегунка на **System Pink Color** (свойство **Thumb Tint**).
- ▶ Добавьте иконки слева и справа от слайдера (**01.square.fill** для свойства **Min Image** и **50.square.fill** для **Max Image**).

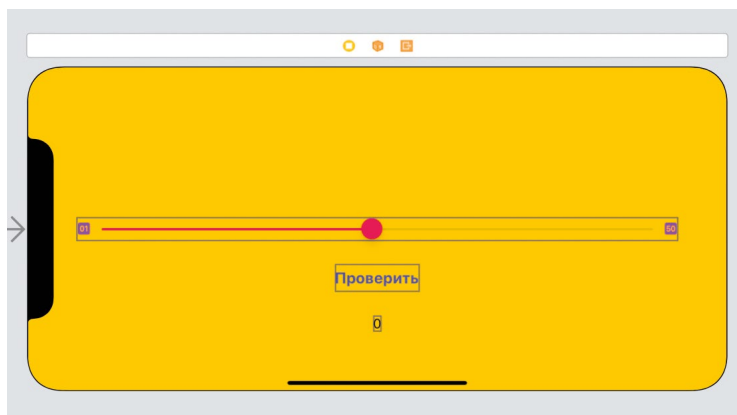


Рис. 1.28. Готовый интерфейс приложения в Interface Builder

Примечание В iOS 13 Apple добавили специальный набор иконок SF Symbols, которые и были использованы в качестве значков слева и справа от слайдера. Данный пакет иконок содержит более 2400 настраиваемых символов (во второй версии пакета) и прекрасно выглядит с системным шрифтом iOS.

Для удобства поиска и настройки символов вы можете скачать одноименное приложение для macOS по ссылке: <https://developer.apple.com/sf-symbols/>

- ▶ Измените цвет иконок на **System Purple Color** (свойство **Tint**).
- ▶ Выделите текстовую метку.
- ▶ Измените текст метки на 0.

На этом работа с графическим интерфейсом завершена, все элементы расположены на своих местах и готовы к выполнению задач по назначению (рис. 1.28).

1.3 Программирование бизнес-логики

Связь кода и элементов на сцене

В процессе игры программный код приложения должен взаимодействовать с графическим интерфейсом, и вы уже неоднократно делали это, связывая элементы с аутлет-свойствами, а события (например, нажатие на кнопку) с экшн-методами. Сейчас нам требуется обеспечить доступ к изменению текста в Label (для того, чтобы вывести загаданное число) и получению выбранного значения из Slider. Для этого необходимо:

1. объявить outlet-свойства в классе **ViewController** (данный класс обслуживает работу сцены, где располагаются элементы) для элементов Slider и Label;
2. реализовать action-метод, который будет вызван по нажатию на кнопку.

Начнем с решения первой задачи.

Примечание В дальнейшем, для того, чтобы упростить материал, я буду допускать использование понятий аутлет (outlet-свойства) и экшн (action-метод), написанные кириллицей.

В первой книге вы связывали аутлеты и элементы на сцене, разделив **Project Editor** на две независимые рабочие области. В одной открывался **Interface Builder**, а в другой — файл с кодом класса **ViewController**. Вопрос решался простым перетягиванием от графического элемента в редактор кода, или наоборот.

Примечание Если вы не понимаете, о чем идет разговор, то вернитесь к главе по разработке приложения на основе фреймворка UIKit в книге «Swift. Разработка приложений под iOS, iPadOS и macOS» (не менее, чем 6-е издание).

В этот раз мы попробуем иной способ: сперва создадим два ауллета в коде класса, а уже потом свяжем их с элементами на сцене с помощью панели **Inspectors**.

- ▶ Откройте файл **ViewController.swift** и добавьте в класс **ViewController** свойства из листинга 1.1.

ЛИСТИНГ 1.1

```
@IBOutlet var slider: UISlider!  
@IBOutlet var label: UILabel!
```

Прошу заметить, что в качестве типа для ауллеров использован опционал с неявным извлечением значения. Я хотел бы обратить ваше внимание на два момента, связанных с этим.

Во-первых, использование именно опционала в данном случае является обязательным, иначе приложение «упадет» в процессе запуска. Дело в том, что в момент создания класса **ViewController** ауллеры еще не имеют значений, их привязка к элементам на сцене происходит несколько позже. Но экземпляр класса не может иметь неопциональных свойств без значений, отсюда и возникает критическая ошибка.

Во-вторых, неявное извлечение позволяет нам не извлекать значение свойств далее в коде, а использовать непосредственно имена параметров **slider** и **label**, вместо **slider!** и **label!** (или других вариантов извлечения значений).

Примечание Напомню, что для создания ауллеров используется префикс **@IBOutlet**. Графический элемент слайдер представлен классом **UISlider**, а метка — классом **UILabel**.

Теперь свяжем элементы с ауллетами (рис. 1.29):

- ▶ Откройте файл **Main.storyboard**.
- ▶ (1) Выделите слайдер на сцене или в **Document Outline**.
- ▶ (2) Откройте панель **Connections inspector**. Она служит для создания различных видов связей, в том числе с ауллетами и экшннами.
- ▶ (3) В разделе **Referencing Outlets** найдите пункт **New Reference Outlet**.
- ▶ (4) Перетащите серый кружок напротив строки **New Reference Outlet** на желтый значок (ссылка на **View Controller**) над сценой.
- ▶ В появившемся окне выберите **slider**. Данный пункт указывает на одноименное свойство в классе **ViewController**.

После этого в разделе **Referencing Outlets** появится информация о созданной связи (рис. 1.30).

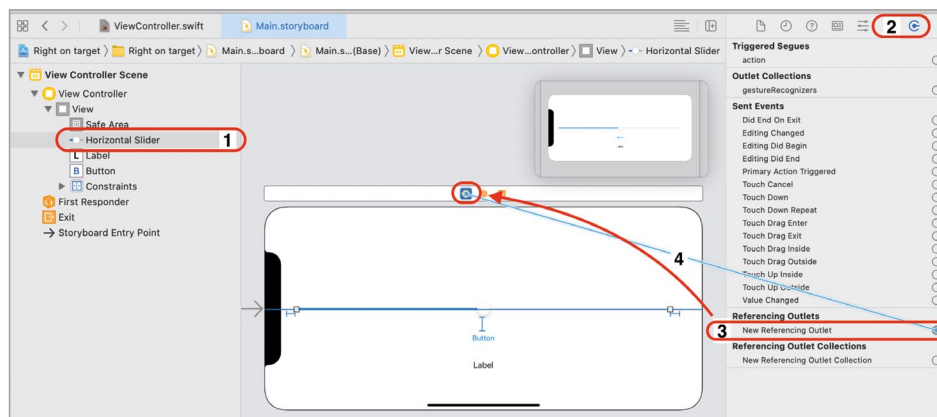


Рис. 1.29. Создание связи UI-элемента с аулетом

- По аналогии со слайдером, свяжите свойство **label** класса **ViewController** с текстовой меткой.

Теперь графические элементы на сцене связаны с outlet-свойствами класса **ViewController**, а значит вы можете работать с ними прямо из кода, изменяя или получая их значения.



Рис. 1.30. Созданная связь элемента с аулетом

Если открыть файл **ViewController.swift**, то слева от созданных аулетов вместо номеров строк вы увидите темно-серые закрашенные круги, указывающие на созданную связь (рис. 1.31). При нажатии на них появится всплывающее окно с информацией об элементе, на который ссылается каждое из свойств.

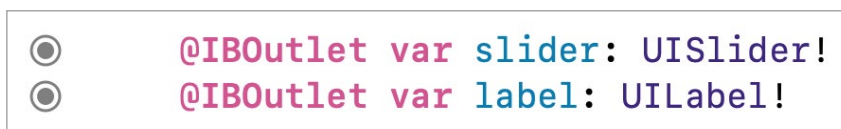


Рис. 1.31. Аулеты, связанные с графическими элементами

Настройка слайдера

По умолчанию слайдер позволяет выбрать значение типа **Float** от 0 до 1. Т.е. его крайнее левое положение соответствует нулю, крайнее правое – единице. Все промежуточные значения – это дробные числа в указанном диапазоне. В нашем случае, в соответствии с правилами, необходимо, чтобы слайдер позволял выбирать целые числа от 1 до 50. С этой целью внесем изменения в настройки графического элемента (рис. 1.32):

- ▶ Выделите слайдер на сцене.
- ▶ Откройте **Attributes Inspector**.
- ▶ Измените значения полей **Value** (исходное значение слайдера) на 25, **Minimum** на 1 и **Maximum** на 50.

Теперь, обращаясь к аутлету **slider** для определения выбранного пользователем числа, мы будем получать значение типа **Float** в диапазоне от 1 до 50.

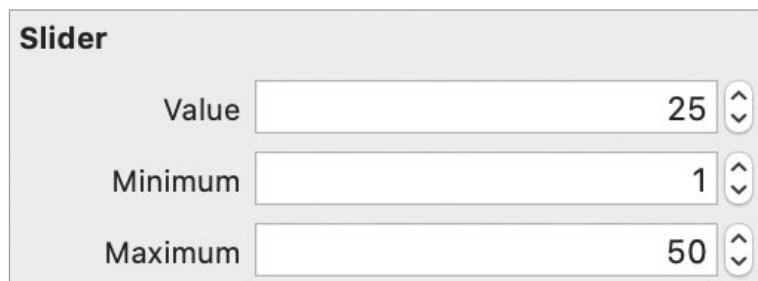


Рис. 1.32. Изменение параметров слайдера

Реализация бизнес-логики

Несмотря на то, что вы можете взаимодействовать с элементами на сцене (перемещать бегунок на слайдере и нажимать кнопку), сцена все еще не решает никаких реальных задач. Следующим этапом будет наполнение приложения жизнью, т.е. реализация бизнес-логики.

Примечание Бизнес-логика – это логика работы приложения, ее внутренние алгоритмы и правила. Если программа решает поставленную перед ней задачу, то говорят, что в ней реализована бизнес-логика.

К бизнес-логике не относится логика работы интерфейса приложения. То, какие в вашем приложении появляются окна, как кнопки меняют цвет при нажатии и т.д. – все это вопросы дизайна и интерфейса. А вот что именно происходит по нажатию в приложении, например, отправка данных на сервер, расчет математических величин, подсчет заработанных очков – это и есть бизнес-логика.

Реализация бизнес-логики = реализация алгоритма приложения.

В первую очередь, в классе **ViewController** необходимо объявить три целочисленных свойства (листинг 1.2).

ЛИСТИНГ 1.2

```
// загаданное число
var number: Int = 0
// раунд
var round: Int = 0
// сумма очков за раунд
var points: Int = 0
```

Изначально, сразу после запуска приложения, загаданное число (хранится в свойстве **number**) равняется 0, что является довольно серьезной проблемой. Дело в том, что для начала игры нам потребуется каким-то образом загадать число, записать его в **number** и отобразить в текстовой метке.

Для решения указанной задачи мы можем пойти одним из следующих путей:

1. Сгенерировать число еще до того, как приложение появится на экране. В этом случае пользователь сможет начать игру сразу после того, как приложение отобразит интерфейс.
2. Сгенерировать число по нажатию кнопки на сцене. В этом случае пользователь сперва увидит интерфейс приложения с нулем в метке, после чего нажмет на кнопку, и уже только после этого начнет играть.

Для реализации первого варианта вам пока еще не хватает знаний, так как прежде требуется познакомиться с понятием «жизненный цикл выю контроллера», поэтому воспользуемся вторым вариантом.

Примечание В дальнейшем мы рассмотрим реализацию первого варианта решения проблемы.

Прежде чем перейти к реализации бизнес-логики, определим методику подсчета очков. Очевидно, что чем ближе выбранное на слайдере число к загаданному, тем больше очков должно быть получено. В нашем случае, я предлагаю начислять очки по следующей формуле:

1. За угаданное число начисляется 50 очков.
2. Если загаданное число больше выбранного, то:
 $50 - \text{загаданное} + \text{выбранное}$
3. Если загаданное число меньше выбранного, то:
 $50 - \text{выбранное} + \text{загаданное}$

Примечание Вы можете использовать собственный алгоритм подсчета очков.

Задание Если вы чувствуете в себе силы, то самостоятельно реализуйте action-метод и свяжите его с кнопкой. Хочу отметить, что вы можете использовать и свой алгоритм работы. Главное, чтобы поставленная задача выполнялась, и пользователь мог проводить игры из 5 раундов, в конце которых видел количество заработанных очков.

В классе **ViewController** реализуем action-метод **checkNumber()**:

- ▶ Откройте файл **ViewController.swift** и включите в класс **ViewController** метод из листинга 1.3.

ЛИСТИНГ 1.3

```
@IBAction func checkNumber() {
    // если игра только начинается
    if self.round == 0 {
        // генерируем случайное число
        self.number = Int.random(in: 1...50)
        // передаем значение случайного числа в label
        self.label.text = String(self.number)
        // устанавливаем счетчик раундов на 1
        self.round = 1
    } else {
        // получаем значение на слайдере
        let numSlider = Int(self.slider.value.rounded())
        // сравниваем значение с загаданным
        // и подсчитываем очки
        if numSlider > self.number {
            self.points += 50 - numSlider + self.number
        } else if numSlider < self.number {
            self.points += 50 - self.number + numSlider
        } else {
            self.points += 50
        }
    }
    if self.round == 5 {
        // выводим информационное окно
        // с результатами игры
        let alert = UIAlertController(
            title: "Игра окончена",
            message: "Вы заработали \(self.points) очков",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "Начать заново", style:
.default, handler: nil))
        self.present(alert, animated: true, completion: nil)
        self.round = 1
        self.points = 0
    }
}
```

```
    } else {  
        self.round += 1  
    }  
    // генерируем случайное число  
    self.number = Int.random(in: 1...50)  
    // передаем значение случайного числа в label  
    self.label.text = String(self.number)  
}  
}
```

Примечание Использование **self** в данном листинге не является обязательным. Ключевое слово используется лишь для того, чтобы вы обращали внимание на том, к какому именно свойству происходит обращение.

В методе **checkNumber** происходит проверка текущего раунда, и если он равен **0** (сразу после запуска приложения), то выполняется первичная генерация загаданного числа.

Теперь свяжем событие нажатия на кнопку на сцене с вызовом метода **checkNumber()**. Для этого:

- ▶ Перейдите к файлу **Main.storyboard**.
- ▶ Выделите кнопку.
- ▶ Откройте панель **Connections Inspector**.
- ▶ В разделе **Sent Events** найдите событие **Touch Up Inside** и перетяните его на желтый значок **View Controller** над сценой.
- ▶ В появившемся списке выберите элемент **checkNumber**.

Событие **Touch Up Inside** срабатывает, когда пользователь заканчивает взаимодействие с графическим элементом на экране, т.е. когда после совершенного нажатия на кнопку он отрывает палец от экрана (прерывает касание). В момент окончания касания палец должен находиться в пределах кнопки. Таким образом, если бы вы нажали на кнопку, не отрывая палец от экрана передвинули его за пределы кнопки и отпустили, то сработало бы событие **Touch Up Outside** (оно также присутствует в разделе **Sent Events**).

Примечание В следующих главах мы подробно разберем, что такое события и как они работают в операционной системе и ваших программах.

- ▶ Запустите сборку проекта и попробуйте провести в игре несколько раундов.

Приложение функционирует в точности так, как мы этого хотели: все элементы выполняют свои задачи, а метод **checkNumber** корректно реализует бизнес-логику (алгоритм работы приложения). Но на этом разработка приложения еще не окончена, мы продолжим его доработку в следующих главах.

Глава 2.

Введение в шаблон проектирования MVC

В этой главе вы:

- получите основные знания о шаблонах проектирования, а также их важности для профессионального разработчика;
- откроете для себя популярный архитектурный шаблон MVC (Model-View-Controller);
- узнаете, как распределяются элементы реального проекта по ролям в соответствии с шаблоном проектирования MVC.



Далее мы продолжим работу над программой «Right on target». Скачать ее можно, перейдя по ссылке: <https://swiftme.ru/listings21>

Вы реализовали уже несколько несложных iOS-приложений, но все еще довольно мало знаете об особенностях их внутреннего устройства и принципах функционирования, а также способах их взаимодействия с операционной системой. Начиная с этой главы, вы начнете углублять свои знания в этой области. В качестве объекта исследования мы будем использовать созданную ранее игру «**Right on target**».

2.1 Архитектурные шаблоны проектирования

Пожалуй, самой фундаментальной книгой, рассматривающей вопросы организации кода и создания структуры приложений (не только для iOS, но и для любых других платформ), является книга «Банды четырех» (Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес) под названием «Приемы объектно-ориентированного проектирования. Паттерны проектирования», написанная в

далеком 1994 году. В ней систематизированы имеющиеся на тот момент знания о шаблонах проектирования (также называемых паттернами), а также приведены конкретные примеры и рекомендации по их использованию. Если вы нацелены на долгую и плодотворную работу в качестве разработчика, рано или поздно вы обязательно ее прочтете и попробуете в деле большинство из описанных там «рецептов».

Примечание С некоторыми из шаблонов (Singleton, Delegation и MVC) вы уже встречались в первой книге.

Шаблон в общем виде представляет собой набор некоторых инструкций, выполняя которые (следуя шаблону) обеспечивается решение определенной проблемы.

Шаблон проектирования – это описание программной конструкции, позволяющей решить определенную типовую проблему, возникающую в ходе разработки приложения. Шаблоны описывают самые разные аспекты работы над программным кодом приложений: создание объектов (порождающие шаблоны), их взаимодействие (поведенческие шаблоны) или раскрывают вопросы построения архитектуры приложений в целом (структурные, или архитектурные шаблоны). Именно об архитектурных шаблонах и пойдет речь в этой главе.

Архитектурные шаблоны базируются на **принципе разделения полномочий**, который предполагает, что:

1. каждый элемент проекта должен выполнять задачи только в рамках строго отведенной ему роли;
2. элементы проекта должны взаимодействовать с другими элементами только в соответствии с установленными правилами, обеспечивающими максимальную независимость отдельных элементов друг от друга.

Непосредственно сами роли и правила взаимодействия описываются в архитектурных шаблонах проектирования. Другими словами, архитектурные шаблоны говорят о том, на какие группы (по предназначению) необходимо разделить ваши классы, структуры и перечисления и каким образом организовать их взаимодействие. Архитектурные шаблоны позволяют избежать ситуации, при которой, к примеру, код, который должен менять цвет фона, также будет отвечать за связь с сервером.

Использование шаблонов проектирования вносит ясность и прозрачность в ваш код. Благодаря этому, вы всегда знаете, где находится тот или иной функциональный элемент.

Проект, созданный без оглядки на шаблоны, с большой долей вероятности, состоит из «плохого» программного кода.

Вы, конечно же, можете писать несложные программы, вообще не обращаясь к шаблонам, и они, вполне вероятно, будут корректно работать. Но как только дело дойдет до отладки, поддержки или доработки, у вас могут возникнуть большие проблемы.

2.2 Шаблон проектирования MVC

Существует большое количество различных архитектурных шаблонов, используемых iOS-разработчиками, но каждый из них преследует одну цель – создание хорошей архитектуры ваших приложений.

Примечание О том, что такое «хорошая архитектура», и какие к ней предъявляются требования, мы уже говорили во вступлении к первой части этой книги.

Как было сказано ранее, архитектурный шаблон определяет набор функциональных ролей, по которым необходимо распределить элементы, а также правила взаимодействия этих элементов. Каждый элемент программы (класс, структура, перечисление, функция и т.д.) должен относиться к одной роли и в соответствии с ней реализовывать свою функциональность.

Одним из наиболее популярных архитектурных паттернов является **MVC** (Model-View-Controller), который предлагает использовать Apple в ваших проектах. Данный принцип построения архитектуры, если так можно сказать, глубоко интегрирован в среду разработки Xcode. На его основе созданы десятки тысяч приложений для iOS, iPadOS и macOS.

Примечание Сама концепция MVC появилась в далеком 1978 году и в процессе развития языков программирования продолжала эволюционировать. Сегодня MVC имеет несколько вариантов реализации, незначительно отличающихся друг от друга. В этой книге мы будем говорить и использовать MVC от Apple.

Шаблон MVC предусматривает выделение в проекте трех ролей (или трех групп элементов):

- Model – модель;
- View – представление или отображение;
- Controller – контроллер.

Именно по первым буквам названий этих групп элементов и носит свое имя данный шаблон.

Любой элемент, который вы создаете в ходе работы над приложением, должен быть отнесен к одной из указанных групп.

Model — модель

Model отвечает за данные и реализацию бизнес-логики. Другими словами, к Модели относятся основные сущности и код, который их обрабатывает и реализует логику работы приложения, например, сохраняет информацию в базу данных и производит различные вычисления.

Рассмотрим пару примеров.

В приложении «Заметки» к данным могли бы относиться непосредственно сами заметки (сущность «Заметка»), а к бизнес-логике – код, обеспечивающий хранение (запись и загрузка) заметок в локальном хранилище (базе данных).

В приложении «Каталог вакансий» к данным могли бы относиться сущности «Вакансия», «Работодатель», «Соискатель», а к бизнес-логике – код, который обеспечивает отправку этих данных на сервер и получение обратно.

View — представление

View отвечает за графическое представление данных, а также за взаимодействие с пользователем. View имеет представление только о том, каким образом данные из Модели должны отображаться на экране, и как пользователь может взаимодействовать с ними. Проще говоря, View – это то, что видит пользователь. Как бы это грубо не звучало, но View должно быть глупым. View показывает то, что ему передал Controller, а сам View, в свою очередь, передает в Controller информацию о действиях пользователя (например, о нажатиях на элементы интерфейса).

В программе «Заметки» к View будет относиться код, который:

- выводит список заметок на экран в виде таблицы;
- выводит форму создания/редактирования заметки;
- обеспечивает переход к странице создания/редактирования заметки и обратно.

Для View совершенно не имеет значения, какие именно заметки будут храниться в Модели. Их список всегда будет отображаться в табличном виде, а для редактирования всегда будет использоваться одна и та же форма.

В приложении «Каталог вакансий», по аналогии с «Заметками», к View будут относиться элементы, обеспечивающие отображение и редактирование сущностей, а также переход между ними.

Примечание Будьте внимательны! Слово View используется как для обозначения роли в составе MVC, так и для указания на представления графических элементов в Xcode. При необходимости я буду уточнять, о каком именно View идет речь.

Controller — контроллер

Controller – это связующее звено, своеобразный клей, между Model и View. Контроллер реализует две основные функции:

- принимает от View команды на обновление данных и передает их в Model;
- получает от Model извещения об изменении данных и отправляет их во View для отображения новых, обновленных.

В приложении «Заметки» пользователь может отметить одну из заметок как удаленную прямо в интерфейсе приложения. При этом заметка удаляется с экрана, а информация об этом отправляется в Контроллер, и далее в Модель, где заметка уже удаляется из базы данных.

Если вернуться к примеру с приложением «Каталог вакансий», то при запуске приложения Модель в фоновом режиме загружает список актуальных вакансий, после чего передает информацию о них в Controller, который, в свою очередь, отправляет данные во View для их отображения на экране устройства.

Model-View-Controller

На рисунке 2.1 приведена схема шаблона MVC.

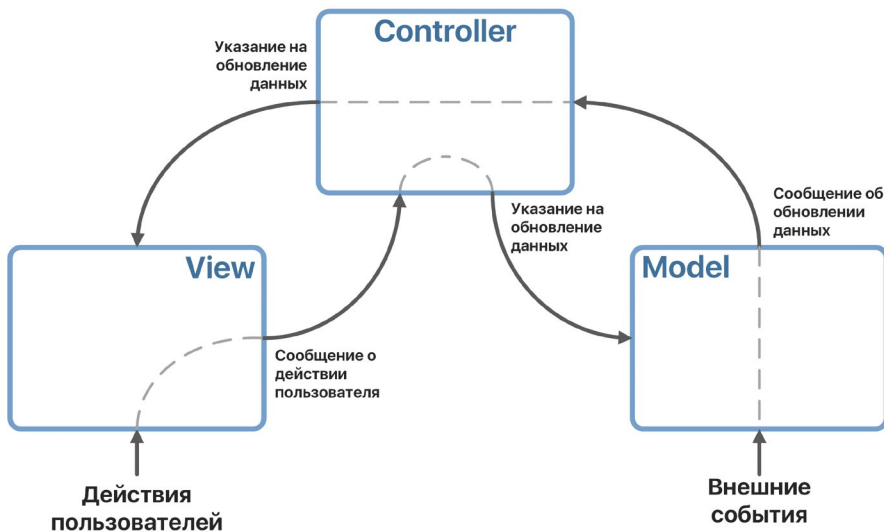


Рис. 2.1. Схема шаблона MVC

В общем случае, для того, чтобы наблюдать какие-либо изменения, должен быть совершен один из указанных вариантов действий:

- манипуляции пользователя с использованием графического интерфейса приложения;
- возникновение внешних событий (запрос с сервера, событие по таймеру и т.д.).

Примечание Не переживайте, если данный шаблон все еще не понятен вам. Дальше все встанет на свои места, когда мы поэлементно разберем приложение «Right on target», а также попрактикуемся в использовании MVC.

Шаблон MVC является лишь одним из возможных вариантов архитектуры ваших приложений, но знакомству именно с ним будет посвящен материал в данной книге. Одним из важнейших преимуществ данного шаблона является его низкая стоимость (не в рублях и долларах, а в человеко-часах) и высокая эффективность обслуживания программ. В любой момент времени вы знаете, где искать необходимую функциональность, чтобы внести в нее требуемые правки.

В дальнейшем, уже за пределами данной книги, помимо MVC вам предстоит познакомиться с такими архитектурными шаблонами, как MVP, MVVM, VIPER и многими другими.

2.3 Шаблон MVC в приложении «Right on target»

Добавим в наше знакомство с MVC немного практики и разберем, как же распределяются элементы реального уже реализованного нами проекта по ролям в MVC. Для этого вернемся к игре «**Right on target**».

- ▶ Откройте в Xcode проект «**Right on target**».

View

Все, что вы видите на экране своего устройства в процессе функционирования приложения, относится к View: root View, UI-элементы Slider, Button, Label, всплывающие окна и т.д. Также в состав View входят и storyboard-файлы. Все они определяют графический интерфейс и реакцию на действия пользователя.

Запомните: View **не должен** обрабатывать данные! View **не должен** хранить данные! View **должен** отображать данные!

Взглянем на элементы View своими глазами.

- ▶ Откройте файл **Main.storyboard**.

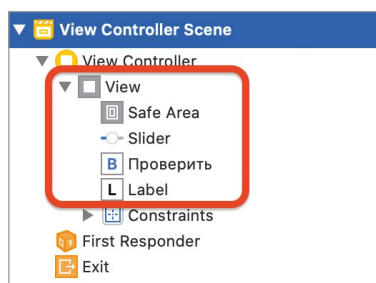


Рис. 2.2. Корневое представление на storyboard

Взгляните на панель **Document Outline**: root View и все вложенные в него элементы, относятся к View в MVC (рис. 2.2).

Также к View (в MVC) в данном проекте относятся файлы **Main.storyboard** и **LaunchScreen.storyboard**, которые вы можете найти в **Project Navigator**.

На рисунке 2.3 показана схема MVC, отражающая составные элементы View.

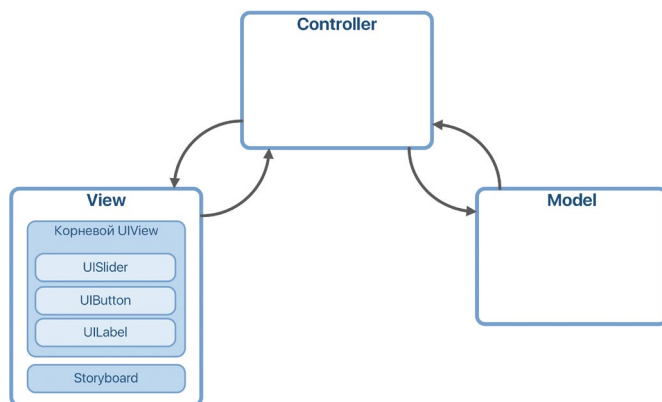


Рис. 2.3. Элементы приложения «Right on target» в составе MVC

Controller

Элемент View Controller реализует роль Контроллера (в MVC). Для того, чтобы увидеть его, взгляните на структуру сцены в **Document Outline** (рис. 2.4).

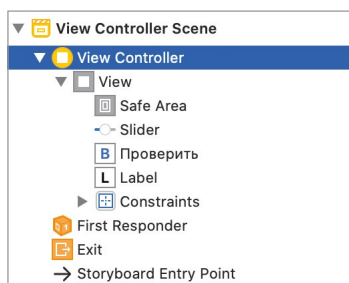


Рис. 2.4. View Controller в составе сцены

View Controller – это основа любой сцены iOS-приложения, написанного с использованием фреймворка UIKit. Это своего рода менеджер, который управляет всем, что происходит со сценой и на ней: отвечает за ее загрузку на экран устройства, следит за всеми событиями и элементами. Каждый отдельный рабочий экран (каждая сцена) имеет собственный View Controller, который организует вывод всех графических элементов и их взаимодействие с элементами Модели.

View Controller на сцене неразрывно связан с классом, являющимся потомком **UIViewController** (рис. 2.5):

- ▶ (1) Выделите **View Controller** в **Document Outline**.
- ▶ (2) Откройте панель **Identity inspector**.

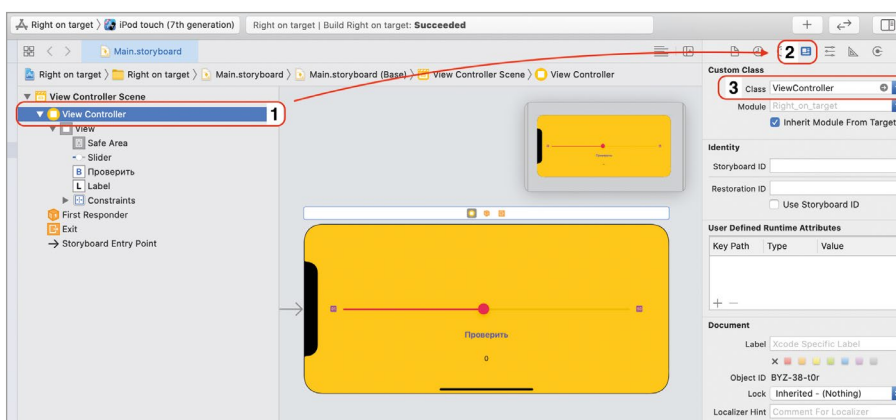


Рис. 2.5. Настройки View Controller

В поле **Class** указано имя класса **ViewController** (рисунок 2.5, пункт 3), который как раз и связан с элементом **View Controller** на сцене.

- ▶ Нажмите на серую стрелочку в поле **Class**, расположенную в его правой части.

Теперь в **Project Editor** открыт файл **ViewController.swift**, в котором описан класс **ViewController** (потомок **UIViewController**).

View Controller предоставляет очень широкие возможности контроля за работой сцены, с которыми мы начнем знакомиться в следующей главе. На рисунке 2.6 показано место View Controller в общей структуре MVC проекта.

Model

На данный момент в приложении «**Right on target**» отсутствует Модель, а элементы бизнес-логики включены в состав Контроллера (метод **checkNumber**). Такой подход допустим для очень простых приложений, но он в любом случае нарушает требования MVC.

Это было лишь первое знакомство с MVC от Apple на примере реального проекта. В последующих главах мы проведем небольшой рефакторинг кода, сделаем несколько доработок, реализуем Модель, в которую перенесем бизнес-логику из View Controller, и выпустим несколько новых версий приложения.

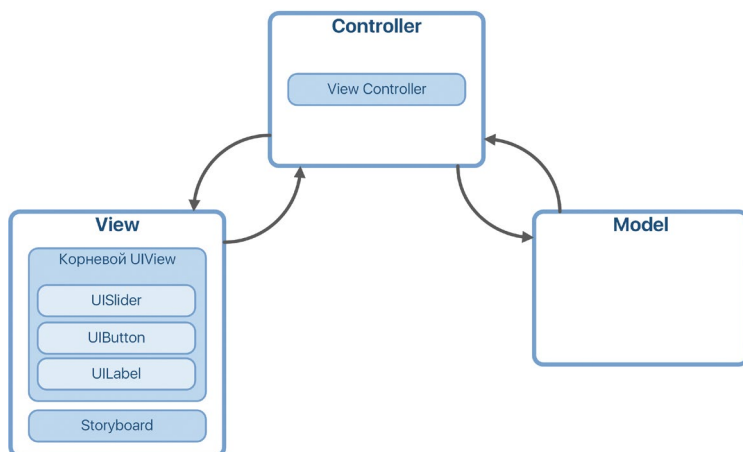


Рис. 2.6. Элементы приложения «Right on target» в составе MVC

Глава 3.

Введение в жизненный цикл View Controller

В этой главе вы:

- познакомитесь с понятием жизненного цикла;
- узнаете, что такое иерархия графических элементов;
- подробно рассмотрите жизненный цикл View Controller;
- изучите различные способы переходов между сценами и их основные отличия;
- произведете доработку приложения «Right on target», в том числе разработаете Модель (MVC).



Далее мы продолжим работу над программой «Right on target». Скачать ее можно, перейдя по ссылке: <https://swiftme.ru/listings21>

3.1 Понятие жизненного цикла

Жизненный цикл – это последовательная смена состояний объекта, начиная с момента его появления и до уничтожения.

Жизненный цикл есть у любых реальных и программных объектов. Например, обычная бытовая посудомоечная машина циклично принимает два состояния: «моет» и «не моет». При этом, перед тем, как машина примет очередное состояние, должны быть произведены определенные подготовительные работы (набор или откачка воды, проверка наличия моющего средства, ополаскивателя и т.д.). У вас как у пользователя машины есть возможность вмешаться в этот процесс: добавить моющее средство, добавить или убрать грязную посуду, разгрузить уже помытую посуду и т.д. Т.е. вы контролируете состояния, которые принимает объект, и в определенные моменты влияете на них.

На рисунке 3.1 показана схема смены состояний посудомоечной машины. Круги – это состояния, которые принимает машина, незакрашенные прямоугольники – это процессы, которые происходят при смене состояний, а закрашенные – процессы, в которых может участвовать пользователь.

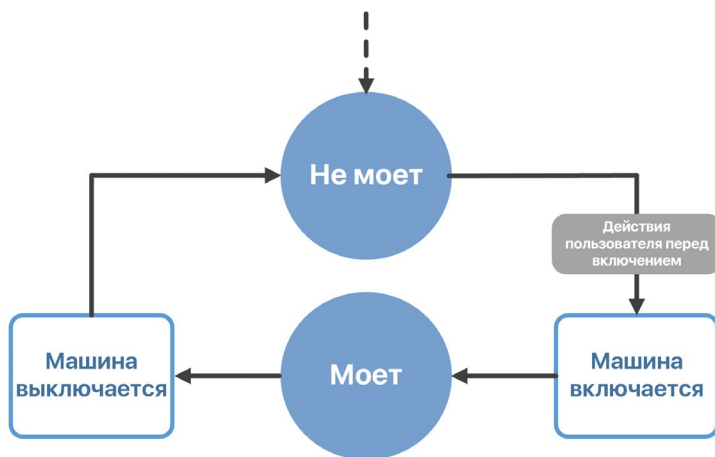


Рис. 3.1. Жизненный цикл посудомоечной машинки

Рассмотрим смену состояний поэтапно:

1. Изначально машина имеет состояние «Не моет» (соответствует верхнему кругу).
2. Пользователь производит необходимые действия перед мойкой: убирает или добавляет посуду, моющее средство, соль, ополаскиватель и нажимает кнопку запуска (соответствует закрашенному прямоугольнику справа).
3. Машина начинает накачивать воду и запускает процесс мойки (соответствует не закрашенному прямоугольнику справа).
4. Машина переходит в состояние «Моет» (соответствует нижнему кругу).
5. После того, как мойка завершена, машина производит подготовительные работы (откачивает воду, открывает дверцу) и выключается (соответствует не закрашенному прямоугольнику слева).
6. Машина вновь принимает состояние «Не моет».

3.2 Жизненный цикл View Controller

Жизненный цикл есть в том числе и у всех объектов в ваших программах: у самого iOS-приложения, экземпляров классов и структур, параметров, графических

элементов и т.д. Жизненный цикл в понимании iOS-разработчика описывает, как рождаются элементы, какие состояния они принимают в процессе функционирования, и как можно повлиять на смену этих состояний. Изучив жизненный цикл целостного приложения и его отдельных элементов, вы сможете понять, как именно оно функционирует.

Одним из важнейших элементов приложения является View Controller, который представляет своего рода сердце сцены. И у него точно также есть жизненный цикл, который описывает последовательную смену состояний сцены, которой он управляет.

Примечание Хочу акцентировать ваше внимание на том, что в данном случае происходит некоторая подмена понятий. Рассказывая о жизненном цикле View Controller, мы на самом деле подразумеваем жизненный цикл сцены, которой он управляет, а если быть еще точнее, иерархию представлений (View), входящих в состав сцены. Тем не менее, разработчики негласно договорились между собой употреблять термин «жизненный цикл View Controller».

На рисунке 3.2 показана схема жизненного цикла View Controller.

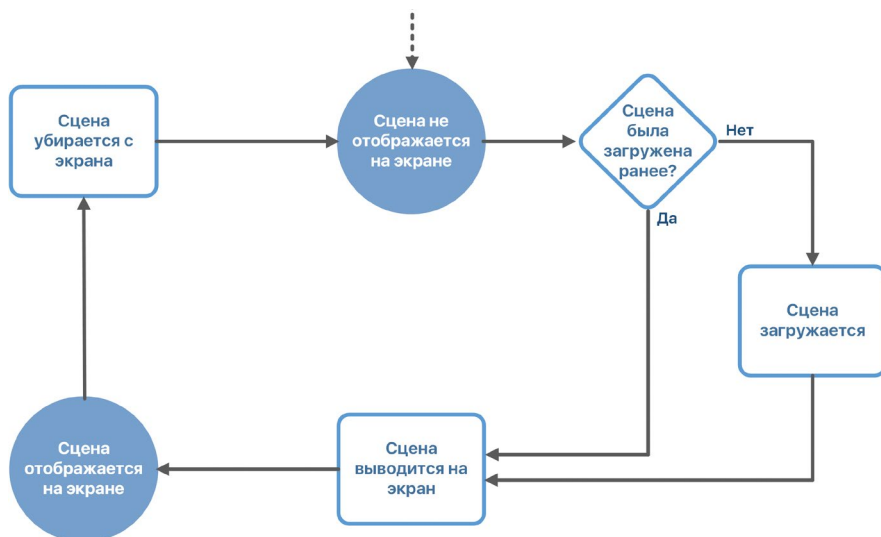


Рис. 3.2. Жизненный цикл View Controller

Сцена может иметь всего два состояния (отмечены кругами на рисунке 3.2):

- «Сцена не отображается на экране»;
- «Сцена отображается на экране».

Не закрашенные прямоугольники определяют процессы, которые происходят при изменении состояния сцены, а ромб — условие, которое может запустить цикл по разным путям.

Разберем каждый элемент схемы:

1. Перед тем, как сцена будет показана на экране устройства, она имеет состояние «Не отображается на экране».
2. Каждый раз перед ее отображением происходит проверка: была ли данная сцена ранее загружена в память устройства или нет.
3. Если сцена отображается впервые (т.е. пользователь первый раз переходит к данному рабочему экрану), она загружается в память.
4. Далее происходит процесс отображения сцены на экране устройства.
5. Сцена имеет состояние «Отображается на экране».
6. Когда работа со сценой окончена (пользователь переходит к другой сцене или сворачивает приложение), View Controller запускает процесс скрытия сцены с экрана устройства, после чего сцена вновь переходит в состояние «Не отображается на экране». При этом она остается в памяти устройства.

Как вы можете заметить, ничего сложного в этом процессе нет. Но наверняка в вашей голове возникли следующие вопросы: «А зачем мне вообще все это знать?», «Где эти знания будут использоваться?». Обо всем этом и не только мы поговорим на примере приложения «**Right on target**».

3.3 «Right on target», версия 1.1

Следует напомнить, что в ходе работы над приложением «**Right on target**» у нас осталась одна нерешенная проблема: для того, чтобы начать игру, необходимо совершить «бесполезное» для пользователя нажатие кнопки на сцене, после которого будет выбрано случайное число, и игра сможет начаться. Предлагаю решить данную проблему, опираясь на жизненный цикл View Controller.

Метод `viewDidLoad`

Жизненный цикл описывает не только состояния, которые принимает тот или иной объект, но и особые моменты, когда пользователь может оказать влияние на объект. Для View Controller такими моментами являются специальные методы, которые автоматически вызываются в процессе жизни сцены. Все эти методы уже определены в классе **UIViewController** и могут быть переопределены в его подклассах (в нашем случае в классе **ViewController**, который обеспечивает работу единственной сцены в игре «**Right on target**»).

- откройте файл **ViewController.swift**.

Обратите внимание на то, что в составе класса **ViewController** уже объявлен метод **viewDidLoad()**, который как раз и относится к жизненному циклу View Controller (листинг 3.1).

ЛИСТИНГ 3.1

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    // Do any additional setup after loading the view.  
}
```

- Добавьте в тело метода **viewDidLoad()** вызов функции **print(_:)** с аргументом «**viewDidLoad**» (листинг 3.2).

ЛИСТИНГ 3.2

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    print("viewDidLoad")  
}
```

- Запустите приложение на симуляторе.

В процессе запуска приложения на отладочной консоли должна появиться надпись «**viewDidLoad**» (рис. 3.3).

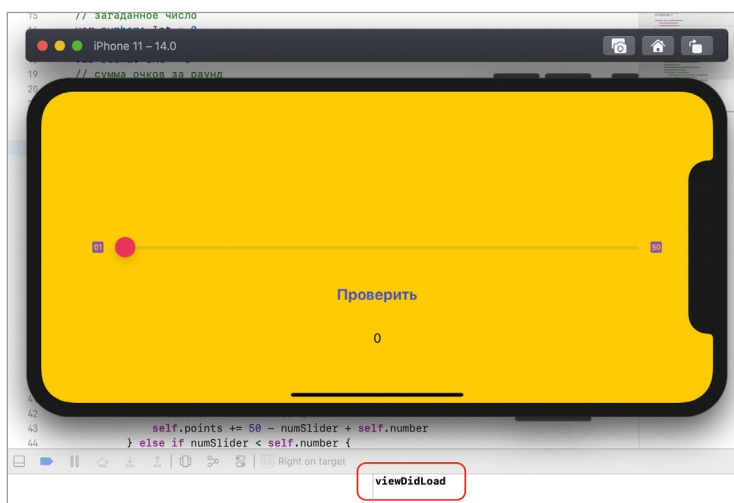


Рис. 3.3. Вывод на консоль

Но в какой именно момент происходит вызов метода **viewDidLoad()** и вывод строки на консоль? Если посмотреть внимательно, то можно заметить, что надпись выводится за несколько мгновений до того, как будет отображен интерфейс приложения. Это происходит по той причине, что метод **viewDidLoad()** вызывается до того, как сцена переходит в состояние «Отображается на экране», а точнее в тот момент, когда все View, из которых состоит сцена, уже загружены

и готовы к выводу на экран (рис. 3.4). С помощью **viewDidLoad()** у вас появляется возможность внести в графические элементы любые финальные коррективы перед их отображением (например, изменить текст или сменить цвет).

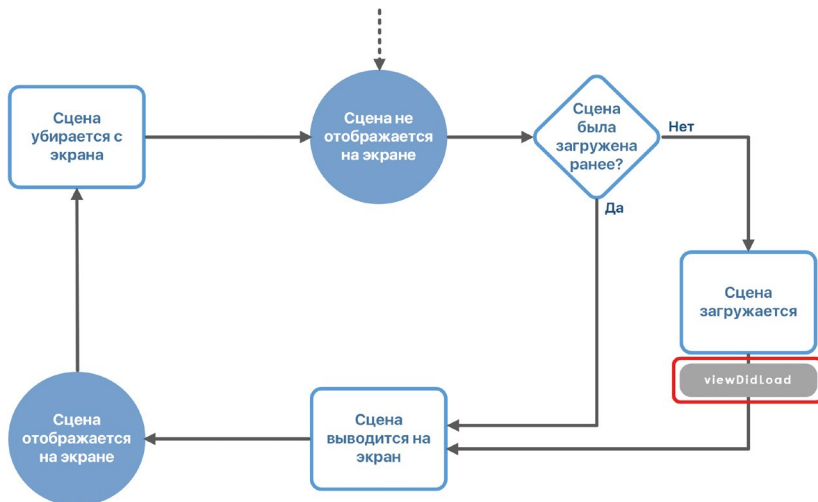


Рис. 3.4. Метод **viewDidLoad()** в жизненном цикле View Controller

Примечание Хочу отметить, что для того, чтобы определить предназначение того или иного метода, порой достаточно просто посмотреть на его название. Так, метод **viewDidLoad()** условно можно перевести как «корневое представление было загружено». Т.е. метод вызывается после того, как сцена загрузилась в память устройства.

Запомните, метод **viewDidLoad()** вызывается только один раз за всю жизнь сцены. В следующий раз, когда сцена будет повторно отображаться, блок проверки (ромб на рис. 3.4) отправит жизненный цикл по другой ветке. Если в проекте содержится несколько сцен, данный метод будет вызван один раз для каждой из них.

Доработка приложения

С помощью **viewDidLoad()** мы можем исправить озвученную ранее проблему игры «**Right on target**», т.е. совершить все подготовительные действия по выбору и отображению случайного числа заранее и не требовать нажатия кнопки. Это означает, что пользователь сможет начать игру сразу после загрузки приложения!

- Перенесите код подготовительных действий из **checkNumber()** в метод **viewDidLoad()** (листинг 3.3). При этом не удаляйте вызов метода **print**, так как он понадобится нам чуть позже.

ЛИСТИНГ 3.3

```
override func viewDidLoad() {
    super.viewDidLoad()
    print("viewDidLoad")
    // генерируем случайное число
    self.number = Int.random(in: 1...50)
    // устанавливаем загаданное число в метку
    self.label.text = String(self.number)
}
```

- Внесите правки в метод **checkNumber()**, убрав функциональность, перенесенную во **viewDidLoad()** (листинг 3.4).

ЛИСТИНГ 3.4

```
@IBAction func checkNumber() {
    // получаем значение на слайдере
    let numSlider = Int(self.slider.value)
    // сравниваем значение с загаданным
    // и подсчитываем очки
    if numSlider > self.number {
        self.points += 50 - numSlider + self.number
    } else if numSlider < self.number {
        self.points += 50 - self.number + numSlider
    } else {
        self.points += 50
    }
    if self.round == 5 {
        // выводим информационное окно
        // с результатами игры
        let alert = UIAlertController(
            title: "Игра окончена",
            message: "Вы заработали \(self.points) очков",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "Начать заново", style:
.default, handler: nil))
        self.present(alert, animated: true, completion: nil)
        self.round = 1
        self.points = 0
    } else {
        self.round += 1
    }
    // генерируем случайное число
    self.number = Int.random(in: 1...50)
}
```

```
// передаем значение случайного числа в label
self.label.text = String(self.number)
}
```

- Измените значение свойства **round** класса **ViewController** на 1 (листинг 3.5).

ЛИСТИНГ 3.5

```
// раунд
var round: Int = 1
```

- Запустите приложение и проверьте его работоспособность.

Теперь игра начинается сразу после загрузки приложения и отображения интерфейса на экране устройства. И сделано это было благодаря возможностям жизненного цикла View Controller.

Смена версии приложения в Xcode

Довольно часто разработчик ограничивает функциональность первой версии приложения, ограничиваясь реализацией так называемого MVP (Minimum Viable Product, или минимальный жизнеспособный продукт). Т.е. сперва создается версия продукта с минимальным набором функций, а уже после получения фидбека (обратной связи) от пользователей исправляются выявленные ошибки и добавляются новые возможности. Это экономит большое количество времени разработчиков, а также позволяет им «прощупать» почву, проанализировав реакцию потенциальной аудитории.

Для того, чтобы отличать версии продукта друг от друга и сообщать пользователю о нововведениях, каждая из них должна иметь уникальный номер (например, версия 1.0, версия 1.3, версия 2.0). Вы неоднократно могли видеть такой подход в App Store (рис. 3.5).

Вы как разработчик можете самостоятельно определять версию приложения, последовательно изменяя номер после очередной доработки. Последняя правка **«Right on target»** была довольно важной, поскольку мы позволили пользователю начинать игру сразу после загрузки, исключив «мусорное» нажатие на кнопку. По этой причине целесообразно выпустить новую версию игры с новым номером.

- Изменим версию проекта с 1.0 на 1.1.
- В **Project Navigator** щелкните по файлу проекта.
- В левой части **Project Editor** в разделе **Targets** выберите пункт **Right on target**.
- Выберите вкладку **General**.
- Измените значение поля **Version** на 1.1.

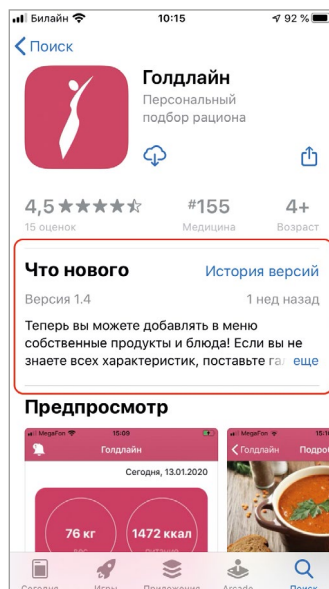


Рис. 3.5. Сообщение о новых функциях в приложении

Теперь, если бы вы загружали приложение в AppStore Connect (сервис, который используется для размещения приложений в App Store), то увидели бы игру с новым номером версии.

3.4 Введение в отображение графических элементов

Прежде чем продолжить рассмотрение жизненного цикла View Controller, необходимо обсудить несколько вопросов, связанных с отображением графических элементов, из которых состоит сцена.

Иерархия графических элементов

Первое понятие, с которым необходимо познакомиться, это «**иерархия графических элементов**» (или иерархия View) в приложении.


Все графические элементы, которые в текущий момент времени выводятся на экран, находятся в иерархии, т.е. в такой организации, при которой одни элементы подчиняются другим (или, иными словами, входят в состав других). Такой подход уже должен быть знаком вам по приложению с шариками из первой книги: цветная подложка (представлена отдельным view) включала в свой

состав шарики (множество view), тем самым создавалась иерархия व्यूшек. Точно таким же способом организована иерархия элементов в любых приложениях на iOS, в том числе и в «**Right on target**». В состав корневого View (оранжевого цвета) входят три независимые view: слайдер, кнопка и текстовая метка. При этом «под капотом» слайдер реализован также с помощью нескольких отдельных view. Таким образом, получается многоуровневая иерархия.

Состав и порядок элементов в иерархии изменяется в процессе функционирования приложения. К примеру:

- если программе требуется отобразить на экране новый элемент, он добавляется в соответствующее место в иерархии;
- если происходит переход к новому экрану, View Controller этого экрана загружает все view и добавляет их в иерархию, тем самым отображая сцену на экране. При этом в зависимости от реализованной логики предыдущий экран может оставаться в иерархии, но перекрываться элементами новой сцены или удаляться из иерархии.

Посмотрим на то, как выглядит иерархия запущенного приложения.

- ▶ Запустите проект на симуляторе.
- ▶ После отображения интерфейса в нижней части нажмите на кнопку **Debug View Hierarchy**  (рис. 3.6).

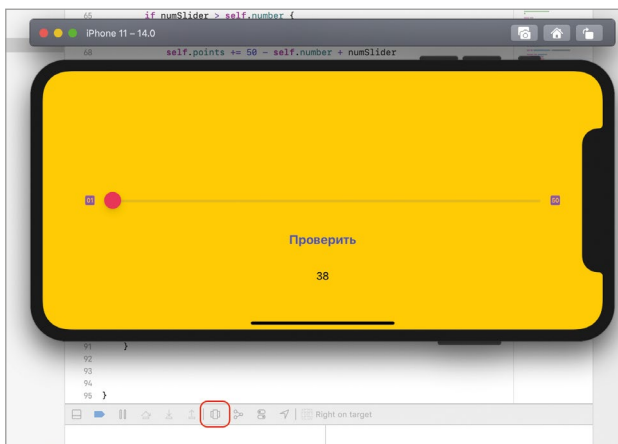


Рис. 3.6. Кнопка Debug View Hierarchy

После нажатия на кнопку приложение будет приостановлено (поставлено на паузу), а в Xcode отобразится иерархия всех графических элементов приложения, размещенных на экране устройства в момент нажатия кнопки (рис. 3.7). В левой панели среды элементы будут показаны в виде вложенного списка, в **Project Editor** – в графическом виде.

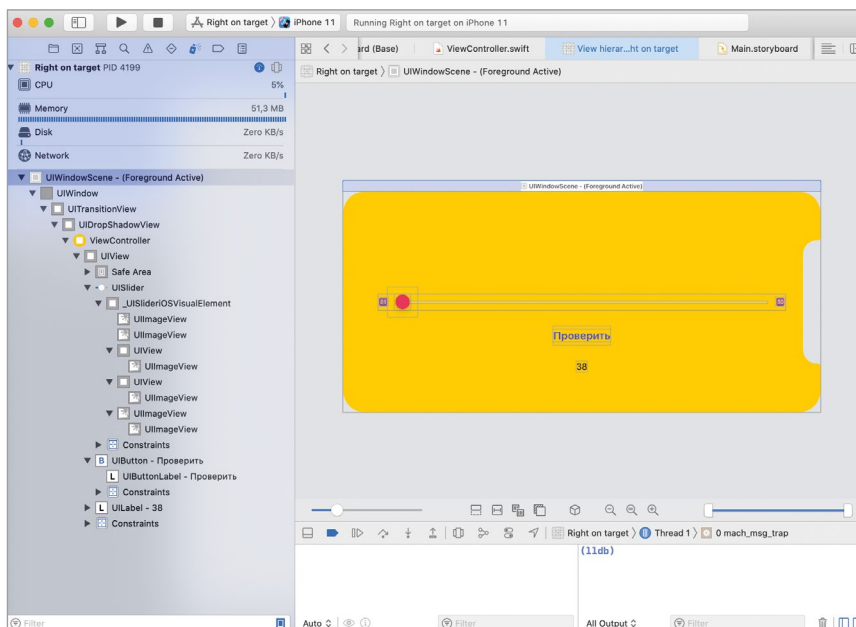


Рис. 3.7. Иерархия view проекта Right on target

Примечание В иерархии помимо знакомых вам элементов содержатся и несколько системных, вроде **UITransitionView**, **UIDropShadowView**, **UIWindow** и **UIWindowScene**. Некоторые из них мы рассмотрим в следующей главе.

► В нижней части **Project Editor** нажмите на кнопку **Orient to 3D** .

Теперь иерархия представлений показана в трехмерном режиме (рис. 3.8). Используя мышку, бегунки и кнопки в нижней части **Project Editor**, вы имеете возможность перемещать и поворачивать представления.

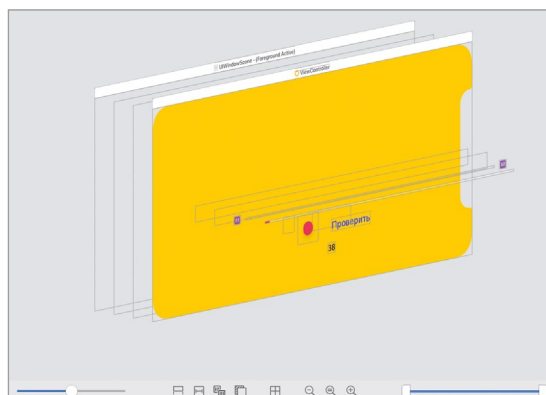



Рис. 3.8. Иерархия view в трехмерном режиме

Для того, чтобы элемент был отображен на экране, он обязательно должен находиться в данной иерархии. Элементы, которые находятся ниже в иерархии, могут перекрывать находящиеся выше. Это как стопка книг, если смотреть на нее сверху. Вы кладете книги друг на друга, и каждая новая книга закрывает полностью или частично те, что лежат ниже.

В момент вызова метода **viewDidLoad** все графические представления уже загружены в память устройства, но пока еще не добавлены в иерархию и поэтому не отображаются на экране. Понятие иерархии понадобится нам при дальнейшем рассмотрении жизненного цикла.

- Для снятия выполнения проекта с паузы нажмите кнопку **Continue Program Execution** .

Позиционирование графических элементов

При размещении графических элементов в иерархии они должны содержать данные для их позиционирования на сцене: координаты, размеры и ограничения (constraints). Очень важным является тот факт, что координаты элементов указываются относительно родительского (старшего) к нему элемента, а не относительно корневого.

Рассмотрим пример небольшой иерархии (рис. 3.9).



Рис. 3.9. Позиционирование элемента **childView**

В данном примере графический элемент **rootView** является корневым, т.е. самым старшим. В его состав входит элемент **parentView**, который в свою очередь включает в себя элемент **childView**. Вам важно понять и запомнить, что координаты **childView** указываются относительно системы координат внутри **parentView**, т.е. внутри родительского элемента, но не внутри **rootView**. А вот **parentView** позиционируется внутри системы координат **rootView**.

Если бы координаты **(x, y)** у **childView** были бы равны **(0,0)**, левый верхний угол элемента совпадал бы с левым верхним углом **parentView**. А если бы ширина и высота **childView** равнялись ширине и высоте **parentView**, то в этом случае он бы полностью перекрыл собой **parentView**. Иерархия элементов оставалась бы неизменной и включала бы в себя все три описанные выше элемента, но **parentView** при этом был бы скрыт:

rootView

- **parentView**

-- **childView**

Графический элемент, в состав которого входит текущий элемент, называется **superview**. Например, **parentView** – это **superview** для **childView**.

Графический элемент, который входит в состав текущего, называется **subview**. Например, **parentView** – это **subview** для **rootView**.

Способ позиционирования элемента в системе координат своего **superview** называется **frame**. Это очень важное понятие, о котором вас наверняка будут спрашивать на собеседовании. Помимо **frame** также существует **bounds** – позиционирование графического элемента в своей собственной системе координат. В последней части книги мы более детально разберемся с тем, что же такое **frame** и **bounds**.

Примечание Я привожу данные понятия на английском не просто так. Далее они будут встречаться вам в коде именно в указанных формах.

3.5 Схема жизненного цикла View Controller

Хочется отметить, что сделанная нами ранее привязка жизненного цикла View Controller к отображению сцены на экране не совсем корректна. Дело в том, что сцена может находиться в иерархии графических элементов, но при этом не отображаться на экране по той причине, что она будет перекрыта другой сценой. Поэтому нам необходимо внести соответствующие изменения в схему жизненного цикла, изменив текст в блоках (рис. 3.10).

Теперь состояния сцены основываются не на отображении сцены на экране, а на ее вхождении в иерархию представлений приложения.

Жизненный цикл View Controller помимо метода **viewDidLoad()** включает в себя и другие, каждый из которых, точно как **viewDidLoad()**, автоматически вызывается в определенные моменты жизни сцены. На рисунке 3.11 показана схема жизненного цикла с доступными разработчику методами и их привязкой к моментам вызова.

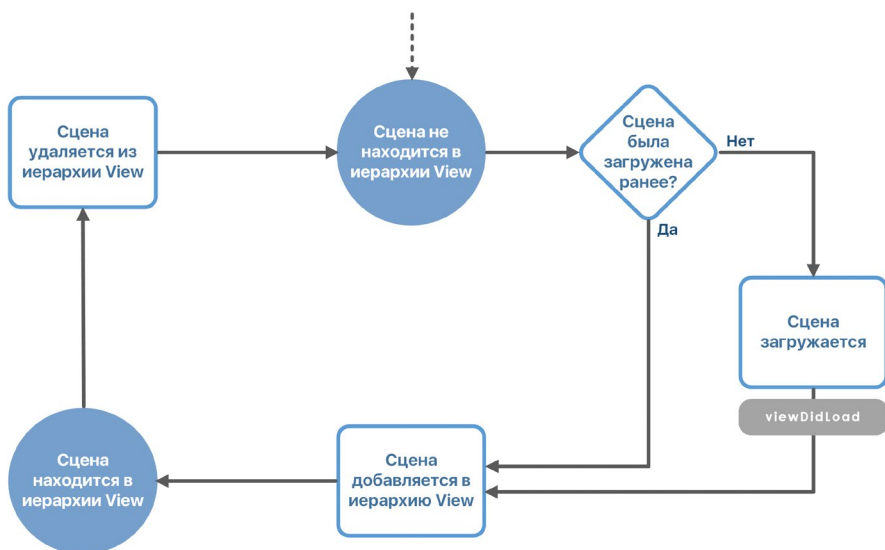


Рис. 3.10. Жизненный цикл View Controller

Примечание В этой главе мы сперва рассмотрим методы, потом внесем правки в код и уже только после этого посмотрим, как вызываются методы в процессе работы приложения.

Я буду давать краткую характеристику каждому методу, однако полное понимание их использования придет к вам несколько позже, когда мы начнем применять их на практике.

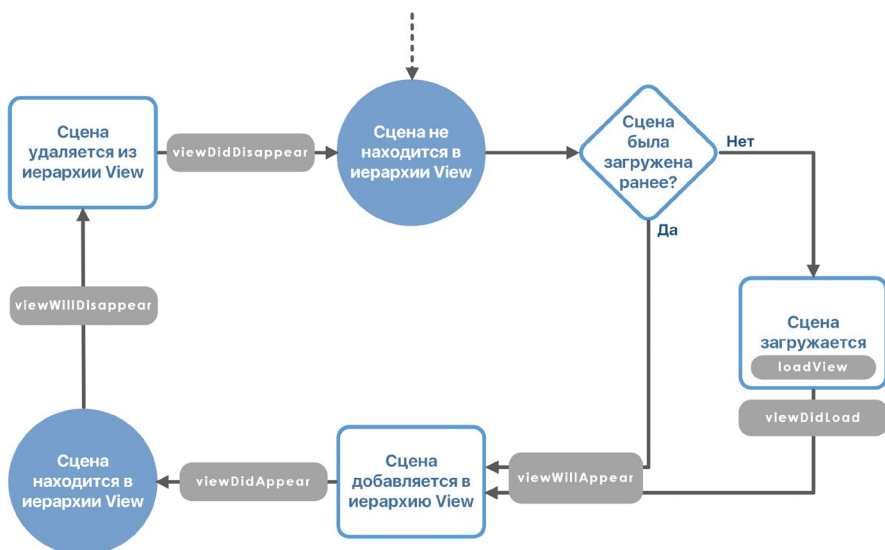


Рис. 3.11. Жизненный цикл View Controller

Метод `loadView`

Метод `loadView` выполняется первым в жизненном цикле. Как и `viewDidLoad`, он вызывается лишь один раз за все время жизни сцены. Если сцена создана с помощью **Interface Builder** (другим вариантом является создание элементов сцены с помощью программного кода), в данном методе производится загрузка всех размещенных на сцене графических элементов.

Примечание В первой книге мы смотрели с вами на структуру storyboard-файла. В данном случае Xcode загружает сцену из этого файла, анализирует ее и самостоятельно создает все необходимые объекты.

Если вы переопределите любой метод жизненного цикла в дочернем к `UIViewController` классе (в нашем случае это `ViewController`), то обязательно должны будете вызвать родительскую реализацию метода с помощью ключевого слова `super`. Дело в том, что эта родительская реализация содержит множество скрытых от разработчика действий, необходимых для выполнения жизненного цикла View Controller.

- ▶ Переопределите метод `loadView` в классе `ViewController` и реализуйте его, как показано в листинге 3.6.

Примечание Обратите внимание, что для быстрой реализации метода вы можете использовать функцию автодополнения Xcode. Просто начните вводить имя метода `loadView` и в появившемся списке выберите необходимый пункт (рис. 3.12).

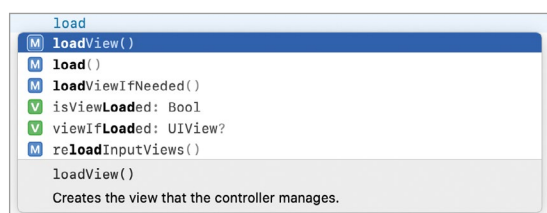


Рис. 3.12. Окно автодополнения в Xcode

ЛИСТИНГ 3.6

```
override func loadView() {  
    super.loadView()  
    print("loadView")  
}
```

Подумайте над тем, что будет, если пропустить вызов `super.loadView()`?

В этом случае приложение экстренно завершит работу. Дело в том, что в классе `ViewController` объявлены аутлеты, которые ссылаются на графические элементы. Но так как элементы не были загружены, попытка обращения к аутлетам как раз и приведет к критической ошибке.

Метод **loadView** прекрасно подходит для того, чтобы создать новые графические элементы с помощью программного кода. В листинге 3.7 показан пример добавления текстовой метки и вставки ее на сцену.

ЛИСТИНГ 3.7

```
override func loadView() {
    super.loadView()
    print("loadView")

    // Создаем метку для вывода номера версии
    let versionLabel = UILabel(frame: CGRect(x: 20, y: 10, width: 200,
height: 20))
    // изменяем текст метки
    versionLabel.text = "Версия 1.1"
    // добавляем метку в родительский view
    self.view.addSubview(versionLabel)
}
```

Класс **CGRect** описывает сущность «Прямоугольник», а в качестве аргументов при его создании передаются координаты по осям *x* и *y*, ширина и высота. Координаты указывают, где именно должен находиться левый верхний угол данного элемента относительно левого верхнего угла родительского элемента (вспомните про **frame** – позиционирование в **superview**).

В инициализатор класса **UILabel** передается экземпляр типа **CGRect**, который как раз и описывает, в какую прямоугольную область необходимо вписать текстовую метку (где ее разместить и какого размера она должна быть).

Текстовая метка добавляется в иерархию с помощью метода **addSubview** и таким образом размещается на сцене. Т.е., с помощью данного метода мы добавляем **subview** (дочерний графический элемент).

Теперь, если запустить приложение, в левом верхнем углу будет отображена текстовая метка, созданная и добавленная в иерархию **view** с помощью программного кода в методе **loadView** (рис. 3.13).

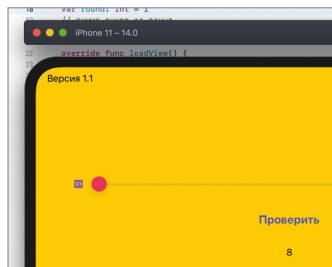


Рис. 3.13. Вывод текстовой метки с помощью кода

Но вот что интересно! Если написать код создания метки до вызова **super.loadView()**, приложение экстренно завершит работу уже в процессе загрузки (рис. 3.14). Это произойдет по причине того, что корневой **view** сцены еще не загружен, и обращение к свойству **self.view** приводит к ошибке, так как до момента вызова **super.loadView()** свойство **self.view** соответствует **nil**, т.е. в нем нет значения.

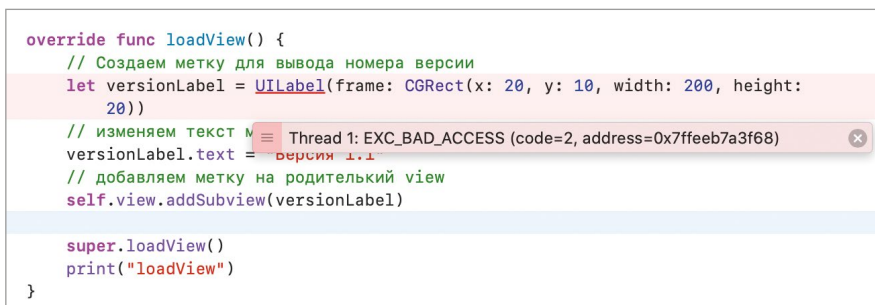


Рис. 3.14. Ошибка при попытке отобразить метку

Понимание всех процессов жизненного цикла View Controller является очень важным знанием для Swift-разработчика, так как это позволит избежать ошибок и использовать предоставляемые возможности с максимальной эффективностью.

- Если вы добавили в метод **loadView** код для создания текстовой метки, удалите его, так как он больше не понадобится. Но вызов функции **print** при этом не удаляйте.

Метод **viewDidLoad**

Метод **viewDidLoad** уже был рассмотрен нами ранее. Напомню, что он вызывается сразу после загрузки всех отображений (всех графических элементов) и прекрасно подходит для того, чтобы внести финальные правки перед выводом сцены на экран (или другими словами, перед добавлением графических элементов в иерархию вьюшек).

Данный метод вызывается один раз за все время жизни View Controller и сцены, которой он управляет.

Метод **viewWillAppear**

Метод **viewWillAppear** вызывается перед тем, как графические элементы сцены будут добавлены в иерархию графических элементов. Но в отличие от **viewDidLoad** он вызывается не один раз, а каждый раз, когда сцена добавляется в иерархию.

- Реализуйте метод **viewWillAppear** так, как показано в листинге 3.8.

ЛИСТИНГ 3.8

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    print("viewWillAppear")
}
```

Метод **viewDidAppear**

Метод **viewDidAppear** вызывается после того, как графические элементы сцены добавлены в иерархию view. В данном методе вы можете произвести действия, которые должны быть выполнены уже после отображения элементов на экране (например, запустить анимацию на сцене или синхронизировать данные с сервером).

- Реализуйте метод **viewDidAppear** так, как показано в листинге 3.9.

ЛИСТИНГ 3.9

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    print("viewDidAppear")
}
```

Методы **viewWillDisappear** и **viewDidDisappear**

Методы **viewWillDisappear** и **viewDidDisappear** похожи на **viewWillAppear** и **viewDidAppear** с той лишь разницей, что они вызываются до и после удаления элементов сцены из иерархии view.

- Реализуйте методы **viewWillDisappear** и **viewDidDisappear** так, как показано в листинге 3.10.

ЛИСТИНГ 3.10

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    print("viewWillDisappear")
}

override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
    print("viewDidDisappear")
}
```

Создание второй сцены в приложении

Для того, чтобы продемонстрировать работу методов жизненного цикла, создадим в приложении еще одну сцену и посмотрим, какие сообщения будут выводиться на консоль при переходе между сценами.

В составе проекта создадим новый класс **SecondViewController**, потомок **UIViewController**.

- ▶ В главном меню Xcode нажмите **File > New > File** (или сочетание клавиш **⌘Command + N**).
- ▶ В появившемся окне выберите **Cocoa Touch Class** и нажмите **Next**.
- ▶ В следующем окне заполните поля, как показано ниже:
 - **Subclass of - UIViewController**
 - **Class - SecondViewController**
- ▶ Нажмите **Next** и сохраните файл.

Теперь в составе проекта в **Project Navigator** появился новый файл **SecondViewController**, в котором будет реализован View Controller новой сцены.

- ▶ В классе **SecondViewController** реализуйте все рассмотренные ранее методы жизненного цикла так, как показано в листинге 3.11.

ЛИСТИНГ 3.11

```
class SecondViewController: UIViewController {  
    override func loadView() {  
        super.loadView()  
        print("loadView SecondViewController")  
    }  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        print("viewDidLoad SecondViewController")  
    }  
  
    override func viewWillAppear(_ animated: Bool) {  
        super.viewWillAppear(animated)  
        print("viewWillAppear SecondViewController")  
    }  
  
    override func viewDidAppear(_ animated: Bool) {  
        super.viewDidAppear(animated)  
        print("viewDidAppear SecondViewController")  
    }  
}
```



```
override func viewWillDisappear(_ animated: Bool) {  
    super.viewWillDisappear(animated)  
    print("viewWillDisappear SecondViewController")  
}  
  
override func viewDidDisappear(_ animated: Bool) {  
    super.viewDidDisappear(animated)  
    print("viewDidDisappear SecondViewController")  
}  
}
```

Теперь каждый из двух view контроллеров, которые есть в нашем приложении, имеет собственные реализации методов жизненного цикла. В дальнейшем, по выводу на консоль мы сможем определить, в каком порядке вызываются данные методы.

С помощью **Interface Builder** добавим на storyboard новую сцену.

- ▶ Откройте файл **Main.storyboard**.
- ▶ Разместите на storyboard новый элемент View Controller (для этого найдите его в библиотеке объектов и перетащите).
- ▶ Выделите новую сцену на storyboard.
- ▶ На панели **Inspectors** перейдите во вкладку **Identity Inspector**.
- ▶ В поле **Class** укажите **SecondViewController**.
- ▶ Разместите на новой сцене кнопку (Button).
- ▶ Создайте для кнопки ограничения (constraints) так, чтобы она была выровнена по центру сцены вертикальной и горизонтальной осей.
- ▶ Текст кнопки измените на «**Назад**».

В результате проделанных действий ваш storyboard должен выглядеть примерно так, как показано на рисунке 3.15.

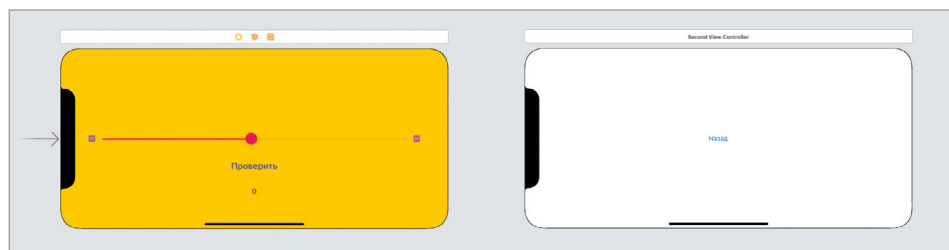


Рис. 3.15. Новая сцена на storyboard

Добавим на «оранжевую» сцену кнопку, по нажатию которой и будет осуществляться переход на новую сцену:

- ▶ Добавьте в правый нижний угол сцены кнопку.
- ▶ Измените текст кнопки на «**О программе**».
- ▶ Укажите для кнопки следующие ограничения:
 - Отступ от правого элемента — 20 точек.
 - Отступ от нижнего элемента — 0 точек.

Примечание Для создания ограничений выделите элемент на сцене и нажмите кнопку **Add New Constraints**, расположенную в нижней части **Interface Builder** (рис. 3.16).

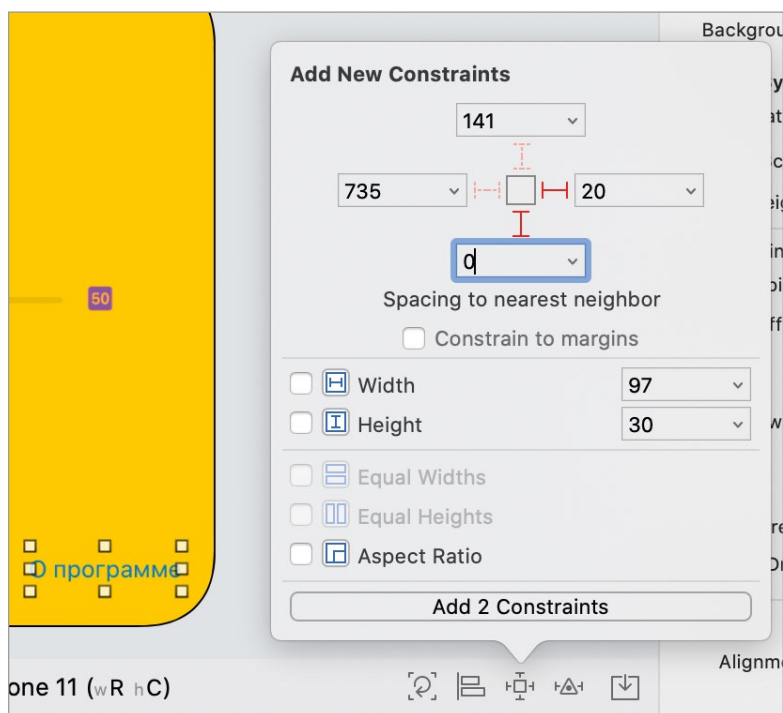


Рис. 3.16. Создание ограничений для графического элемента

На этом мы закончили все подготовительные работы.

Перейдем к созданию переходов между сценами. Всего нами будут рассмотрены три варианта организации смены сцен. При этом мы будем обращать внимание на то, как срабатывают методы жизненного цикла. Это позволит не просто изучить, как именно работают данные методы, но и понять, к чему приводит каждый из описанных способов.

Вариант смены сцены № 1. Взаимные segue

Самый простой и уже знакомый вам по первой книге вариант – создать переход (segue или сиквей) для каждой из кнопок.

- ▶ Выделите кнопку «**О программе**» на первой сцене.
- ▶ Зажмите клавишу **Control** и перетяните кнопку на вторую сцену. При этом в процессе перетаскивания сам элемент останется на месте, но от него будет тянуться синяя линия.
- ▶ Когда вторая сцена подсветится синим, отпустите мышь, а в появившемся меню выберите пункт **Action Segue > Show**.

Вы создали **segue**, который сработает при нажатии кнопки на первой сцене и обеспечит отображение второй сцены на экране устройства. Данный **segue** на **storyboard** обозначен в виде стрелки с пиктограммой (рис. 3.17).

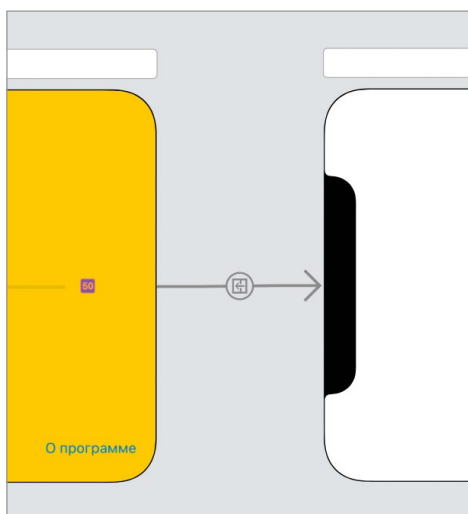


Рис. 3.17. Обозначение segue на storyboard

Теперь создадим обратный переход от второй сцены к первой:

- ▶ Выделите кнопку «**Назад**» на второй сцене.
- ▶ Зажмите клавишу **Control** и перетяните кнопку на «оранжевую» сцену. В процессе перетаскивания сам элемент останется на месте, но от него будет тянуться синяя линия.
- ▶ Когда первая сцена подсветится синим, отпустите мышь, а в появившемся меню выберите пункт **Action Segue > Show**.

Теперь на storyboard отображаются два перехода (рис. 3.18).

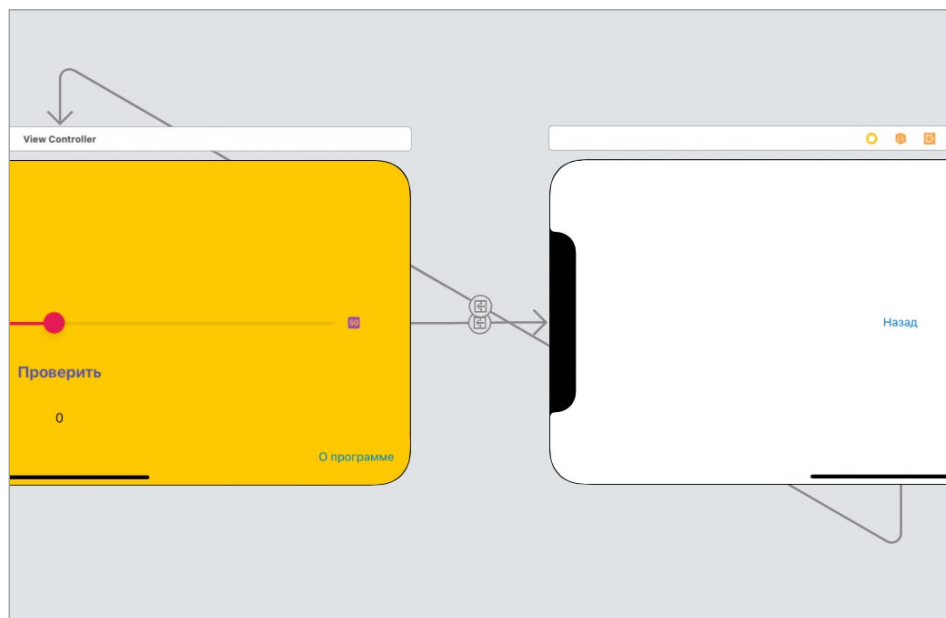


Рис. 3.18. Обозначение segue на storyboard

► Запустите приложение.

После появления оранжевой сцены на консоли отобразятся сообщения, соответствующие последовательным вызовам методов **loadView**, **viewDidLoad**, **viewWillAppear** и **viewDidAppear** класса **ViewController** (рис. 3.19). Каждый из методов был вызван в соответствии с жизненным циклом View Controller при загрузке и отображении сцены.



Рис. 3.19. Вывод на консоль

Теперь осуществим переход ко второй сцене:

► Нажмите на кнопку «**О программе**», после чего на экране отобразится новая сцена.

Если вы посмотрите на консоль, то сможете заметить новые сообщения, аналогичные тем, что показаны на рисунке 3.19, только теперь для класса **SecondViewController** (рис. 3.20).

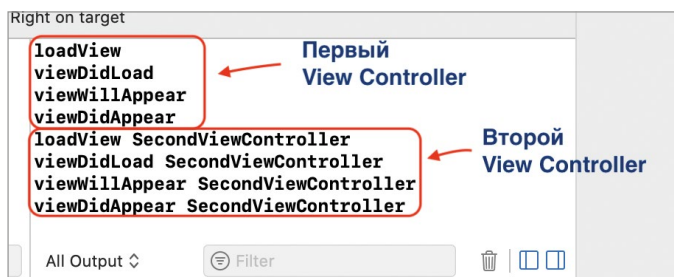


Рис. 3.20. Вывод на консоль

Обратите внимание, что ни **viewWillDisappear**, ни **viewDidDisappear** для первого экрана вызваны не были, так как на консоли отсутствуют соответствующие сообщения. Это говорит о том, что графические элементы первой сцены все еще находятся в иерархии. И действительно, если посмотреть на иерархию view (рис. 3.21), вы увидите, что первая сцена все еще там, просто она не видна, так как перекрыта второй сценой.

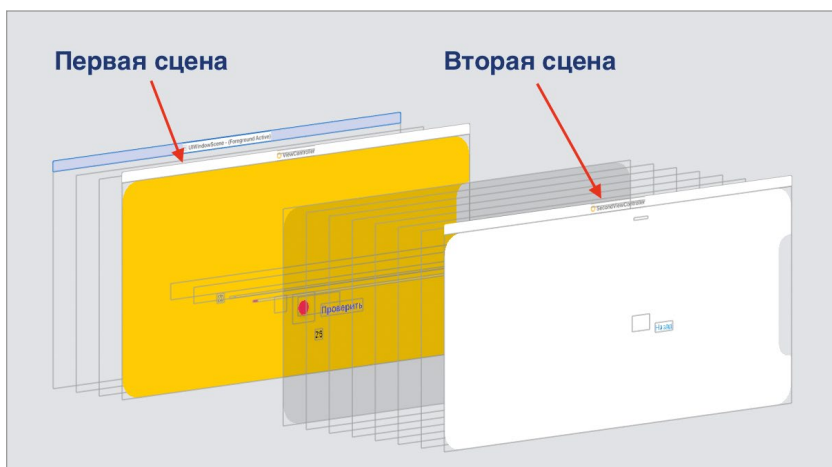


Рис. 3.21. Иерархия графических элементов

В этом состоит особенность использования segue: текущая сцена не удаляется из иерархии, а новая просто накладывается сверху. Подумайте, что будет, если сейчас нажать на кнопку «Назад» на второй сцене, ведь там точно также используется segue?

- ▶ Снимите приложение с паузы.
- ▶ Нажмите на кнопку «Назад» на второй сцене.

После вызова сивгея будет снова отображена «оранжевая» сцена, и мы, кажется, произвели обратный переход, но вот на что стоит обратить внимание:

1. несмотря на то, что методы **loadView** и **viewDidLoad** должны вызываться всего один раз за всю жизнь **View Controller**, на консоли вновь были выведены соответствующие сообщения (рис. 3.22);
2. методы **viewWillDisappear** и **viewDidDisappear** класса **SecondViewController** не были вызваны (на консоли отсутствуют сообщения об этом).

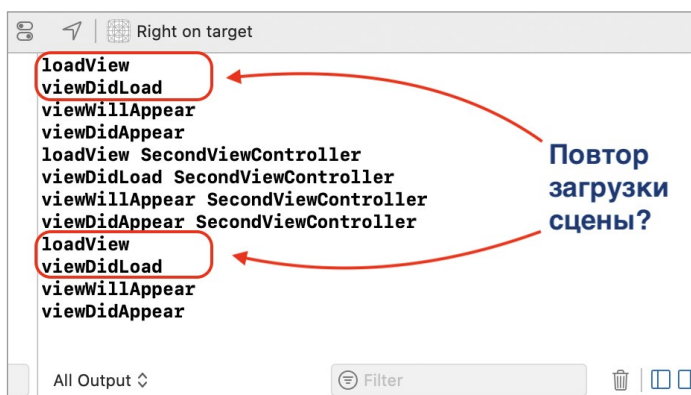


Рис. 3.22. Вывод на консоль

И вновь причина заключается в принципе работы segue. Если посмотреть на иерархию view, то возможно вы удивитесь (рис. 3.23), но в ней находится два комплекта «оранжевых» сцен. Таким образом, «оранжевая» сцена, которая сейчас отображается на экране, это не та сцена, которая была загружена ранее и сохранена в памяти. При нажатии кнопки «Назад» приложение автоматически создало новый экземпляр класса **ViewController**, и все методы жизненного цикла были вызваны вновь.

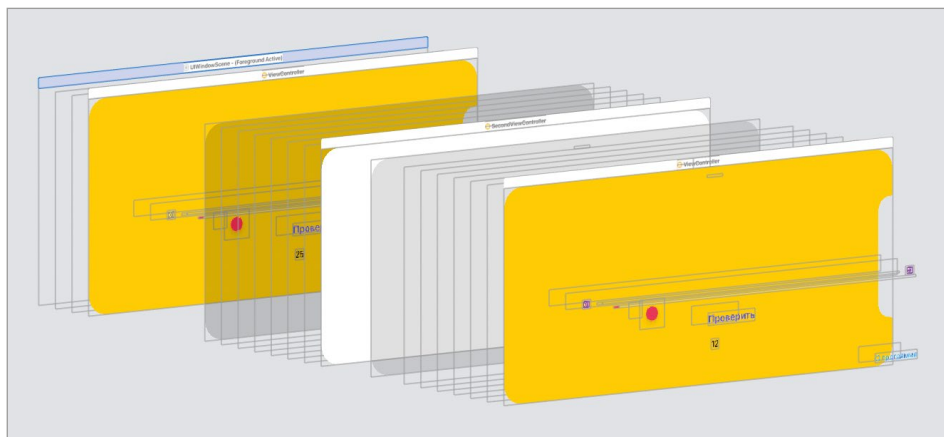


Рис. 3.23. Иерархия графических элементов

И сколько бы вы ни совершили переходов от одной сцены к другой, каждый раз будет создаваться новый экземпляр класса, вновь будет произведена загрузка сцены, а ее графические элементы будут добавлены в иерархию. И на каждый новый экземпляр будет резервироваться место в памяти. На рисунке 3.24 показан пример иерархии после осуществления четырех переходов.



Рис. 3.24. Иерархия графических элементов после четырех переходов

Бездумное использование segue (а это не редкость в самом начале карьеры) может привести к растрате оперативной памяти и потенциальной ее утечке.

Вариант смены сцены № 2. Методы `present` и `dismiss`

Класс `UIViewController` включает в себя методы `present` и `dismiss`, которые позволяют отображать и скрывать сцены.

Метод `present`

Данный метод уже должен быть знаком вам по материалу первой книги. Он используется и в текущем проекте. С его помощью отображается всплывающее сообщение с результатами игры.

СИНТАКСИС

Метод `UIViewController.present(_:animated:completion:)`

Добавляет `View Controller` и его сцену в иерархию представлений и отображает на экране устройства.

Аргументы

- `_:` `UIViewController` — выю контроллер, который будет отображен на экране устройства.
- `animated:` `Bool` — флаг, указывающий на необходимость использования анимации при отображении элемента.
- `completion:` `((() -> Void)? = nil` — замыкание, исполняемое после завершения вывода на экран.

Пример

```
self.present(alertController, animated: true, completion: nil)
```

В качестве первого аргумента необходимо передать значение типа **UIViewController**, которое определяет, какая именно сцена должна быть отображена на экране. Таким образом, для вывода второй сцены мы можем передать экземпляр **SecondViewController**, так как он является дочерним по отношению к **UIViewController** классом.

Примечание В нашем проекте в качестве первого аргумента методу `present` передается экземпляр типа **UIAlertController**. Это возможно благодаря тому, что **UIAlertController** — это дочерний класс **UIViewController**! Да, всплывающее модальное окно — это отдельная небольшая сцена, которая управляется своим собственным View Controller (экземпляром класса **UIAlertController**).

Исходя из этого можно сделать вывод о том, что сцена не обязательно должна занимать весь экран устройства целиком, а интерфейс приложения, который в определенный момент отображается на экране, может строиться сразу из нескольких View Controller!

Метод dismiss

Метод **dismiss** позволяет удалить из иерархии все графические элементы сцены. Другими словами, он выполняет противоположную методу **present** работу.

СИНТАКСИС

Метод `UIViewController.dismiss(animated:completion:)`

Скрывает View Controller и сцену, отображение которой он обеспечивает, удаляя их из иерархии представлений.

Аргументы

- `animated:` `Bool` — флаг, указывающий на необходимость использования анимации при удалении элемента.
- `completion:` `((() -> Void)? = nil` — замыкание, исполняемое после скрытия сцены.

Пример

```
self.dismiss(animated: true, completion: nil)
```

Метод **dismiss** применяется к View Controller, сцену которого необходимо убрать из иерархии. При этом сам экземпляр класса удаляется из памяти только в том случае, если на него не будет никаких ссылок (для этого используется ARC). Это важно!

Используя описанные методы, мы можем организовать следующий вариант перехода между сценами.

1. При нажатии на кнопку на первой сцене будет вызван метод **present**, которому в качестве аргумента будет передан экземпляр класса **SecondViewController**. После этого сцена загрузится, добавится в иерархию представлений и отобразится на экране.
2. При нажатии на кнопку на второй сцене будет вызван метод **dismiss**, после чего все графические элементы второй сцены будут убраны из иерархии, а значит сцена скроется с экрана. Пользователь вновь увидит «оранжевую» сцену, причем не новый ее экземпляр, а созданный ранее.

Реализуем переход от первой сцены ко второй.

- ▶ На **storyboard** удалите созданные segue (стрелки). Для этого выделите каждый segue и нажмите клавишу **Backspace**.
- ▶ В классе **ViewController** создайте пустой action-метод **showNextScreen** в соответствии с листингом 3.12.

ЛИСТИНГ 3.12

```
@IBAction func showNextScreen() {}
```

- ▶ Свяжите вызов метода **showNextScreen()** с нажатием кнопки «**О программе**» на первой сцене.

Как вы думаете, каким образом мы можем создать объект типа **SecondViewController** для того, чтобы передать его в метод **present**? Первое, что приходит на ум: просто создать экземпляр типа **SecondViewController** используя встроенный инициализатор (листинг 3.13).

ЛИСТИНГ 3.13

```
@IBAction func showNextScreen() {  
    let viewController = SecondViewController()  
    self.present(viewController, animated: true, completion: nil)  
}
```

- ▶ Запустите проект на симуляторе.
- ▶ Нажмите кнопку «**О программе**».

Мы получили интересный эффект: кнопка нажимается, экран слегка затемняется, но вторая сцена не отображается. Почему? Посмотрите внимательно на код класса **SecondViewController**, есть ли в нем хоть какая-то привязка к сцене, описанной на **storyboard**?

Не нашли? Посмотрите еще раз внимательнее. Снова не нашли?

И правильно, ее там нет. Данный вариант будет корректно функционировать лишь в том случае, когда все элементы сцены создаются с помощью кода в методе **loadView** класса **SecondViewController**. Но в нашем случае класс «пустой», так как все элементы сцены созданы в **Interface Builder**, а значит и действовать нам надо не от **SecondViewController**, а от **storyboard**.

► Дополните метод **showNextScreen** в соответствии с листингом 3.14.

ЛИСТИНГ 3.14

```
@IBAction func showNextScreen() {
    // загрузка Storyboard
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    // загрузка View Controller и его сцены со Storyboard
    let viewController = storyboard.instantiateViewController(identifier:
"SecondViewController")
    // отображение сцены на экране
    self.present(viewController, animated: true, completion: nil)
}
```

Класс **UINavigationController** позволяет представить storyboard-файл в виде программной сущности. В данном случае мы загружаем файл с именем **Main.storyboard**. После этого мы можем загрузить необходимую нам сцену, передав в метод **instantiateViewController** идентификатор требуемого **View Controller**, и отобразить ее на экране с помощью метода **present**.

Нам осталось лишь определить идентификатор:

- На **storyboard** выделите View Controller второй сцены.
- Откройте панель **Identity Inspector**.
- Измените значение поля **Storyboard ID** на «**SecondViewController**».
- Произведите запуск приложения и проверьте функциональность кнопки «**О программе**».

Перейдем к реализации скрытия второй сцены.

- В классе **SecondViewController** создайте action-метод в соответствии с листингом 3.15.

ЛИСТИНГ 3.15

```
@IBAction func hideCurrentScene() {
    self.dismiss(animated: true, completion: nil)
}
```

- Свяжите нажатие кнопки на второй сцене с вызовом метода **hideCurrentScene()**.

- ▶ Запустите приложение на симуляторе.

Ровно так же, как и в прошлый раз, сразу после загрузки приложения на консоли отобразятся сообщения, соответствующие загрузке и выводу на экран первой сцены.

- ▶ Нажмите на кнопку «О программе».

После перехода ко второй сцене на консоли появятся сообщения, соответствующие загрузке класса **SecondViewController** и отображению его сцены (рис. 3.25).

- ▶ Нажмите на кнопку «Назад».

В этот раз, в отличие от использования segue, на консоли появятся сообщения, указывающие на вызов методов **viewWillDisappear** и **viewDidDisappear** класса **SecondViewController**. Это значит, что элементы сцены убраны из иерархии view. При этом ни один из методов жизненного цикла класса **ViewController** вызван не был, так как мы увидели ту же самую сцену, что была загружена и отображена ранее. С ней не происходили никакие события, она не загружалась повторно, не удалялась из иерархии и не добавлялась в нее повторно. «Оранжевая» сцена находилась в иерархии даже тогда, когда отображалась вторая сцена.

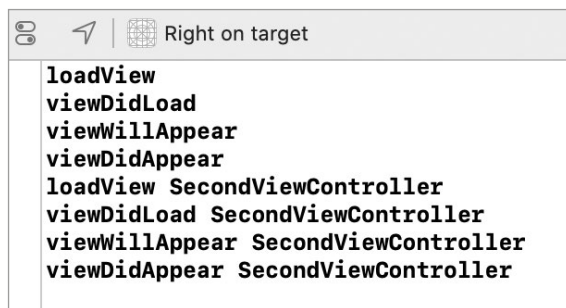


Рис. 3.25. Сообщения на консоли

Если сейчас посмотреть на иерархию графических элементов, мы увидим в ней всего одну сцену (рис. 3.26). И сколько бы мы не переходили между экранами, **present** всегда будет добавлять элементы сцены в иерархию, а **dismiss** – удалять.

При каждом переходе на вторую сцену на консоль выводятся сообщения методов **loadView** и **viewDidLoad** класса **SecondViewController**. Это говорит о том, что каждый раз происходит создание нового экземпляра данного типа.

Задание Подумайте, каким образом мы можем сделать так, чтобы сцена загружалась лишь один раз, а при повторных переходах загружалась из памяти? Напоминаю, что view контроллер удаляется из памяти, когда на него нет ссылок.

Для решения задания в классе **ViewController** создадим ленивое хранимое свойство типа **SecondViewController**, в который будет помещаться экземпляр

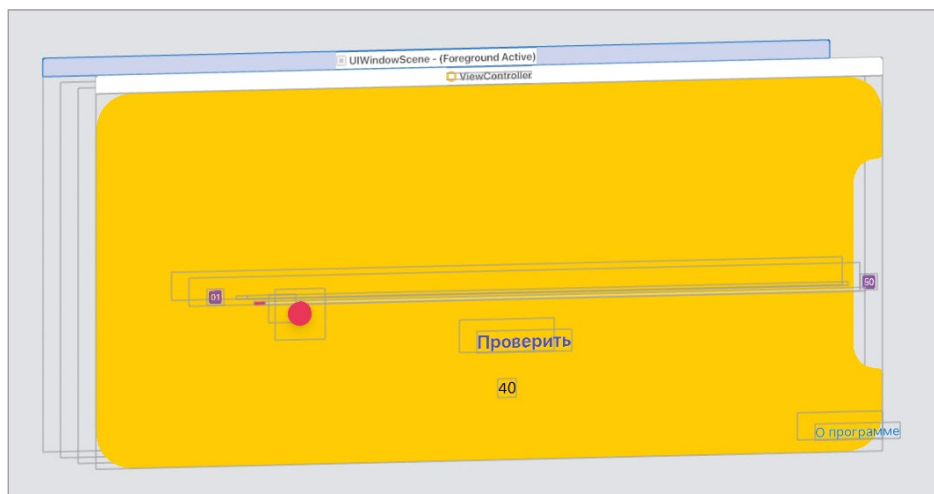


Рис. 3.26. Иерархия графических элементов

второй сцены (листинг 3.16). В этом случае вторая сцена загрузится лишь однажды, а при переходе к ней будут вызываться только методы **viewWillAppear** и **viewDidAppear**.

ЛИСТИНГ 3.16

```
// ленивое свойство для хранения View Controller
lazy var secondViewController: SecondViewController =
    getSecondViewController()

// приватный метод, загружающий View Controller
private func getSecondViewController() -> SecondViewController {
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    let viewController = storyboard.instantiateViewController(identifier:
        "SecondViewController")
    return viewController as! SecondViewController
}

@IBAction func showNextScreen() {
    self.present(secondViewController, animated: true, completion: nil)
}
```

Использование методов **present** и **dismiss** является прекрасным способом организации навигации между сценами.

Примечание Помимо **present** класс **UIViewController** включает еще и метод **show**, который также можно использовать для отображения сцены на экране.

Вариант смены сцены № 3. Navigation Controller и segue

Последним рассматриваемым вариантом перехода между сценами является использование навигационного контроллера и панели навигации. Вы неоднократно могли видеть ее, как в самой системе iOS, так и в различных приложениях (рис. 3.27).

Примечание В данном примере мы лишь немного познакомимся с тем, как работает навигационный контроллер. Более подробно данный элемент будет рассмотрен в следующих главах книги.

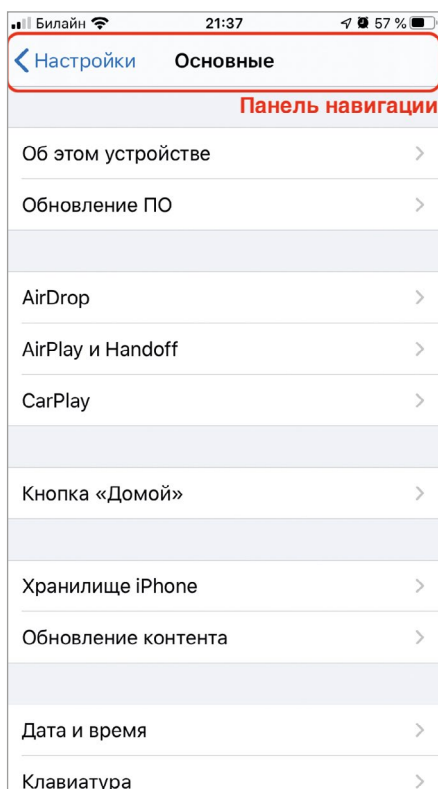


Рис. 3.27. Панель навигации в Настройках iOS

Добавим навигационный контроллер в проект.

- ▶ Удалите кнопку «Назад» со второй сцены.
- ▶ Удалите методы `showNextScreen` и `hideCurrentScene`.
- ▶ Удалите связь между кнопкой «О программе» и методом `showNextScreen`. Для этого используйте панель **Connection Inspector**, выделив кнопку.

- ▶ Создайте segue от кнопки «О программе» ко второй сцене.
- ▶ Выделите «оранжевую» сцену на **storyboard**.
- ▶ Выберите пункт главного меню **Editor > Embed In > Navigation Controller**.

Теперь на storyboard появился новый элемент – Navigation Controller (навигационный контроллер), связанный с «оранжевой» сценой (рис. 3.28).

Элемент Navigation Controller входит в состав фреймворка **UIKit**. Он представлен классом **UINavigationController**, потомком уже знакомого вам **UIViewController**. Основная идея Navigation Controller заключается в том, что он выступает в качестве контейнера, способного отображать внутри себя другие сцены, при этом автоматически управляя иерархией view (своевременно удаляя и добавляя элементы).

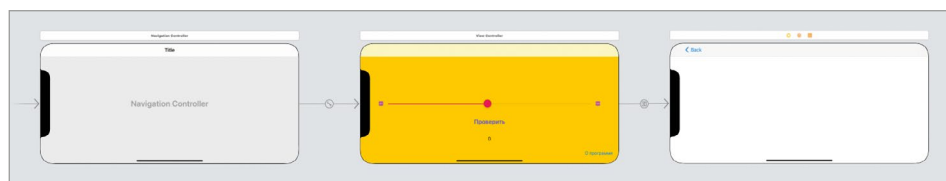


Рис. 3.28. Navigation Controller в составе проекта

- ▶ Запустите проект на симуляторе.

В интерфейсе приложения появился новый элемент – панель навигации в верхней части сцены (рис. 3.29). В данный момент панель пуста, но при переходе на новую сцену в ней появится кнопка для обратного перехода.

Вывод на консоли в текущий момент соответствует загрузке и отображению сцены. Посмотрим на то, как будут реагировать оба View Controller на переход ко второй сцене.

- ▶ Нажмите на кнопку «О программе».

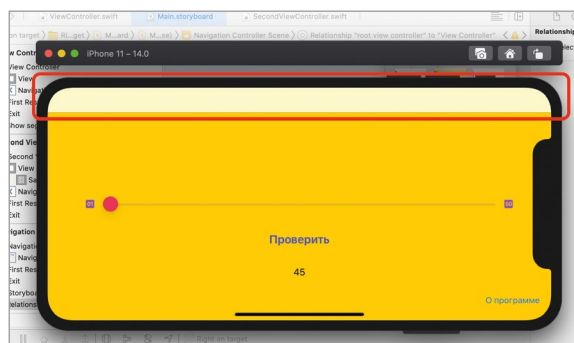
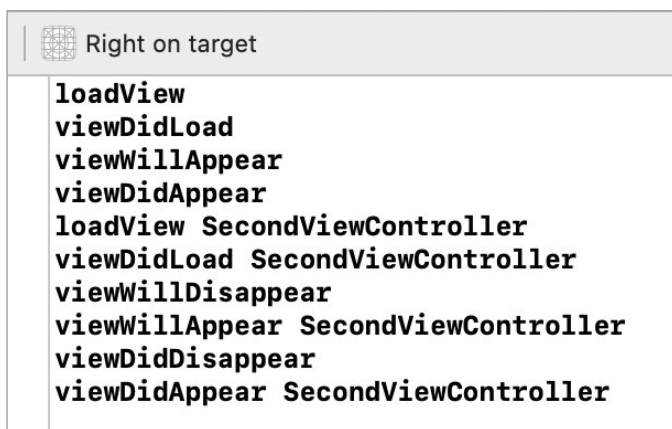


Рис. 3.29. Панель навигации на сцене

На рисунке 3.30 показан вывод на консоль. При переходе выполняется следующая последовательность шагов:

1. Загружается вторая сцена (**loadView** и **viewDidLoad** класса **SecondViewController**).
 2. Начинает скрываться первая сцена (**viewWillDisappear** класса **ViewController**).
 3. Начинает отображаться вторая сцена (**viewWillAppear** класса **SecondViewController**).
 4. Скрывается первая сцена (**viewDidDisappear** класса **ViewController**).
 5. Отображается вторая сцена (**viewDidAppear** класса **SecondViewController**).
- Нажмите на кнопку «< Назад», расположенную на панели навигации приложения.

Теперь вывод на консоль дополнился сообщениями, соответствующими вызовам методов **viewWillDisappear** и **viewDidDisappear** класса **SecondViewController**, после каждого из которых выведены сообщения, соответствующие методам **viewWillAppear** и **viewDidAppear** класса **ViewController**.



```
loadView
viewDidLoad
viewWillAppear
viewDidAppear
loadView SecondViewController
viewDidLoad SecondViewController
viewWillDisappear
viewWillAppear SecondViewController
viewDidDisappear
viewDidAppear SecondViewController
```

Рис. 3.30. Вывод на консоль

При движении вперед (т.е. при переходе к следующему экрану) с помощью Navigation Controller сцена всегда загружается заново (у соответствующего View Controller вызываются методы **loadView** и **viewDidLoad**). При движении назад (т.е. при переходе к предыдущему экрану) на экран выводится уже загруженная ранее сцена.

Навигационный контроллер производит довольно большую и очень полезную работу! При отображении новой сцены он всегда убирает текущую сцену из иерархии и добавляет туда ту, на которую осуществляется переход.

Примечание При использовании Navigation Controller вы можете создавать цепочки навигации любой глубины и разветвленности, а не только из двух сцен, как в данном проекте.

Очень важно, чтобы вы поняли, в какие моменты вызываются те или иные методы жизненного цикла View Controller. Эти знания будут активно использоваться вами в дальнейшем при разработке практически любого приложения.

Теперь вы можете удалить из проекта вторую сцену, кнопку «**О программе**», а также Navigation Controller. Но не забудьте установить флажок «**Is Initial View Controller**» для первой сцены на панели **Attributes Inspector**. В противном случае, приложение не будет знать, какой view контроллер загрузить первым.

Глава 4.

Рефакторинг программного кода

В этой главе вы:

- узнаете, что такое рефакторинг;
- проведете рефакторинг кода приложения «**Right on target**»;
- выпустите несколько новых версий приложения «**Right on target**».

В своей книге «Implementation Patterns» Кент Бек сказал: «... эта книга базируется на довольно непрочном утверждении, что хороший код важен ...». Но на мой взгляд, хороший код действительно важен.

Но как сделать чтобы код стал по-настоящему хорошим? Единственный верный способ — не прекращать получать опыт разработки. Такой подход позволит со временем по-новому взглянуть на написанный ранее код, переосмыслить его, найти новые, порой самые неожиданные способы решения старых задач.

Вы как разработчик должны постоянно улучшать качество программ и стараться писать действительно хороший код. Если бы качество вашего кода не было настолько важным, то в природе не существовало бы такого понятия, как **рефакторинг**.

Эта глава посвящена рефакторингу приложения «**Right on target**». Сложность учебного материала будет постепенно повышаться, а некоторые из заданий возможно окажутся довольно трудными, и вы не сможете их выполнить самостоятельно с первой попытки. Помните: цель состоит не в том, чтобы выполнить все практические задания верно, а попытаться это сделать! В случае возникновения проблем изучайте мои варианты решения, доступные по ссылке ниже.



В этой главе мы продолжим работу над программой «Right on target» версии 1.1.

Скачать ее можно, перейдя по ссылке: <https://swiftme.ru/listings21>

4.1 Рефакторинг программного кода

Рефакторинг — это анализ и переработка структуры и состава программного кода, а также других элементов приложения, с целью создания более простых для понимания и обслуживания компонентов.

Зачем нужен рефакторинг? Далеко не всегда удастся добиться того, чтобы код проекта изначально был стройным, красивым и функциональным. Основная задача рефакторинга заключается в том, чтобы сделать его таковым.

В процессе рефакторинга программист анализирует уже написанный код и при необходимости перерабатывает его:

- удаляет неиспользуемые участки кода;
- переименовывает элементы (классы, структуры, параметры и т.д.);
- обеспечивает переиспользуемость компонентов в будущем и исключает дублирование кода;
- разбивает большие функции и методы на более мелкие;
- оптимизирует объектные типы, выделяя новые сущности;
- дополнительно комментирует сложные участки кода.

Когда стоит делать рефакторинг? Ответ однозначный — при любой возможности по чуть-чуть. Таким образом, будет минимизирована возможность внесения неконтролируемых изменений.

Рефакторинг — это изобретение велосипеда раз за разом. Но с каждой новой итерацией ваш велосипед будет лучше управляться, обладать меньшей массой и мчаться быстрее.

В этой главе мы выпустим несколько новых версий проекта «**Right on target**», выделив новые сущности и проведя рефакторинг кода, а в последнем разделе вы самостоятельно создадите новый режим игры, в котором будет необходимо верно определить цвет по его коду.

4.2 «Right on target», версия 1.2

Несмотря на то, что «**Right on target**» в полной мере играбельна, ее внутренняя структура далека от совершенства. Более того, она просто кричит: улучшите меня!

Ранее, мы рассмотрели паттерн MVC, но View Controller приложения, который должен быть «прослойкой» между Представлением и Моделью, все еще выполняет не свойственные ему функции. В проекте отсутствует Модель, в

результате чего вся нагрузка по реализации бизнес-логики ложится именно на плечи Контроллера.

Первая задача, которую нам необходимо решить в рамках рефакторинга – разработать Модель.

Разработка Модели

Для хранения файлов Модели в составе проекта выделим специальную папку с соответствующим названием.

- ▶ В **Project Navigator** щелкните правой кнопкой мыши по папке с названием «**Right on target**».
- ▶ В появившемся списке выберите пункт **New Group**.
- ▶ Измените название созданной папки на «**Model**».

В составе проекта появилась новая папка (рис. 4.1).

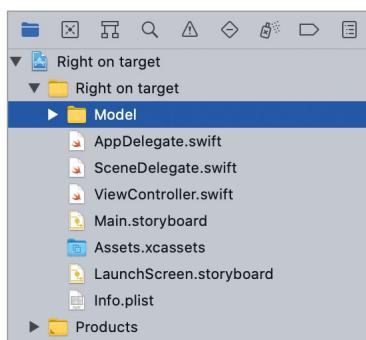


Рис. 4.1. Состав проекта «Right on target»

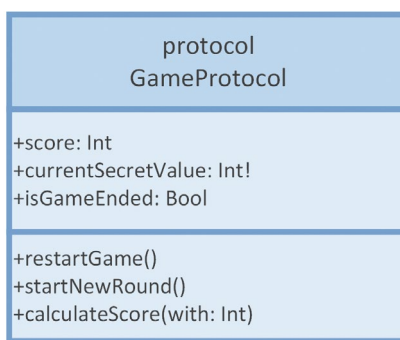


Рис. 4.2. Схематичный вид протокола

В первую очередь, для облегчения процесса разработки мы вынесем всю бизнес-логику из класса **ViewController**. Для этого создадим новый класс **Game**, который будет представлять из себя программную реализацию сущности «Игра». Он прекрасно подойдет для хранения бизнес-логики приложения.

- ▶ В папке **Model** создайте новый файл с исходным кодом с именем **Game.swift**.

Примечание В соответствии с принципами протокол-ориентированного программирования (мы говорили о них в первой книге), разработку любой сущности необходимо начинать с протокола.

В первую очередь реализуем протокол **GameProtocol**, который послужит основой для класса **Game**. На рисунке 4.2 схематично отражена структура будущего протокола.

Примечание Схема, приведенная на рисунке 4.2, разработана по стандарту UML. UML – это язык графического описания объектной модели программного кода. С его помощью можно схематично изобразить все сущности и их зависимости. Я советую вам потратить время на знакомство с ним.

Прямоугольник целиком описывает какую-либо сущность. В данном случае, это протокол **GameProtocol**, о чем сказано в его верхней части. В центральной части приводится перечень свойств, а в нижней – перечень методов. Символ «+» перед названием свойств и методов указывает на то, что эти элементы являются публичными (public), а символ «-» наоборот говорит об их приватности (private).

Рассмотрим протокол **GameProtocol**:

- Свойство **score**

Содержит количество очков, заработанных пользователем за все завершенные раунды игры. Будет использоваться для отображения очков в **UIAlertController** по окончании игры.

- Свойство **currentSecretValue**

Содержит текущее загаданное программой число, которое необходимо указать пользователю с помощью слайдера. Будет использоваться для отображения числа в Label.

- Свойство **isGameEnded**

Флаг, позволяющий узнать, закончилась ли игра (завершены ли все раунды). Будет использоваться для того, чтобы определить, когда отображать **UIAlertController** и начинать новую игру.

- Метод **restartGame**

Позволяет перезапустить уже начатую игру (сбросить счет, счетчик раундов и загадать новое число).

- Метод **startNewRound**

Позволяет начать новый раунд игры (загадать новое число).

- Метод **calculateScore**

Подсчитывает количество заработанных за раунд очков.

По моим оценкам, указанных методов и свойств будет достаточно для выноса всей логики игры в Модель.

Примечание Набор элементов любого протокола можно оспорить, так как их выделяет разработчик, а этот процесс довольно субъективный. Каждый из нас сам определяет логику работы с сущностью, а, соответственно, и что будет содержаться в протоколе. Если у вас есть свой взгляд на структуру **GameProtocol**, реализуйте ее – это послужит вам прекрасным опытом.

- В файле **Game.swift** реализуйте протокол **GameProtocol**, как это показано в листинге 4.1.

ЛИСТИНГ 4.1

```
protocol GameProtocol {
    // Количество заработанных очков
    var score: Int { get }
    // Загаданное значение
    var currentSecretValue: Int { get }
    // Проверяет, закончена ли игра
    var isGameEnded: Bool { get }
    // Начинает новую игру и сразу стартует первый раунд
    func restartGame()
    // Начинает новый раунд (обновляет загаданное число)
    func startNewRound()
    // Сравнивает переданное значение с загаданным и начисляет очки
    func calculateScore(with value: Int)
}
```

Теперь перейдем к непосредственной реализации типа **Game**.

По вашему мнению, его необходимо реализовать с помощью класса (**class**) или структуры (**struct**)?

С учетом текущей сложности проекта (он содержит всего одну сцену и довольно простую бизнес-логику) выбор между классом или структурой не будет иметь какого-либо значения. Экземпляр типа будет храниться только в свойстве класса **ViewController** и функционировать исключительно в пределах одной сцены. При необходимости получить доступ к экземпляру мы всегда будем обращаться к одному и тому же свойству класса **ViewController**, а значит и к одному и тому же значению.

Но взглянем на вопрос шире. Вполне вероятно, что в дальнейшем игра будет развиваться и совершенствоваться, в ней появятся новые экраны и возможности. В определенный момент вам может потребоваться всегда иметь доступ к текущей игре (экземпляру типа **Game**, описывающего игру) не только с главного экрана, но и с одного из дочерних, например, для изменения его настроек.

При использовании структуры сущность будет передаваться копированием, а значит все изменения будут вноситься не в текущую «Игру», а в ее копию. В этом случае потребуются реализация различных механизмов передачи данных между контроллерами и обработка данных настроек для их применения в текущей игре.

Если выбрать класс, мы с легкостью сможем модифицировать тот самый экземпляр, описывающий текущую игру, с любого другого экрана. Именно по этой причине мы реализуем тип **Game** в виде класса.

► В файле **Game.swift** реализуйте класс **Game** (листинг 4.2).

ЛИСТИНГ 4.2

```
class Game: GameProtocol {

    var score: Int = 0
    // Минимальное загаданное значение
    private var minSecretValue: Int
    // Максимальное загаданное значение
    private var maxSecretValue: Int
    var currentSecretValue: Int = 0
    // Количество раундов
    private var lastRound: Int
    private var currentRound: Int = 1
    var isGameEnded: Bool {
        if currentRound >= lastRound {
            return true
        } else {
            return false
        }
    }

    init?(startValue: Int, endValue: Int, rounds: Int) {
        // Стартовое значение для выбора случайного числа не может быть
        // больше конечного
        guard startValue <= endValue else {
            return nil
        }
        minSecretValue = startValue
        maxSecretValue = endValue
        lastRound = rounds
        currentSecretValue = self.getNewSecretValue()
    }

    func restartGame() {
        currentRound = 0
        score = 0
        startNewRound()
    }
}
```

```

    func startNewRound() {
        currentSecretValue = self.getNewSecretValue()
        currentRound += 1
    }

    // Загадать и вернуть новое случайное значение
    private func getNewSecretValue() -> Int {
        (minSecretValue...maxSecretValue).randomElement()!
    }

    // Подсчитывает количество очков
    func calculateScore(with value: Int) {
        if value > currentSecretValue {
            score += 50 - value + currentSecretValue
        } else if value < currentSecretValue {
            score += 50 - currentSecretValue + value
        } else {
            score += 50
        }
    }
}

```

ПРИМЕЧАНИЕ Ответственность за то, как именно в объектном типе реализовать функциональность, основа которой заложена в протоколе – это ответственность разработчика. Если вы видите более интересные пути реализации класса **Game**, то попробуйте написать его код самостоятельно.

Потратьте время и разберите каждое свойство и метод класса **Game**. В них нет абсолютно ничего сложного.

Переработка Контроллера

Работа с классом **Game** завершена, и вся бизнес-логика перенесена в него. Теперь нам необходимо переработать код вью контроллера таким образом, чтобы в нем не осталось и намека на перенесенную в **Game** функциональность. Контроллер должен использовать возможности Модели.

Задание Самостоятельно перепишите тело класса **ViewController**, убрав из него всю перенесенную функциональность и добавив работу с Моделью.

Игра должна начинаться сразу после появления интерфейса на экране устройства, а по окончании (когда закончился последний раунд) — сообщать результат во всплывающем окне и предлагать начать игру заново.

В листинге 4.3 показан мой вариант реализации класса **ViewController**.

ЛИСТИНГ 4.3

```
class ViewController: UIViewController {

    // Сущность "Игра"
    var game: Game!

    // Элементы на сцене
    @IBOutlet var slider: UISlider!
    @IBOutlet var label: UILabel!

    // MARK: - Жизненный цикл

    override func viewDidLoad() {
        super.viewDidLoad()
        // Создаем экземпляр сущности "Игра"
        game = Game(startValue: 1, endValue: 50, rounds: 5)
        // Обновляем данные о текущем значении загаданного числа
        updateLabelWithSecretNumber(newText: String(game.
currentSecretValue))
    }

    // MARK: - Взаимодействие View - Model

    // Проверка выбранного пользователем числа
    @IBAction func checkNumber() {
        // Вычисляем очки за раунд
        game.calculateScore(with: Int(slider.value))
        // Проверяем, окончена ли игра
        if game.isGameEnded {
            showAlertWith(score: game.score)
            // Начинаем игру заново
            game.restartGame()
        } else {
            game.startNewRound()
        }
        // Обновляем данные о текущем значении загаданного числа
        updateLabelWithSecretNumber(newText: String(game.
currentSecretValue))
    }

    // MARK: - Обновление View
    // Обновление текста загаданного числа
    private func updateLabelWithSecretNumber(newText: String ) {
```



```

        label.text = newText
    }

    // Отображение всплывающего окна со счетом
    private func showAlertWith(score: Int) {
        let alert = UIAlertController(
            title: "Игра окончена",
            message: "Вы заработали \(score) очков",
            preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "Начать заново", style:
.default, handler: nil))
        self.present(alert, animated: true, completion: nil)
    }
}

```

Обратите внимание на то, что:

- вся логика по обновлению графических элементов (взаимодействию с View), выведена в отдельные методы;
- конструкция **MARK**: позволяет визуально отделять блоки кода в навигаторе и панели быстрого перехода (рис. 4.3). Это прекрасное средство для улучшения навигации по коду.

Мы достигли поставленной перед нами цели: в проекте появилась Модель, а Контроллер избавился от лишней функциональности. Все что он делает — это взаимодействует с Моделью и Представлением, осуществляя обмен данными между ними.

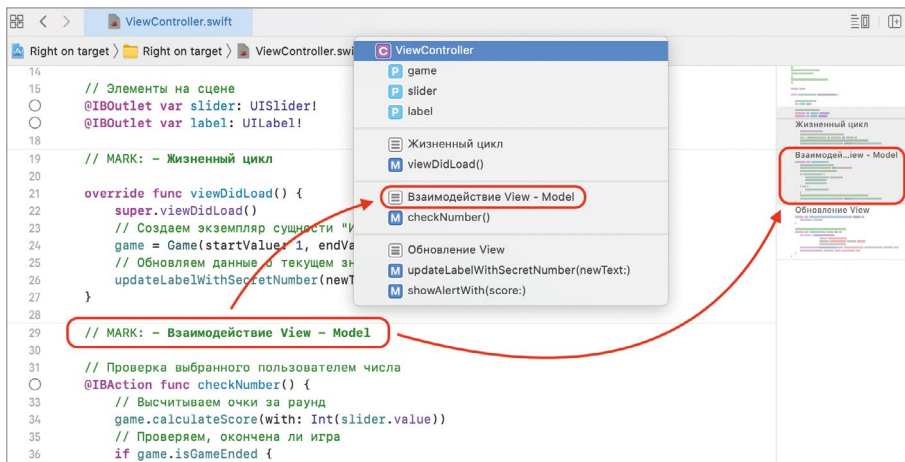


Рис. 4.3. Разделы при навигации по коду

Внесем финальные правки в проект:

- Измените версию проекта на 1.2.
- Запустите проект на симуляторе и проверьте его работоспособность.

Мои поздравления! Это был ваш первый рефакторинг (или, другими словами, анализ и доработка) программного кода. Самое важное, чтобы он не стал для вас последним. Благодаря этому процессу вы можете значительно повысить свой уровень владения языком и создавать элементы, которые можно будет использовать вновь в дальнейшем.

С точки зрения пользователя, программа не стала лучше или хуже: она все так же выполняет поставленные перед ней задачи. Но в будущем, если вы захотите расширить возможности проекта (а вы захотите), это будет сделать значительно проще.

MVC, как и любой другой шаблон проектирования, привносит в ваше приложение логичность и удобство. Создавая новый элемент, всегда задавайтесь вопросом: какую роль он должен исполнять и где он должен находиться.

4.3 «Right on target», версия 1.3. Самостоятельная работа

Выполненная переработка проекта «**Right on target**» сделала его более интересным с точки зрения архитектуры программного кода, а также снизила стоимость дальнейшего развития. К примеру, если бы вам потребовалось изменить алгоритм выбора случайного числа, сделать это можно всего лишь изменив код в методе `getNewSecretValue` класса `Game`. Просто и удобно! А вот в версии 1.1 для этого пришлось бы вносить правки как минимум в двух местах, контролируя при этом их идентичность. Благодаря проделанной работе вы экономите время, а значит можете сделать значительно больше за гораздо меньшее время. Тем не менее, код проекта еще далек от идеала.

Рассмотрим один пример. Представьте, что перед вами возникла задача реализовать алгоритм выбора случайного числа с использованием аппаратной платы генерации псевдослучайных чисел, а не просто метода `randomElement()`. В подавляющем большинстве случаев разработчик вел бы работу по следующему плану:

1. Закомментировал текущую реализацию метода `getNewSecretValue` класса `Game`, возвращающую случайное число. Зачем закомментировал? На всякий случай, чтобы не потерять код, на случай, если потребуется все вернуть назад.
2. Реализовал новый метод `getNewSecretValue`, использующий аппаратную плату.

В результате такого подхода старый закомментированный код останется лежать в коде «мертвым грузом», ожидая, что когда-то он возможно будет вновь использован. И чем больше подобных задач поступает, тем больше будет разрастаться класс **Game**.

Примечание Если у вас уже есть опыт разработки на других языках, то уверен, что ситуация типа «закомментирую и оставлю на всякий» вам очень знакома.

Почему так происходит?

Прежде всего, разработку любого проекта стоит начинать с планирования структуры и выделения основных сущностей, так как каждая отдельная функция приложения должна решаться отдельным компонентом. В нашем случае, реализовав в проекте Модель, мы создали «класс-комбайн» **Game**, выполняющий слишком большой спектр задач. Данный класс решает совершенно все вопросы, связанные с игрой.

Почему класс **Game** отвечает за выбор случайного числа? Не правильнее ли вынести эту функциональность в отдельную сущность? Более того, пусть класс **Game** вообще отвечает только за игру в целом, ведь за отдельные раунды игры может отвечать сущность «Раунд».

Теперь при необходимости создать новый «Генератор случайных чисел» мы реализуем новый тип на основе уже существующего протокола, после чего его экземпляр передадим в **Game**. А при необходимости откатиться – воспользуемся старым типом. Удобно!

Задание В проекте «Right on target» реализуйте сущности «Раунд» и «Генератор случайных целых чисел», перенеся в них соответствующую функциональность из класса **Game**.

На рисунке 4.4 я привел схему своего варианта реализации протоколов. Тем не менее, вы имеете полную свободу действий, и можете сделать задание полностью по-своему.

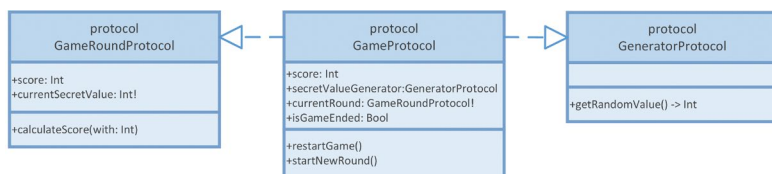


Рис. 4.4. Схематичный вид протоколов



Мой вариант решения вы можете найти, перейдя по этой ссылке:
<https://swiftme.ru/listings21>

4.4 «Right on target», версия 1.4. Самостоятельная работа

Мы выпустили уже три обновленные версии приложения «**Right on target**». То, как мы дорабатываем программу – просто чудесно, но настоящая магия ждет вас впереди. В этой главе вы попробуете самостоятельно добавить в игру новый режим «**Select color**» (в переводе на русский – «**Выбери цвет**») – этакий тренажер для дизайнеров и верстальщиков (рис. 4.5).

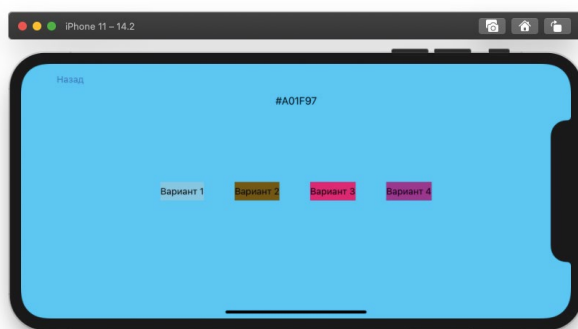


Рис. 4.5. Интерфейс игры по выбору цвета

Цель нового режима заключается в том, что пользователю предлагается выбрать цвет, соответствующий отображенному в текстовой метке на сцене HEX-коду. Для выбора варианта цвета пользователем могут быть использованы обычные кнопки (**UIButton**) с различным фоновым цветом. Одна из кнопок всегда показывает правильный вариант, а другие окрашены в случайный цвет.

Примечание Одним из вариантов кодирования цвета является использование HEX-кода – шестизначной последовательности букв и цифр, начинающегося с символа # (например, белый – это **#ffffff**, черный – **#000000**, а красный – **#ffa0b2**). Данная последовательность содержит три двухзначных числа в шестнадцатеричной системе счисления (цифры от 0 до F), объединенных в одно строковое значение. Для того, чтобы определить по HEX-коду цвет, компьютер разбивает его на три пары символов (например, для **#ffa0b2**, розовый цвет – это ff, a0 и b2), каждая из которых определяет интенсивность цвета для одного из каналов в схеме RGB (Red-Green-Blue).

Задание Добавьте в приложение дополнительный вариант игры, в котором пользователю показывается HEX-код случайного цвета и 4 варианта на выбор. При выборе правильного варианта пользователь получает одно очко. После окончания игры программа показывает информацию о количестве правильно выбранных цветов и предлагает (по аналогии с игрой про выбор числа) начать угадывать заново.

Фантазируйте и используйте все свои идеи. И помните, что суть обучения заключается не в том, чтобы вы выполнили задание, а в том, чтобы вы по-

старались его сделать, потратили время на рассуждения и самостоятельные попытки его решения!

В процессе выполнения задания вам самостоятельно потребуется рассмотреть работу с типом **UIColor**, позволяющим описать требуемый цвет в коде.

На рисунке 4.6 показан сториборд моего варианта решения, который вы можете найти в репозитории с листингами из книги на **GitHub**.

Для выбора режима игры я добавил новый стартовый выю контроллер и разместил на нем две кнопки, при нажатии на которые происходит переход к игре «Right on target» или «Select color».

Наиболее интересным в приведенном мной варианте решения является исходный код. При создании Модели я использовал возможности дженериков и создал такие универсальные типы данных, которые смогли бы обеспечить оба режима игры. Суть такого подхода заключается в том, что обе игры, по своей сути, являются одним и тем же. Так как Представление (MVC) в нашем проекте отделено от Модели, то выбор верного цвета с помощью кнопок с точки бизнес-логики ничем не отличается от выбора числа с помощью слайдера. Подумайте об этом и вы обязательно сможете создать хорошую универсальную переиспользуемую архитектуру.



Мой вариант решения вы можете найти, перейдя по этой ссылке:
<https://swiftme.ru/listings21>

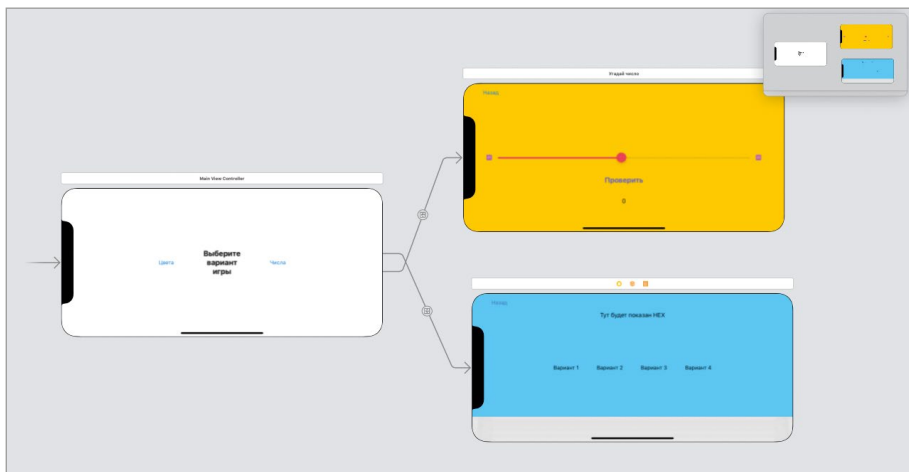


Рис. 4.6. Сториборд игры

Глава 5.

Структура и запуск iOS-приложения

В этой главе вы:

- узнаете, как запускается приложение в среде операционной системы;
- познакомитесь с базовыми классами, из которых состоит приложение;
- изучите жизненный цикл приложения и его компонентов.



Далее мы продолжим работу над программой «Right on target» (любой из доступных вам версий).

Скачать ее можно, перейдя по ссылке: <https://swiftme.ru/listings21>

Рассмотрев жизненный цикл View Controller, вы стали немного ближе к пониманию того, как именно функционируют приложения и их отдельные элементы. Теперь мы поднимемся на уровень выше и поговорим о том, чем представлено приложение в среде операционной системы, как оно запускается (что является стартовой точкой) и как отображается его интерфейс.

5.1 Класс UIApplication

По умолчанию любое установленное приложение не запущено. Оно просто хранится на устройстве в виде двоичных файлов и различных ресурсов (картинок, текстовых файлов и т.д.), и никакие процессы, связанные с приложением, не происходят. Но стоит пользователю нажать на иконку приложения, как операционная система производит его запуск, загрузку и отображает интерфейс на экране.

Примечание Существует множество причин, по которым приложение может быть запущено: нажатие на иконку на домашнем экране, поступление push-уведомления, перехват ссылки, меню быстрых действий и др. В одних случаях будет загружен и отображен графический интерфейс, а в других вся работа будет выполняться в фоновом режиме.

В этой главе мы будем рассматривать процесс запуска после нажатия на иконку на домашнем экране.

Когда операционной системе (в данном случае мы говорим исключительно про iOS и iPadOS, но не macOS) требуется произвести запуск приложения, происходит автоматический вызов глобальной функции **UIApplicationMain**. В результате вызова возвращается экземпляр класса **UIApplication**, который является программной реализацией сущности «Приложение» и описывает запущенное приложение. Этот экземпляр является точкой входа в приложение: через него проходят все команды и события, т.е. с его помощью операционная система общается с приложением.

Примечание **UIApplicationMain** – это функция, которая используется исключительно операционной системой. У вас нет причин пытаться вызвать ее.

Примечание Функция **UIApplicationMain** и класс **UIApplication** входят в состав фреймворка **UIKit**. Данный фреймворк глубоко интегрирован в операционную систему. Именно он во многом обеспечивает функционирование приложений.

Для каждого запущенного приложения в операционной системе существует собственный экземпляр класса **UIApplication**. **Запомните: каждое запущенное приложение имеет ровно один экземпляр UIApplication!**

Примечание Класс **UIApplication** написан на основе шаблона проектирования **Singleton** (Одиночка). Благодаря этому при каждом обращении к его экземпляру возвращается одно и то же значение. Мы уже говорили о данном шаблоне в первой книге, и в настоящий момент нет необходимости более подробно останавливаться на нем: нам важно знать лишь то, что в нашем приложении есть только один экземпляр **UIApplication**. И это является очень логичным решением, так как в этом случае приложение имеет только одну точку входа.

У вас может появиться необходимость получить доступ к экземпляру приложения. Для этого необходимо использовать выражение **UIApplication.shared**, которое всегда возвращает ссылку текущий экземпляр. В ходе дальнейшего обучения вы узнаете несколько ситуаций, в которых оно может пригодиться. В частности, с помощью **UIApplication.shared.delegate** можно обратиться к делегату приложения (экземпляру класса **AppDelegate**).

В задачи класса **UIApplication** входит координация работы приложения не только при его запуске, но и в ходе дальнейшего функционирования. Все события, происходящие с приложением в рамках его работы, берут свое начало именно в **UIApplication**.

На рисунке 5.1 показана схема жизненного цикла приложения с учетом рассмотренного материала.

В самом простом случае приложение может иметь всего два состояния: «Запущено» и «Не запущено». Но за сменой состояний кроется довольно большая работа, производимая в автоматическом режиме операционной системой и компонентами, входящими в состав приложения (один из них – это View Controller).

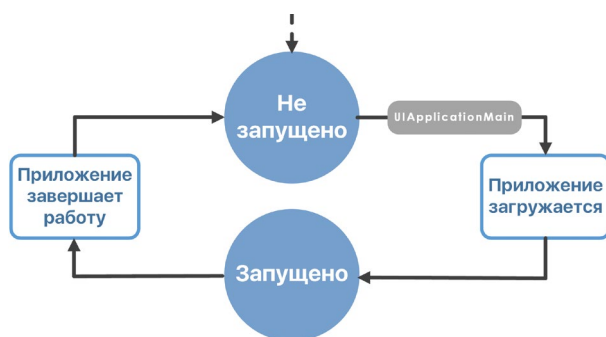


Рис. 5.1. Жизненный цикл приложения

Резюмируем. Когда приложение должно быть запущено, операционная система производит вызов функции **UIApplicationMain**. Возвращенный после вызова экземпляр типа **UIApplication** олицетворяет собой запущенное приложение. Он хранится в оперативной памяти и используется операционной системой при необходимости передачи в приложение дополнительных данных (например, информации о событии касания экрана, когда интерфейс приложения отображается, или поступившего push-уведомления). Также разработчик может сам обратиться к экземпляру своего приложения в любой точке программы с помощью конструкции **UIApplication.shared**.

5.2 Паттерн Делегирование и класс AppDelegate

В процессе разработки приложений вам предстоит использовать множество различных шаблонов проектирования, одним из которых является «Делегирование» (Delegation).

Примечание Если какая-то часть материала, приведенного в главе, покажется вам не до конца понятной, не переживайте: в третьей части этой книги мы вновь вернемся к данному шаблону и рассмотрим его очень подробно.

Паттерн «Делегирование» предполагает, что ответственность за выполнение каких-либо задач передается (делегруется) от одного объекта другому.

Чтобы лучше понять его, рассмотрим пример из реальной жизни. Предположим, что вы создали стартап по разработке мобильных приложений. Вы оформились как индивидуальный предприниматель и с успехом выполняете все задачи самостоятельно: от финансовых вопросов до написания программного кода. Время идет, компания растет, количество заказов увеличивается, но вместе с этим увеличивается и количество встающих перед вами задач.

В какой-то момент вы замечаете, что с трудом справляетесь, и вам требуется помощь. Вы нанимаете нового сотрудника и передаете (**делегируете**) ему часть своих обязанностей, например, вопросы бухгалтерии. Теперь при поступлении финансовых документов они незамедлительно передаются на исполнение новому сотруднику (вашему делегату), он несет полную ответственность за решение данной задачи. Данный сотрудник – ваш делегат, т.е. вы делегировали ему полномочия в определенной сфере для решения определенных вопросов.

В состав приложения входит огромное количество элементов, каждый из которых решает те или иные задачи. В ряде случаев для некоторых из этих элементов могут быть определены делегаты. В большинстве из них это делается из-за того, что у вас нет доступа к самому элементу для его модификации. Вы не можете изменить его поведение или исходный код напрямую, но с помощью делегата у вас появляется такая возможность.

Например, класс **UIApplication**. Он входит в состав **UIKit**, и у вас нет возможности внести в него какие-либо корректировки, чтобы как-то повлиять на процесс запуска приложения. Но для экземпляра этого класса в приложении определен класс-делегат, который позволяет выполнить необходимый код в определенные моменты жизни приложения.

- ▶ Откройте проект «**Right on target**».
- ▶ В **Project Navigator** откройте файл **AppDelegate.swift**.

В файле **AppDelegate.swift** объявлен класс **AppDelegate**, который является делегатом экземпляра класса **UIApplication**, соответствующего приложению «**Right on target**». В определенные моменты жизненного цикла операционная система обращается не к классу **UIApplication**, а к классу **AppDelegate**, вызывая его методы.

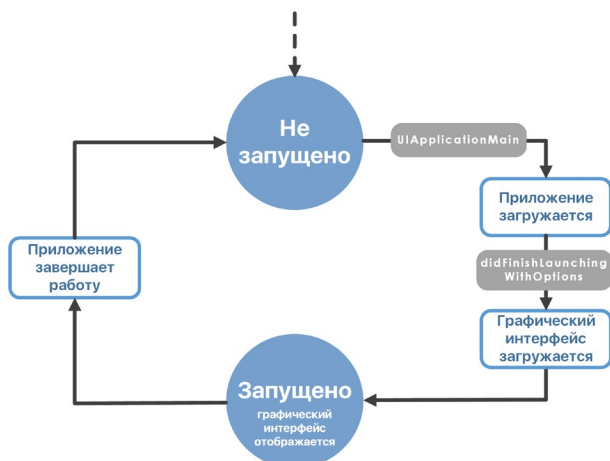


Рис. 5.2. Жизненный цикл приложения

Как вы можете видеть, в классе **AppDelegate** уже реализован метод **application(_:didFinishLaunchingWithOptions:)**. Вызов данного метода происходит сразу после того, как приложение загружено, но до того, как начнется загрузка пользовательского интерфейса (рис. 5.2).

Примечание Методы многих классов, используемых в процессе разработки приложений, имеют довольно сложную сигнатуру, в частности, из-за длинных имен входных параметров. В таких случаях следует использовать сокращенные имена, указывая только имя уникального параметра. Так, **application(_:didFinishLaunchingWithOptions:)** превратится в **didFinishLaunchingWithOptions**. Но вы должны ясно понимать, о каком конкретно методе идет речь.

С помощью автодополнения вы всегда можете восстановить полную сигнатуру функции в вашем коде. Для этого достаточно начать вводить имя уникального параметра, после чего Xcode подскажет все подходящие доступные элементы, среди которых будет и искомый метод. На рисунке 5.3 показан пример ввода строки **didFinish**. Во всплывающем окне отображается два метода:

- **applicationDidFinishLaunching(:)**, используемый при работе с watchOS;
- **application(_:didFinishLaunchingWithOptions:)**.

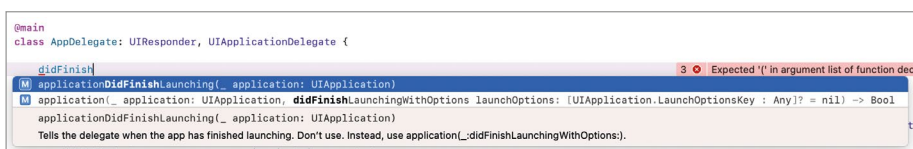


Рис. 5.3. Автодополнение Xcode

Обратите внимание, что в примере метод **application(_:didFinishLaunchingWithOptions:)** показан в окне автодополнения, так как я сознательно удалил его реализацию в классе **AppDelegate** с целью продемонстрировать данную возможность. В вашем случае этот метод уже реализован в классе, поэтому во всплывающем окне будет только **applicationDidFinishLaunching(:)**.

Используя метод **didFinishLaunchingWithOptions**, вы можете внести изменения в процесс загрузки приложения, которые могут быть использованы во всей программе. К примеру, вы можете активировать подключение к базе данных или запустить загрузку каких-либо ресурсов.

Помимо **didFinishLaunchingWithOptions** вам также доступен метод **willFinishLaunchingWithOptions**, который, как следует из названия, вызывается до завершения загрузки приложения. Но немкакио **configurationForConnecting**, и **didDiscardSceneSessions** (уже определены в классе **AppDelegate**) и других методах, не стоит задумываться на данном этапе обучения.

- Добавьте в метод **didFinishLaunchingWithOptions** вызов функции **print** так, как показано в листинге 5.1.

ЛИСТИНГ 5.1

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [UIApplication.
LaunchOptionsKey: Any]?) -> Bool {
    print("applicationDidFinishLaunching")
    return true
}
```

- ▶ Проверьте, есть ли в методах жизненного цикла класса **ViewController** вызовы функции **print**. Мы добавляли их в предыдущих главах, чтобы определить порядок вызова методов. Если их нет, то их требуется добавить.
- ▶ Запустите приложение на симуляторе.

В процессе запуска приложения на консоль будет выведена надпись **applicationDidFinishLaunching**, а уже после нее сообщения, соответствующие вызовам методов жизненного цикла View Controller (рис. 5.4).



```
applicationDidFinishLaunching
loadView
viewDidLoad
viewWillAppear
viewDidAppear
```

Рис. 5.4. Вывод на консоли

Примечание Приложение знает, что класс **AppDelegate** является его делегатом, благодаря тому, что данный класс имеет атрибут **@UIApplicationMain** (рис. 5.5).

В зависимости от того, какую версию Xcode вы используете, **AppDelegate** может иметь и другой атрибут — **@main**.



```
9 import UIKit
10
11 @UIApplicationMain
12 class AppDelegate: UIResponder, UIApplicationDelegate {
13
14     func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
```

Рис. 5.5. Атрибут класса-делегата

Резюмируем. После создания экземпляра класса **UIApplication** приложение создает экземпляр класса **AppDelegate**, который является делегатом **UIApplication**. В нем могут быть определены методы, вызываемые в тот или иной момент жизни приложения.

5.3 Классы UIWindowScene и SceneDelegate

После того, как приложение загружено (созданы экземпляры типов **UIApplication** и **AppDelegate**), начинается процесс подготовки и отображения графического интерфейса.

Вместе с выходом iPadOS (соответствует iOS 13) у iPad появилась поддержка многооконного режима, позволяющего на одном устройстве одновременно держать запущенными нескольких экземпляров графического интерфейса одного и того же приложения. На рисунке 5.6 показан браузер Safari, запущенный в двухоконном режиме. В данном случае, Safari содержит две независимые области, отображающие графический интерфейс приложения, или, другими словами, два независимых экземпляра графического интерфейса.

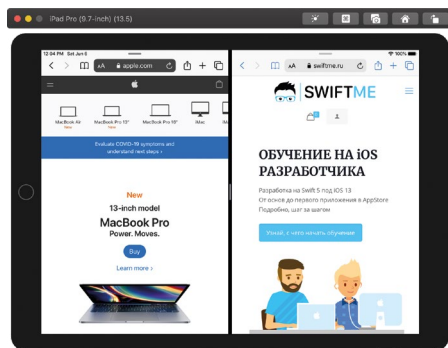


Рис. 5.6. Двухоконный режим iPadOS

Такие области (или экземпляры интерфейса) называются **сценами**. Ранее мы уже встречались с этим понятием, когда говорили о View Controller и сцене, отображение которой он обеспечивает.

Можно сказать, что сцена в контексте приложения — это экземпляр графического интерфейса, независимая область для отображения, а сцена View Controller — это непосредственно сам интерфейс, который отображается в этой области. Это два близких, но все же отличающихся понятия. Мы будем говорить о сценах и в том, и в другом контексте.

Каждый экземпляр графического интерфейса представлен в приложении экземпляром класса **UIWindowScene**. Он обеспечивает отображение сцены на экране устройства, а также контролирует то, что происходит с ней. На iPhone (в iOS) не поддерживается многооконный режим, но используются те же самые программные компоненты, что и в iPadOS, просто там не может отображаться больше одной сцены за раз. Важно запомнить, что для запущенного приложения, независимо от количества открытых окон, существует только

3. Отображена и доступна для взаимодействия (Foreground Active) — сцена отображается на экране и доступна для взаимодействия, т.е. пользователь может взаимодействовать с графическими элементами сцены.

4. Не отображена (Background) — сцена скрыта с экрана, но продолжает выполнять код в фоновом режиме.

5. Приостановлена (Suspended) — сцена больше не может выполнять фоновую работу. Она находится в памяти до тех пор, пока не будет удалена или вновь переведена в другое состояние.

Сцена (экземпляр класса **UIWindowScene**), так же как и приложение в целом (экземпляр класса **UIApplication**), имеет собственный класс-делегат, который позволяет отслеживать ее текущее состояние, и, в зависимости от него выполнять программный код.

► В **Project Navigator** откройте файл **SceneDelegate.swift**.

Класс **SceneDelegate**, объявленный в файле **SceneDelegate.swift**, будет делегатом для каждой сцены, которая создается в процессе работы приложения.

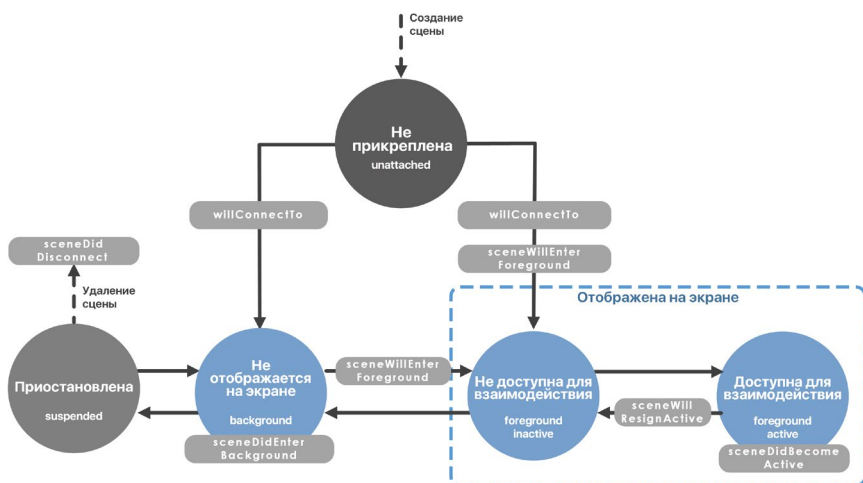


Рис. 5.8. Схема жизненного цикла сцены

В нем уже объявлены несколько методов. На рисунке 5.8 представлена схема жизненного цикла сцены с указанием всех доступных вам методов.

Разберем более подробно каждый из методов жизненного цикла.

Метод willConnectTo

Метод **willConnectTo** вызывается перед тем, как **UIKit** присоединяет новую сцену (экземпляр интерфейса) к приложению.

Одним из вариантов использования метода является определение того, какой View Controller должен быть загружен первым, так как не всегда работу приложения нужно начинать с одной и той же сцены. К примеру, вы разработали приложение, разграничивающее доступ для авторизованного и неавторизованного пользователя. Если пользователь не авторизован, ему необходимо показать экран ввода логина и пароля. В ином случае, он может быть сразу переброшен на главный экран приложения.

► Добавьте в метод **willConnectTo** вызов функции **print** (листинг 5.2).

ЛИСТИНГ 5.2

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options
connectionOptions: UIScene.ConnectionOptions) {
    guard let _ = (scene as? UIWindowScene) else { return }
    print("willConnectTo")
}
```

Метод **sceneWillEnterForeground**

Состояние **Foreground** говорит о том, что сцена (экземпляр интерфейса) находится на переднем плане, т.е. отображается на экране устройства. Метод **sceneWillEnterForeground** вызывается перед переходом сцены в состояние **Foreground**, т.е. прямо перед тем, как будет отображен графический интерфейс.

Состояние **Foreground** включает в себя два варианта:

- **Foreground Inactive** – сцена уже отображается на экране устройства, но все элементы «мертвы», т.е. недоступны пользователю (кнопки не нажимаются, текстовые поля не активируются и т.д.). Данное состояние используется системой для внесения финальных правок в интерфейс, а также в случае поступления особых «прерываний», вроде входящего звонка, поступления смс или активации голосового помощника Siri.
- **Foreground Active** – пользователь может взаимодействовать с элементами сцены.

Примеры использования метода:

- загрузка необходимых для работы приложения данных с диска или из сети.
- Добавьте в метод **sceneWillEnterForeground** вызов функции **print** (листинг 5.3).

ЛИСТИНГ 5.3

```
func sceneWillEnterForeground(_ scene: UIScene) {
    print("sceneWillEnterForeground")
}
```

Метод `sceneDidBecomeActive`

После того, как сцена отобразилась на экране и стала активной для взаимодействия, вызывается метод `sceneDidBecomeActive`. Вызов происходит уже после загрузки и отображения на экране всех графических элементов сцены.

Примеры использования метода:

- запуск таймеров для выполнения задач с определенным периодом;
 - отображение кнопки для снятия игры с паузы.
- Добавьте в метод `sceneDidBecomeActive` вызов функции `print` (листинг 5.4).

ЛИСТИНГ 5.4

```
func sceneDidBecomeActive(_ scene: UIScene) {  
    print("sceneDidBecomeActive")  
}
```

Метод `sceneWillResignActive`

Метод `sceneWillResignActive` вызывается перед тем, как сцена перейдет в состояние `Foreground Inactive` и перестанет отвечать на действия пользователя.

Примеры использования метода:

- остановка таймеров;
 - остановка фоновых задач, в выполнении которых нет необходимости;
 - постановка игры на паузу;
 - сохранение данных пользователя в файл или базу данных.
- Добавьте в метод `sceneWillResignActive` вызов функции `print` (листинг 5.5).

ЛИСТИНГ 5.5

```
func sceneWillResignActive(_ scene: UIScene) {  
    print("sceneWillResignActive")  
}
```

Метод `sceneDidEnterBackground`

Метод `sceneDidEnterBackground` вызывается сразу после перехода приложения в фоновый режим. Это происходит, например, при сворачивании приложения по нажатию кнопки `Home` (или свайпом вверх на моделях iPhone / iPad без кнопки).

Примеры использования метода:

- удаление элементов, занимающих большой объем памяти, которые могут безболезненно быть загружены в будущем;
- скрывание информации, к которой должен быть ограничен доступ (таких, как пароль или номер карты);
- закрытие подключения к общим системным базам данных, к которым больше не нужен доступ;

Состояние Background накладывает жесткие ограничения на возможности работы приложения. Тем не менее, у вас остаются следующие возможности:

- получение данных с удаленного сервера;
 - использование AirPlay;
 - использование Bluetooth;
 - использование функции «Картинка в картинке»;
 - прием push-уведомлений и др.
- Добавьте в метод **sceneDidEnterBackground** вызов функции **print** (листинг 5.6).

ЛИСТИНГ 5.6

```
func sceneDidEnterBackground(_ scene: UIScene) {  
    print("sceneDidEnterBackground")  
}
```

Метод **sceneDidDisconnect**

Метод **sceneDidDisconnect** вызывается после того, как сцена удаляется из приложения. Обычно это происходит, когда пользователь закрывает приложение в App Switcher.

Примеры использования метода:

- проведение финальной очистки, удаление временных файлов;
 - отключение от общих ресурсов;
 - сохранение пользовательских данных.
- Добавьте в метод **sceneDidDisconnect** вызов функции **print** (листинг 5.7).

ЛИСТИНГ 5.7

```
func sceneDidDisconnect(_ scene: UIScene) {  
    print("sceneDidDisconnect")  
}
```

Примечание Как было отмечено ранее, сама сцена (экземпляр графического интерфейса) представлена классом **UIWindowScene**, а он, в свою очередь, является дочерним для **UIScene**. Один из входных параметров каждого метода как раз имеет тип **UIScene**, но фактически передается значение типа **UIWindowScene** (это возможно благодаря наследованию в ООП).

Теперь посмотрим, когда именно каждый из методов вызывается. При выполнении следующих шагов после каждого действия смотрите на то, какие сообщения выведены на консоль. Так вы сможете определить, какой метод какого класса (**AppDelegate**, **SceneDelegate** или **ViewController**) в какой момент был вызван.

- Шаг 1. Запустите приложение и дождитесь появления интерфейса на экране.
- Шаг 2. Сверните приложение.
- Шаг 3. Откройте приложение.
- Шаг 4. Откройте App Switcher (сочетание клавиш **Command+Shift+Control+N** или пункт меню **Device > App Switcher**) и завершите приложение.

Итоговый состав сообщений с разделением по шагам приведен на рисунке 5.9.

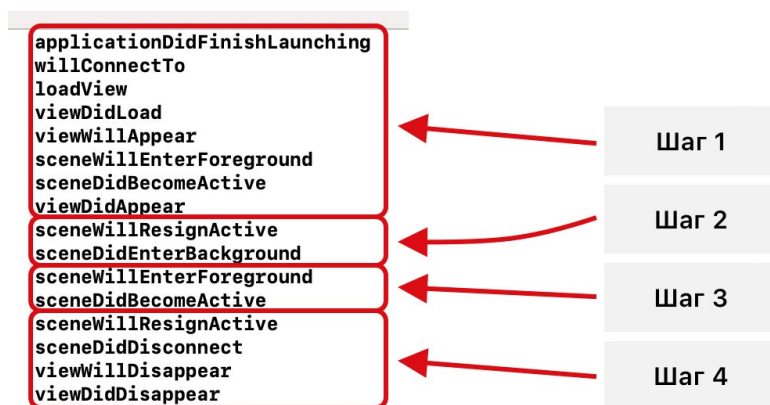


Рис. 5.9. Вывод на консоли

Примечание В зависимости от того, на каком устройстве вы запускаете приложение, на консоли может отсутствовать сообщение «`sceneDidDisconnect`». Но даже если вы его не видите, метод все равно вызывается. Это особенность работы с консолью.

Резюмируем. После того, как созданы значения **UIApplication** и **AppDelegate**, фреймворк **UIKit** создает новый экземпляр графического интерфейса (сцену) и экземпляр типа **SceneDelegate**, который является делегатом данной сцены. В процессе жизнедеятельности сцены (ее вывода на экран, скрытия и удаления) выполняются методы класса **SceneDelegate**.

Жизненный цикл приложения и его отдельных элементов на первый взгляд не так прост. Все это переплетение экземпляров и методов может вызвать не-

которую путаницу. Но в дальнейшем, в ходе изучения материала, в процессе создания собственных приложений, вы станете ориентироваться в доступных возможностях, как рыба в воде.

5.4 Класс UIWindow

Теперь мы готовы к тому, чтобы идти дальше и узнать, что же происходит после того, как созданы экземпляры классов **UIApplication**, **UIWindowScene** и их делегаты. В процессе подготовки графического интерфейса к отображению создается экземпляр класса **UIWindow**, который привязывается к конкретному экземпляру интерфейса (к **UIWindowScene**). Класс **UIWindow** описывает окно, или, другими словами, контейнер в который помещается все содержимое приложения, выведенное на экран. Экземпляр **UIWindow** не имеет внешнего вида — это просто контейнер. Если представить, что приложение (экземпляр **UIApplication**) — это компьютер, то экземпляр **UIWindow** — это его монитор. Сам по себе он не имеет графического интерфейса, а лишь позволяет отображаться содержимому.

Интерфейс выводится за счет того, что к **UIWindow** привязывается определенный View Controller, сцена которого и выводится на экран устройства. Основной задачей **UIWindow** на этом этапе является определение того, какой именно View Controller должен отобразить свою сцену. После этого автоматически создается экземпляр класса, связанного с View Controller, и выполняется дальнейшая работа по отображению интерфейса.

Стартовый View Controller

Поговорим о том, как **UIWindow** узнает, какой View Controller и какую сцену необходимо отобразить на экране устройства в первую очередь.

- ▶ Откройте файл **Main.storyboard**.
- ▶ В Document **Outline** выберите **View Controller**, имеющий в своем составе элемент **Storyboard Entry Point**.
- ▶ На панели **Inspectors** откройте **Attributes inspector**.

В разделе **View Controller** сразу под полем **Title** активирован пункт «**Is Initial View Controller**», к которому вы уже обращались ранее в ходе изучения жизненного цикла View Controller. Наличие галочки указывает на то, что текущий View Controller является начальным, или, другими словами, стартовым. Именно его сцена будет отображена первой сразу после запуска приложения. Данная настройка дублируется в **Interface Builder** и **Document Outline** (рис. 5.10).

Когда ваш проект будет содержать не одну, а множество сцен, с помощью данной настройки вы сможете указать, какая из них должна быть показана первой.

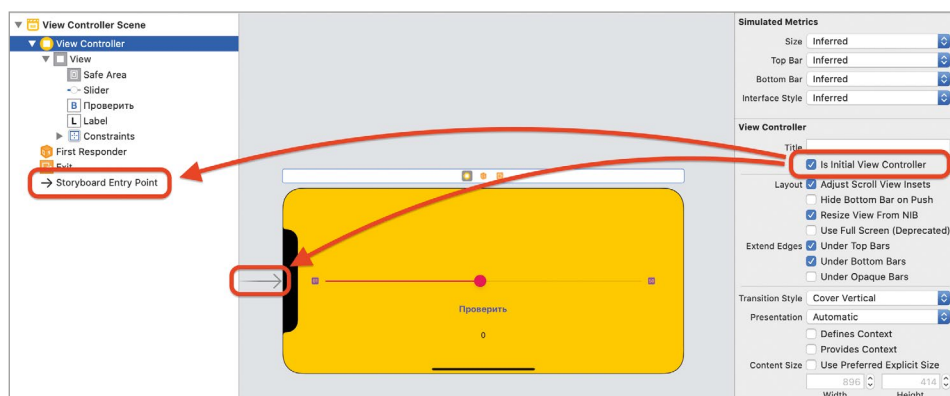


Рис. 5.10. Стартовый View Controller

Только один View Controller в пределах одного storyboard-файла может быть помечен как начальный. Если вы установите флаг «**Is Initial View Controller**» у одного элемента, он автоматически пропадет у того, у которого был ранее.

► Снимите галочку «**Is Initial View Controller**».

Теперь данный View Controller не является начальным. Об этом также говорят и отсутствующие в **Interface Builder** и в **Document Outline** указатели. Но что теперь будет показано при запуске приложения?

► Запустите сборку проекта на симуляторе.

Вместо интерфейса приложения симулятор отображает лишь черный экран, а в консоль была выведена информация об ошибке (рис. 5.11).

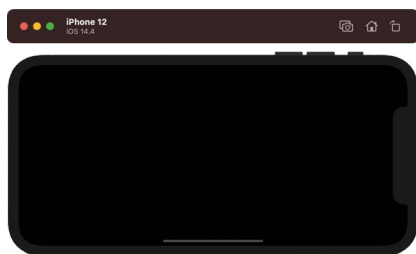


Рис. 5.11. Черный экран при отсутствии стартового вью контроллера

Если вы проанализируете сообщения, то вы увидите: приложение успешно загружается, экземпляр интерфейса создается и даже переходит в состояние **Foreground Active**, но вот View Controller не выполняет своих функций. Это связано с тем, что **UIWindow** не знает, какая сцена должна быть отображена первой, и по этой причине не загружает ничего.

Нам необходимо вернуть отображение графического интерфейса приложения. Сделать это можно двумя способами:

1. Для вью контроллера вновь активировать пункт «**Is Initial View Controller**».
2. С помощью кода указать стартовый вью контроллер.

Первый способ вы с легкостью сможете выполнить сами, а вот второй позволит нам лучше разобраться в устройстве приложения. Им мы сейчас и воспользуемся.

- ▶ Откройте файл **Main.storyboard**.
- ▶ Выделите **View Controller**.
- ▶ Откройте панель **Identity Inspector**.
- ▶ В поле **Storyboard ID** укажите «**MainViewController**».

Storyboard ID – это идентификатор сцены на сториборде, который должен быть уникальным в пределах одного storyboard-файла. С его помощью в дальнейшем мы будем производить загрузку вью контроллера в программном коде.

- ▶ Откройте файл **SceneDelegate.swift**.
- ▶ Дополните метод **willConnectTo** в соответствии с листингом 5.8.

ЛИСТИНГ 5.8

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options
connectionOptions: UIScene.ConnectionOptions) {
```

```
    print("sceneWillConnectTo")
```

```
    // ОБЯЗАТЕЛЬНО внесите изменения в следующую строку кода,
    // добавив имя параметра windowScene, в который извлекается значение
    guard let windowScene = (scene as? UIWindowScene) else { return }
```

```
    // Шаг 1
    window = UIWindow(frame: UIScreen.main.bounds)
    guard let window = window else {
        return
    }
```

```
    // Шаг 2
    window.windowScene = windowScene
```

```
    // Шаг 3
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    let viewController = storyboard.instantiateViewController(withIdentifier:
    er: "MainViewController") as! ViewController
```

```
// Шаг 4
window.rootViewController = viewController

// Шаг 5
window.makeKeyAndVisible()
}
```

Примечание Обратите внимание, что перед шагом 1 необходимо обязательно добавить имя параметра `windowScene`, в который извлекается значение.

Разберем тело метода по шагам.

Шаг 1. В первую очередь создается экземпляр типа **UIWindow**, описывающий окно, в котором в дальнейшем будет выводиться интерфейс.

Размеры **UIWindow** должны соответствовать размерам экрана устройства. Именно для этого используется класс **UIScreen**, описывающий дисплей устройства, на котором запущено приложение. Свойство **main.bounds** возвращает экземпляр типа **CGRect** (прямоугольник), размеры которого соответствуют размеру дисплея.

Таким образом, созданный экземпляр **UIWindow** имеет размеры, соответствующие размеру экрана.

Созданное окно (**UIWindow**) инициализируется в свойство **window** класса **SceneDelegate**. В этом свойстве обязательно должна содержаться ссылка на окно, используемое для отображения текущего экземпляра графического интерфейса. Установим его на следующем шаге.

Шаг 2. Текущий вариант интерфейса (**UIWindowScene**) связывается с созданным окном.

Шаг 3. Производится загрузка вью контроллера. Для этого сперва загружается сториборд, а потом с помощью идентификатора **MainViewController** и сам контроллер.

Шаг 4. Вью контроллер устанавливается в качестве корневого (стартового) для окна.

Примечание Именно данное выражение заменяет пункт «**Is Initial View Controller**» для вью контроллера.

Шаг 5. Окно устанавливается в качестве ключевого и видимого. Для этого используется метод **makeKeyAndVisible**.

Ключевое окно — это окно, которое принимает и обрабатывает события касания, т.е. события, возникающие из-за касаний пользователем экрана устройства. Ключевым может быть только одно окно в один момент времени.

► Запустите приложение на симуляторе.

Теперь вместо черного экрана вновь отобразился интерфейс приложения.

Примечание На консоли отобразится ошибка об отсутствии точки входа в приложение. Ее наличие никак не влияет на работоспособность приложения.

Ошибка формирования графического интерфейса

Примечание Для проверки того, что описано в данном подразделе, вы можете запустить приложение на iPad или его симуляторе в Xcode. Как именно использовать многооконный режим вы можете узнать в инструкции к нему.

Несмотря на то, что с первого взгляда интерфейс создается и отображается вполне корректно, в реализованном нами способе есть один большой недостаток. Если для приложения активировать многооконный режим и попытаться открыть два отдельных экземпляра графического интерфейса, то мы столкнемся с неожиданной проблемой: вторая сцена будет видна лишь наполовину (рис. 5.12).

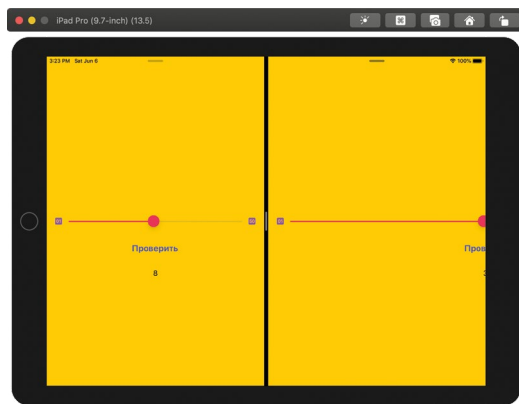


Рис. 5.12. Ошибка, при формировании интерфейса приложения в многооконном режиме

Примечание Для того, чтобы включить поддержку многооконного режима в приложении, в **Project Navigator** щелкните по файлу проекта. Далее в разделе **Target** выберите подраздел **Right on target** и активируйте пункт «**Supports multiple windows**».

Причина этому кроется в том, как именно мы указываем размеры **UIWindow** при создании его кодом в методе **willConnectTo**.

Так как левое окно было создано первым, использование класса **UIScreen** для определения размеров окна было вполне оправданным: его размеры соответствовали размерам экрана iPad.

Но такой способ определения размера второго окна (для второй сцены) совершенно не подходит. Уже в процессе создания окно по ширине должно быть

вдвое меньше дисплея. Но мы же с помощью `UIScreen.main.bounds` опять жестко задаем размеры равные размеру дисплея. И именно по этой причине нам видна лишь половина интерфейса, так как вторая половина прячется правее, за границей экрана.

Для того, что исправить ошибку, нам необходимо вместо размеров экрана устройства (`UIScreen`) обращаться к размеру сцены (`UIWindowScene`).

► Измените код шага 1 на тот, что приведен в листинге 5.9

ЛИСТИНГ 5.9

```
// Шаг 1
window = UIWindow(windowScene: windowScene)
guard let window = window else {
    return
}
```

► Удалите код шага 2.

Теперь размер каждой из сцен будет рассчитываться корректно (рис. 5.13).



Рис. 5.13. Корректный интерфейс в многооконном режиме

Примечание Материал этой главы описывает жизненный цикл приложений, функционирующих в iOS 13 и выше. Именно в этой версии операционной системы появился класс `SceneDelegate`. Если вам требуется обеспечить функционирование приложения в более ранних версиях iOS, вы можете самостоятельно изучить тему **App-based Lifecycle**, т.е. жизненный цикл на основе приложения. Мы же рассматривали **Scene-based Lifecycle**, т.е. жизненный цикл на основе сцены. Ничего кардинально отличающегося в этих двух способах нет, поэтому вы с легкостью освоите материал.

Хочу отметить, что в iOS 14 была добавлена возможность работать с жизненным циклом на основе структуры, унаследовавшей протокол `App`. Но это доступно только при создании интерфейса приложения на основе фреймворка `SwiftUI`.

ИТОГИ ПЕРВОЙ ЧАСТИ КНИГИ

Наиболее важная цель, которую я ставил перед собой в этой части заключается не в том, чтобы показать вам приемы работы со слайдером или дать возможность попрактиковаться с дженериками. Она состоит в том, чтобы дать вам базовые представления о том, как функционирует приложение и входящие в его состав компоненты. Понимание жизненного цикла, порядка загрузки и отображения элементов является крайне важным для любого iOS-разработчика. Эти знания используются практически в любом проекте. Думаю, теперь вы осознаете, что жизненный цикл не так страшен, как это казалось изначально.

Наверняка у вас остался вопрос: а что же делать со всем этим многообразием методов и компонентов? Ответ на него вы будете получать лишь с опытом, в том числе, при прочтении этой книги. Периодически вы будете сталкиваться с новыми задачами, решение которых будет основано именно на использовании методов жизненного цикла.

Часть II

ВВЕДЕНИЕ В ТАБЛИЧНЫЕ ПРЕДСТАВЛЕНИЯ

ПРОЕКТ «CONTACTS»

Эта часть книги будет посвящена разработке приложения «**Contacts**». Его интерфейс представлен на рисунке ниже.

Основным графическим элементом нового приложения станет табличное представление (Table View) – элемент, позволяющий отобразить произвольные данные в виде таблицы прямо на экране устройства. Это один из наиболее часто используемых элементов фреймворка **UIKit**. В основе данного элемента лежат два шаблона проектирования: «Делегирование» и «Источник данных», которые будут подробно рассмотрены в этой части в первую очередь.

Одной из важнейших функций «**Contacts**» будет сохранение созданных записей даже при закрытии приложения. Данная функция будет обеспечиваться с помощью элемента User Defaults – специального интерфейса, позволяющего организовать долговременное хранение произвольных данных.



Программный код, написанный в данной части книги, доступен по следующей ссылке:

<https://swiftme.ru/listings21>

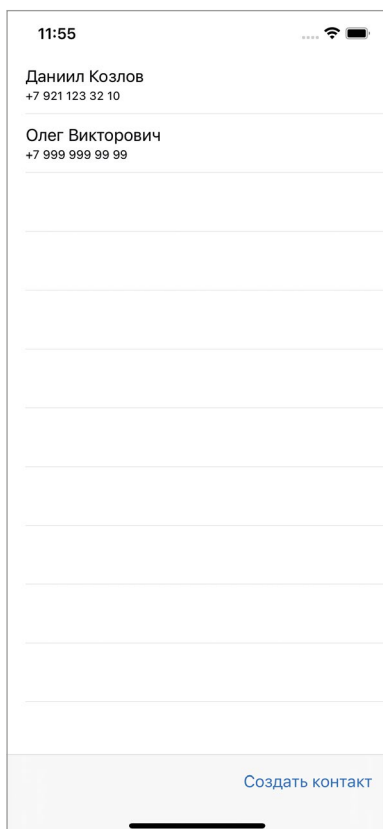


Рис. 1. Интерфейс приложения «Contacts»

Глава 6. Шаблоны «Делегирование» и «Источник данных»

Глава 7. Табличное представление. Класс UITableView

Глава 8. Долговременное хранение данных. User Defaults

Глава 6.

Шаблоны «Делегирование» и «Источник данных»

В этой главе вы:

- более подробно изучите уже знакомый вам шаблон «Делегирование»;
- рассмотрите новый для вас шаблон «Источник данных».

Прежде чем мы перейдем непосредственно к разработке приложения **«Contacts»**, нам необходимо познакомиться с шаблонами проектирования «Делегирование» и «Источник данных». Рассматривая каждый из данных шаблонов, мы будем писать небольшие блоки кода, которые позволят нам лучше усвоить учебный материал.

6.1 Шаблон «Делегирование»

Мы уже неоднократно говорили о паттерне «Делегирование» (Delegation), как в первой книге, так и в этой. Мы уделяем данному шаблону довольно много внимания, поскольку его глубокое понимание является одним из важнейших аспектов вашего уровня знаний как iOS-разработчика.

Что такое делегирование

Делегирование – это передача полномочий на выполнение какой-либо задачи. В одной из предыдущих глав книги был показан хороший пример использования шаблона «Делегирование» в реальной жизни. Рассмотрим его еще раз.

Предположим, вы запустили стартап по разработке мобильных приложений. Вы оформились как индивидуальный предприниматель и с успехом выполняете все задачи самостоятельно: от решения финансовых вопросов до написания программного кода. Время идет, ваша компания растет, но вместе с тем увеличиваются количество и объем решаемых задач.

В один момент вы понимаете, что уже не справляетесь самостоятельно и вам требуется помощь. Вы нанимаете нового сотрудника и передаете (делегируете) ему часть своих обязанностей, например, бухгалтерскую отчетность. Теперь при поступлении финансовых документов они незамедлительно передаются на исполнение новому сотруднику (делегату), который несет полную ответственность за решение данной задачи. Иначе говоря, вы делегировали ему свои полномочия для решения конкретной задачи.

На рисунке 6.1 представлена схема работы делегирования. Структура базового (делегирующего) элемента может быть построена таким образом, что он передает поступившую к нему задачу на выполнение другому элементу (делегату). При этом результат выполнения задачи может быть возвращен, а может и вовсе не возвращаться в базовый элемент: все зависит от конкретного случая.

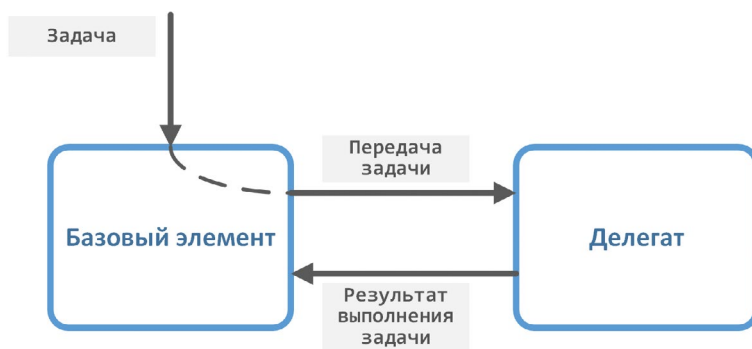


Рис. 6.1. Схема работы шаблона «Делегирование»

Делегирование в Swift

Применение шаблона «Делегирование» в Swift основано на использовании протоколов. Рассмотрим практический пример.

Предположим, вы разрабатываете приложение-мессенджер, которое позволяет пользователям обмениваться сообщениями. При этом необходимо, чтобы приложение собирало статистику того, какие типы сообщений отправляет и получает пользователь: текстовые, графические (картинки), аудиофайлы и т. д. И именно для реализации данной функциональности мы можем применить шаблон «Делегирование»: базовый тип будет отвечать только за прием-передачу сообщений, а все функции ведения статистики будут переданы типу-делегату.

Для написания кода проекта будем использовать playground.

► Создайте новый playground.

Начать реализацию проекта можно с сущности «Сообщение», которая станет базовой для всего приложения.

- Реализуйте протокол **MessageProtocol**, описывающий требования к сущности «Сообщение», и структуру **Message**, реализующую данный протокол (листинг 6.1).

ЛИСТИНГ 6.1

```
import Foundation

protocol MessageProtocol {
    // текст сообщения
    var text: String? { get set }
    // прикрепленное изображение
    var image: Data? { get set }
    // прикрепленный аудиофайл
    var audio: Data? { get set }
    // прикрепленный видеофайл
    var video: Data? { get set }
    // дата отправки
    var sendDate: Date { get set }
    // отправитель
    var senderID: UInt { get set }
}

struct Message: MessageProtocol {
    var text: String?
    var image: Data?
    var audio: Data?
    var video: Data?
    var sendDate: Date
    var senderID: UInt
}
```

Свойства **text**, **image**, **audio** и **video** описывают соответствующие элементы сообщения. Каждое из них представлено в виде опционала, так как данные могут отсутствовать. Например, вы можете отправить картинку с текстом, в этом случае в **audio** и **video** не будет значений (**nil**).

Для параметров **image**, **audio** и **video** используется тип **Data**, который позволяет представить некоторые данные в виде простой байтовой последовательности. Тип **Data** по сути может описывать все, что угодно.

Примечание В дальнейшем вы обязательно начнете работать с типом данных **Data**, но на данном этапе эти навыки не так важны.

Для работы с датой и временем в Swift используется тип данных **Date** (не путать с **Data**). Именно он является типом свойства **sendDate**, содержащим дату и время отправления сообщения.

Примечание Для доступа к типам **Data** и **Date** необходимо подключить библиотеку **Foundation**, что и реализовано в первой строке листинга.

Свойство **senderID** содержит уникальный идентификатор отправителя.

Теперь перейдем к реализации сущностей «Мессенджер» и «Менеджер статистики».

- Реализуйте протоколы **MessengerProtocol** и **StatisticDelegate**, которые определяют требования к набору свойств и методов «Мессенджера» и «Менеджера статистики» (листинг 6.2).

ЛИСТИНГ 6.2

```
protocol StatisticDelegate {
    func handle(message: MessageProtocol)
}

protocol MessengerProtocol {
    // массив всех сообщений
    var messages: [MessageProtocol] { get set }
    // делегат для ведения статистики
    var statisticDelegate: StatisticDelegate? { get set }
    // инициализатор
    init()
    // принять сообщение
    mutating func receive(message: MessageProtocol)
    // отправить сообщение
    mutating func send(message: MessageProtocol)
}
```

Примечание Обратите особое внимание на то, что все зависимости между сущностями основаны на использовании протоколов, т. е. в **MessengerProtocol** и **StatisticDelegate** нигде не используется прямая ссылка на тип **Message**. Вместо этого используется протокол **MessageProtocol**.

Помните, что протоколы должны ссылаться на протоколы, а не на конкретные типы данных. Об этом мы еще поговорим в дальнейшем.

Ссылка на делегат находится в свойстве **statisticDelegate** протокола **MessengerProtocol**, но в данном случае делегат еще не используется, так как нет конкретной реализации типов, подписанных на этот протокол.

Свойство **messages** будет хранить упорядоченный массив переданных и полученных сообщений.

Методы **receive** и **send** помечены модификатором **mutating**, так как предполагается, что они должны добавлять сообщения в свойство **messages**.

Реализуем типы данных **Messenger** и **StatisticDelegate**.

- Добавьте в playground код из листинга 6.3.

ЛИСТИНГ 6.3

```
struct StatisticManager: StatisticDelegate {
    func handle(message: MessageProtocol) {
        // ...
        // обработка сообщения
        // ...
        print("обработка сообщения от User # \(message.senderID) заверше-
на")
    }
}

struct Messenger: MessengerProtocol {
    var messages: [MessageProtocol]
    var statisticDelegate: StatisticDelegate?

    init() {
        messages = []
    }

    mutating func receive(message: MessageProtocol) {
        statisticDelegate?.handle(message: message)
        messages.append(message)
        // ...
        // прием сообщения
        // ...
    }

    mutating func send(message: MessageProtocol) {
        statisticDelegate?.handle(message: message)
        messages.append(message)
        // ...
        // отправка сообщения
        // ...
    }
}
```

Теперь при каждом вызове методов **send** и **receive** данные о сообщении (значения типа **MessageProtocol**) будут переданы в менеджер статистики для обработки. Но это произойдет только в том случае, если делегат будет инициализирован свойству **statisticDelegate** (листинг 6.4). При этом вызов метода делегата остается незаметным для пользователя класса **Messenger**. То есть, при использовании **Messenger** для отправки и получения сообщения вы напрямую не вызываете метод **handle**.

► Реализуйте код из листинга 6.4.

ЛИСТИНГ 6.4

```
var messenger = Messenger()
messenger.statisticDelegate = StatisticManager()
messenger.send(message: Message(text: "Привет!", sendDate: Date(), senderID: 1))
```

Консоль

обработка сообщения от User #1 завершена

Как вы можете видеть, мы просто произвели отправку текста, а на консоли появилось сообщение об успешной обработке.

В некоторых случаях, а вы будете достаточно часто встречать такие подходы при разработке под iOS, в качестве делегата может назначаться сам тип данных. В нашем примере для этого достаточно подписать структуру **Messenger** на протокол **StatisticDelegate** (например, с помощью расширения) и в качестве значения свойства **statisticDelegate** передать себя.

Примечание Среди разработчиков довольно популярным является подход, когда объектный тип подписывают на протокол через создание расширения. Это позволяет сделать код более чистым и структурированным, а также с легкостью удалить реализацию определенного протокола в случае необходимости.

► Реализуйте код из листинга 6.5.

ЛИСТИНГ 6.5

```
extension Messenger: StatisticDelegate {
    func handle(message: MessageProtocol) {
        // ...
        // обработка сообщения
        // ...
        print("обработка сообщения от User # \(message.senderID) завершена")
    }
}

var messenger = Messenger()
messenger.statisticDelegate = messenger.self
```

Теперь при отправке и приеме сообщения все так же будет вызываться метод **handle**, только теперь он объявлен непосредственно в структуре **Messenger**.

Работа с памятью

Я бы хотел обратить ваше внимание на один очень важный момент: если тип сам является своим делегатом (как было показано в листинге выше) и при этом реа-

лизован как структура, то при передаче значения свойству **statisticDelegate** создается копия структуры! То есть, значение, хранящееся в параметре **messenger** и значение, хранящееся в свойстве **statisticDelegate** – это две копии, два различных значения. Почему так? Потому что структуры – это value type.

Проверить это утверждение достаточно просто: отправим сообщение и сравним количество сообщений в свойстве **messages**.

► Реализуйте код из листинга 6.6.

ЛИСТИНГ 6.6

```
messenger.send(message: Message(text: "Привет!", sendDate: Date(), senderID: 1))
messenger.messages.count // 1
(messenger.statisticDelegate as! Messenger).messages.count // 0
```

Как вы можете видеть, в первом случае **messages** хранит 1 элемент, а во втором – 0. Очень важно, чтобы вы понимали то, о чем я только что рассказал, так как у вас может возникнуть желание получить доступ к значению свойства **messages** внутри делегата, а это приведет к ошибочным результатам. Вы должны быть очень внимательны к структурам при работе с делегатами.

Но есть способ избежать описанных проблем – реализовать сущность «Мессенджер» с помощью класса. В это случае в свойстве **statisticDelegate** будет храниться ссылка на текущий экземпляр, а не его копия (классы – это reference type). Но для этого потребуется внести некоторые правки в уже написанный код:

1. Для того, чтобы избежать утечек памяти, ссылка в свойстве **statisticDelegate** должна быть слабой, т.е. ее необходимо пометить с помощью ключевого слова **weak**.

Примечание Если вы не понимаете, почему в данном случае может произойти утечка памяти, советую вам перечитать главу «Управление памятью» в первой книге. Но если коротко: без **weak** класс будет содержать сильную (**strong**) ссылку сам на себя, и он не может быть уничтожен до завершения работы приложения.

2. Протокол **StatisticDelegate** необходимо пометить с помощью ключевого слова **class** (или **AnyObject**). В ином случае предыдущий пункт не сможет быть выполнен (слабой ссылка может быть только на reference type – класс).
3. В реализации класса **Messenger** не должны использоваться ключевые слова **mutating**. Они предназначены только для структур.

В листинге 6.7 показан вариант с использованием класса. При этом проверка количества сообщений и в самом классе, и в делегате выдает одно и то же значение.

ЛИСТИНГ 6.7

```
protocol StatisticDelegate: AnyObject {
    func handle(message: MessageProtocol)
}
```

```
protocol MessengerProtocol {
    // массив всех сообщений
    var messages: [MessageProtocol] { get set }
    // делегат для ведения статистики
    var statisticDelegate: StatisticDelegate? { get set }
    // инициализатор
    init()
    // принять сообщение
    mutating func receive(message: MessageProtocol)
    // отправить сообщение
    mutating func send(message: MessageProtocol)
}

class Messenger: MessengerProtocol {
    var messages: [MessageProtocol]
    weak var statisticDelegate: StatisticDelegate?

    required init() {
        messages = []
    }

    func receive(message: MessageProtocol) {
        statisticDelegate?.handle(message: message)
        messages.append(message)
        // ...
        // прием сообщения
        // ...
    }

    func send(message: MessageProtocol) {
        statisticDelegate?.handle(message: message)
        messages.append(message)
        // ...
        // отправка сообщения
        // ...
    }
}

extension Messenger: StatisticDelegate {
    func handle(message: MessageProtocol) {
        // ...
        // обработка сообщения
        // ...
        print("обработка сообщения от User # \(message.senderID) завершена")
    }
}
```

```
}  
  
var messenger = Messenger()  
messenger.statisticDelegate = messenger.self  
  
messenger.send(message: Message(text: "Привет!", sendDate: Date(),  
senderID: 1))  
messenger.messages.count // 1  
(messenger.statisticDelegate as! Messenger).messages.count // 1
```

Делегаты очень активно применяются в iOS-разработке. Примерами их использования могут быть как уже упомянутые ранее табличные представления, так и другие элементы, доступные при разработке приложений. Так, текстовые поля используют делегаты в том числе для того, чтобы обработать вводимый пользователем текст (например, разрешить ввод только чисел) или определить порядок действий после нажатия кнопки «Готово». С другим примером делегата, классом **AppDelegate**, вы уже встречались в предыдущих главах.

6.2 Шаблон «Источник данных»

Что такое «Источник данных»

Шаблон «Источник данных» (Data Source) – это частный случай шаблона «Делегирование». Если в случае классического делегирования вы передаете делегату данные для обработки, то в «» базовый элемент наоборот запрашивает данные для самостоятельного решения задачи. На рисунке 6.2 показана схема работы шаблона «Источник данных».

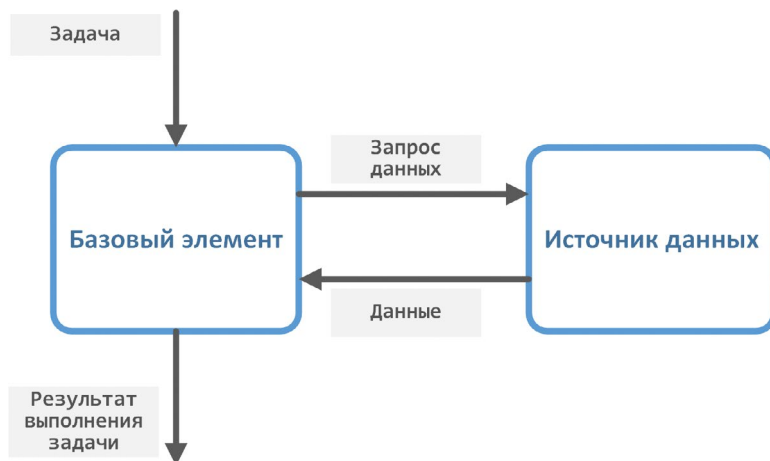


Рис. 6.2. Схема работы шаблона «Источник данных»

Для лучшего понимания рассмотрим пример на основе все той же компании, которую вы уже создали и успешно развиваете.

Делегирование: Вам поступает две внушительные по своим размерам пачки счетов от налоговой, вы просто передаете их своему бухгалтеру (делегату) и даете указание разобраться.

Источник данных: Вы обращаетесь к своему бухгалтеру с указанием выдать список работников с заработной платой более 300 тысяч рублей.

То есть, в первом случае вы передаете данные делегату, а во втором – делегат (источник) передает данные вам.

Как я и говорил, «Источник данных» основан на «Делегировании» – это два очень близких, но все же отличающихся шаблона.

Источник данных в Swift

В качестве примера возьмем разработанный ранее набор протоколов и типов, обеспечивающих работу мессенджера. Где в данном случае может быть использован шаблон «Источник данных»? Например, для получения списка сообщений, их загрузки из базы данных.

- В playground реализуйте новый протокол **MessengerDataSourceProtocol**, описывающий источник данных, и добавьте свойство **dataSource** в протокол **MessengerProtocol** (листинг 6.8).

ЛИСТИНГ 6.8

```
protocol MessengerDataSourceProtocol: class {
    func getMessages() -> [MessageProtocol]
}

protocol MessengerProtocol {
    // ...
    // делегат для загрузки сообщения
    var dataSource: MessengerDataSourceProtocol? { get set }
    // ...
}
```

- Реализуйте свойство **dataSource** в классе **Messenger** (листинг 6.9).

ЛИСТИНГ 6.9

```
class Messenger: MessengerProtocol {
    // ...
    weak var dataSource: MessengerDataSourceProtocol? {
        didSet {
            if let source = dataSource {
                messages = source.getMessages()
            }
        }
    }
}
```

```
        }  
    }  
}  
// ...  
}
```

Свойство **dataSource** включает в себя наблюдатель. Если потребуется изменить значение данного свойства, то незамедлительно будет произведена загрузка всех сообщений.

Теперь нам необходимо реализовать непосредственно сам источник данных.

► Создайте расширение для класса **Messenger** (листинг 6.10).

ЛИСТИНГ 6.10

```
extension Messenger: MessengerDataSourceProtocol {  
    func getMessages() -> [Message] {  
        return [Message(text: "Как дела?", sendDate: Date(), senderID: 2)]  
    }  
}
```

Метод **getMessages** возвращает массив, состоящий из одного сообщения. В реальном проекте вы бы могли реализовать, например, загрузку сообщений с сервера или из базы данных.

Теперь при создании экземпляра класса **Messenger** и инициализации источника данных в свойстве **messages** уже будет находиться одно сообщение.

► Реализуйте код из листинга 6.11.

ЛИСТИНГ 6.11

```
var messenger = Messenger()  
messenger.dataSource = messenger.self  
messenger.messages.count // 1
```

На этом мы завершаем рассмотрение шаблонов «Делегирование» и «Источник данных». Тем не менее вы еще неоднократно столкнетесь с этими понятиями в процессе усвоения учебного материала, в частности, при изучении табличных представлений (Table View).

Глава 7.

Табличные представления.

Класс UITableView.

В этой главе вы:

- узнаете, что такое табличные представления и как они работают;
- узнаете, что такое переиспользуемые ячейки;
- научитесь использовать табличные представления для отображения данных в табличном виде.

Таблица – это один из наиболее простых и доступных способов представления информации не только в iOS, но и в реальной жизни. Данные, помещенные в таблицу, выглядят структурированными и очень удобными для восприятия. Операционная система iOS позволяет использовать таблицы при конструировании интерфейса приложения. Для этого необходимо использовать специальный графический элемент – табличное представление.

Табличное представление (Table View) – это один из важнейших элементов, с которым вам предстоит работать в процессе обучения и дальнейшей карьеры iOS-разработчика. С его помощью можно создавать очень удобные интерфейсы. На рисунке 7.1 показаны знакомые многим приложения, интерфейс которых как раз и построен на основе таблиц. Данные в них расположены в виде объединенных в столбцы строк, каждая из которых отображает необходимую информацию.

Табличные представления представлены специальным классом из состава фреймворка **UIKit** – **UITableView**.

7.1 Введение в табличные представления

Табличные представления обладают следующими особенностями:

1. В ширину всегда имеют одну колонку: не больше и не меньше.
2. В высоту могут иметь произвольное количество строк.

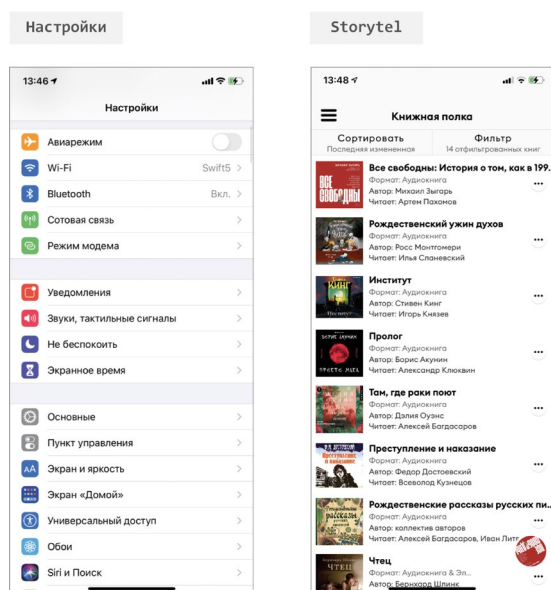


Рис. 7.1. Приложения, построенные на основе Table View

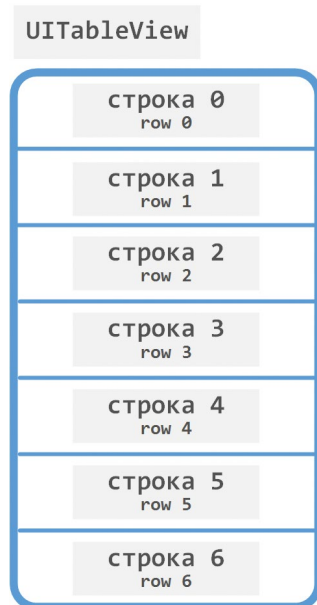


Рис. 7.2. Строки в составе табличного представления

Возможно, вы удивились, прочитав про ограничение в одну колонку (я до сих пор помню свое удивление, узнав об этом). Но именно такой подход позволяет делать по-настоящему удобный для пользователя графический интерфейс, который прокручивается не в двух осях (по вертикали и по горизонтали), а только по одной (по вертикали).

Прежде чем мы приступим к практическому использованию таблиц, рассмотрим несколько базовых понятий.

Строки (row)

Таблица может иметь произвольное количество строк. Верхняя строка всегда имеет индекс 0, а каждая последующая – увеличенный на 1, точно, как в массивах (рис. 7.2).

Количество строк в таблице ограничено исключительно вашими потребностями: их может быть хоть 1, хоть 100 000 (однако делать настолько большие таблицы – не лучшее решение). Табличное представление по умолчанию поддерживает скроллы. Таким образом, если количество строк в них превышает размеры одного экрана, пользователь сможет перемещаться по таблице вверх и вниз (скроллить ее), и от разработчика для реализации этого не потребуются никаких дополнительных действий.

Ячейки (cell)

Сама по себе строка не определяет ничего, кроме своей позиции в таблице. Она не определяет, в том числе, и внешний вид данных, которые будут содержаться в ней. Для графического оформления таблицы используются ячейки.

Ячейка (cell) определяет внешний вид данных, выводимых в строке. Она представлена классом **UITableViewCell**.

Вероятно, на данном этапе такой подход покажется вам немного странным и нелогичным: зачем разделять два этих понятия? Почему строка не могла принять на себя функции ячейки и отвечать за внешний вид данных, а не только за их порядок в таблице? Ответ на это кроется в том, каким образом табличные представления функционируют «под капотом».

Как формируется табличное представление

Рассмотрим следующий пример.

Предположим, вы разработали приложение «Контакты», которое отображает список введенных контактов. Так как контакты – это множество однотипных записей, для реализации графического интерфейса приложения наилучшим решением станет использование табличного представления. Таким образом, вы сможете расположить записи в алфавитном порядке одну под другой. Макет интерфейса приложения представлен на рисунке 7.3.

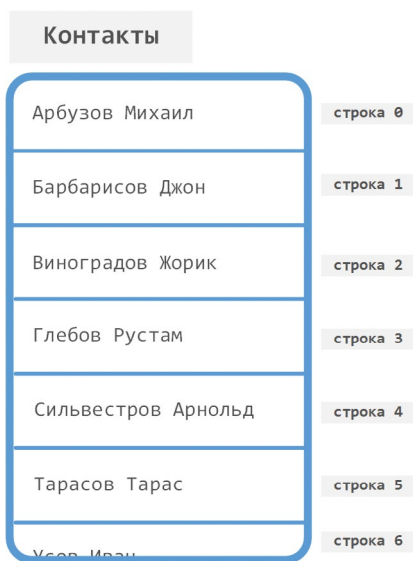


Рис. 7.3. Макет приложения Контакты

После того, как пользователь нажимает на иконку приложения, оно загружается, и на экране устройства появляется список контактов. Но как при этом происходит загрузка и формирование внешнего вида табличного представления?

Шаг 1. В первую очередь, табличное представление определяет общее количество строк, которые будут в нем содержаться. Причем это не количество строк, которое помещается на один экран, а общее количество строк, которые потенциально могут быть отображены при пролистывании таблицы вниз. Предположим, что всего их десять.

На рисунке 7.3 вы можете видеть, что один экран вмещает 7 строк таблицы (шесть – полностью и одну – частично). Общее же количество строк несколько больше, просто некоторые из них пока еще не отображаются (они станут видны при скролле).

Примечание Количество строк, которые могут быть отображены на одном экране, всегда зависит от конкретного устройства, на котором запускается приложение, а точнее от размеров его дисплея.

Шаг 2. Далее формируются ячейки и помещаются в соответствующие строки. Самое важное, что вам стоит запомнить – ячейка для конкретной строки формируется за мгновение до того, как она будет показана на экране. Таким образом, сразу после загрузки сцены мы увидим строки с индексами 0 . . . 6, а значит будут созданы ячейки только для этих строк, и только они. Каждая последующая ячейка будет сформирована за мгновение до того, как она будет отображена на экране.

Внешний вид конкретной строки, то есть ячейка для этой строки, формируется лишь непосредственно перед ее отображением на экране!

Благодаря такому подходу экономится колоссальное количество ресурсов. Только представьте, что было бы, если приложение сразу формировало все ячейки для всех строк таблицы. Что, если в вашей таблице 100 000 строк? Сколько времени при этом могла бы занимать ее загрузка? А что, если пользователь найдет нужный ему контакт среди первых семи строк? Оставшиеся 99 993 будут бесцельно занимать оперативную память, а на их загрузку впустую было бы потрачено процессорное время.

Именно по этой причине табличное представление обеспечивает загрузку внешнего вида только тех строк, которые в данный момент отображены на экране. И именно по этой же причине строки и ячейки – это две разные сущности, которые выполняют разные задачи. Как только очередная строка должна отобразиться на экране, табличное представление формирует ячейку, которая должна быть выведена в данной строке.

Операционная система iOS ставит своим приоритетом высокую отзывчивость и плавную работу графического интерфейса приложений. Вы еще не раз в этом убедитесь, в частности, в процессе изучения многопоточного программирования.

Вот мы и видим загруженный интерфейс приложения «Контакты», содержащий 7 строк данных с индексами от 0 до 6 (рис. 7.4). Для каждой строки в памяти хранится своя ячейка (экземпляр класса **UITableViewCell**).

Примечание Напомню, что общее количество строк, которые потенциально могут быть отображены, соответствует количеству записей в телефонной книге – 10.

Начнем прокрутку таблицы (рис. 7.5). Верхняя строка с индексом 0 постепенно скрывается с экрана, а строка с индексом 6 теперь полностью видна. Количество элементов в памяти и на экране при этом не изменяется.

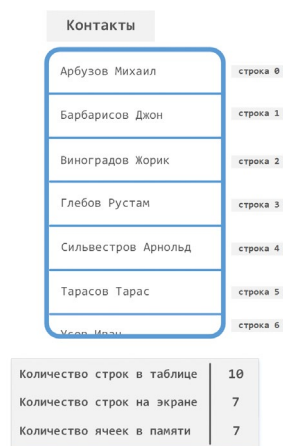


Рис. 7.4. Прокрутка UITableView

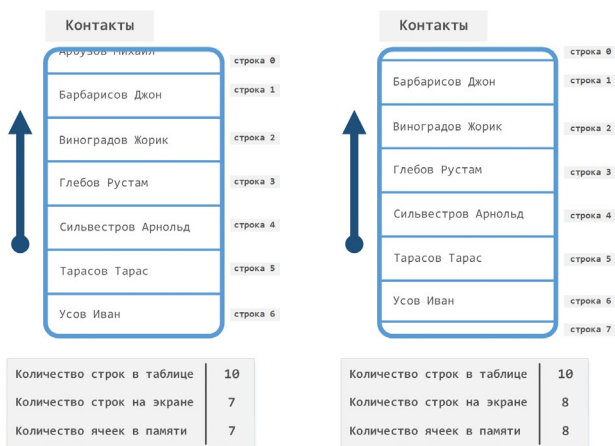


Рис. 7.5. Прокрутка UITableView



Рис. 7.6. Прокрутка UITableView

Переиспользуемые ячейки

Но и это еще не все. Как вы думаете, что происходит с ячейкой после того, как она скрывается с экрана?

Использование табличных представлений использует концепцию «переиспользуемых» ячеек, то есть таких ячеек, которые могут быть многократно использованы для отображения данных. Продолжим рассмотрение того, как это работает, на примере с приложением «Контакты».

Если продолжить скроллить таблицу (рис. 7.7), ячейка строки с индексом 0 полностью скроется с экрана. Но вместо того, чтобы уничтожиться, она будет размещена в специальном хранилище, откуда сможет быть извлечена при отображении новой ячейки!

Несмотря на то, что на экране отображаются 7 строк, количество ячеек в памяти 8 (с учетом переиспользуемой ячейки в хранилище).

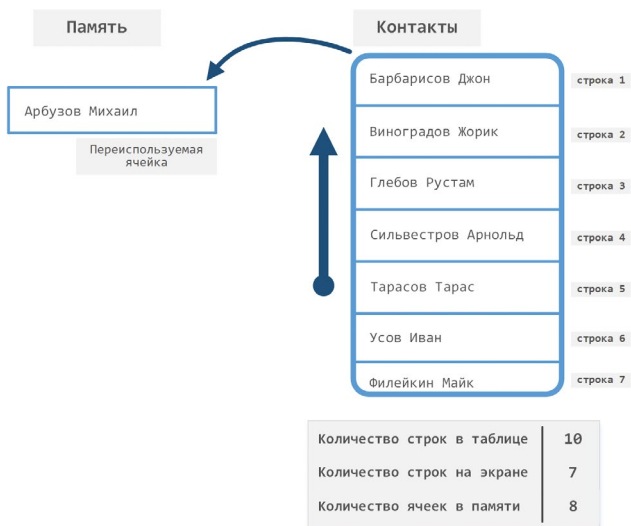


Рис. 7.7. Прокрутка UITableView

Продолжим прокрутку (рис. 7.8). Прямо перед появлением строки с индексом 8 табличное представление должно создать соответствующую ей ячейку. Но вместо того, чтобы сгенерировать новый экземпляр класса **UITableViewCell**, происходит проверка того, а нет ли в хранилище переиспользуемой ячейки. И так как в нашем случае такая ячейка существует, то возвращается именно она.

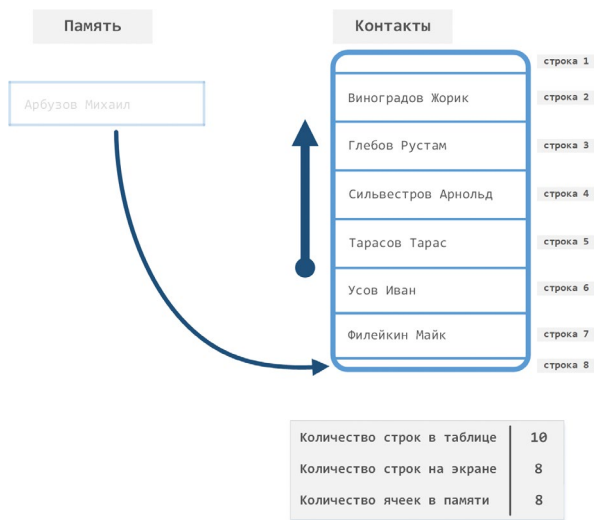


Рис. 7.8. Прокрутка UITableView

Наверняка у вас возникло подозрение, что раз мы повторно используем ячейку, в строке 8 может появиться текст «Арбузов Михаил». Но это не так (рис. 7.9). После того, как переиспользуемая ячейка была получена из памяти, она необходимым образом модифицируется, чтобы отобразить требуемую информацию. В данном случае изменяется текст, и вместо «Арбузов Михаил» мы видим «Храпов Иван».



Рис. 7.9. Прокрутка UITableView

Концепция переиспользуемых ячеек позволяет экономить ресурсы при загрузке каждой очередной ячейки: так как программе не нужно тратить время на создание нового объекта типа **UITableViewCell**, она просто загружает уже существующий.

Примечание Конечно, вы можете и не использовать переиспользуемые ячейки в своих программах: вы можете просто создавать новый экземпляр ячейки при каждом запросе табличного представления. Но это потенциально может привести к перерасходу оперативной памяти и появлению фризов при загрузке интерфейса.

Старайтесь использовать эту возможность!

В следующем разделе мы перейдем к практической части учебного материала и попробуем создать таблицу.

7.2 Использование табличного представления

Размещение элемента Table View

Начнем создание проекта «**Contacts**».

- Откройте Xcode и создайте новый проект (рис. 7.10).

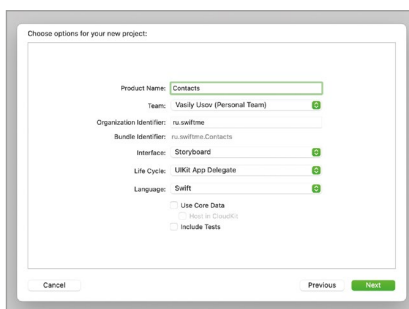


Рис. 7.10. Создание нового проекта

- В составе проекта откройте файл **Main.storyboard**.

Сейчас сториборд содержит одну пустую сцену. Разместим на ней табличное представление.

- Откройте библиотеку объектов, найдите объект Table View и переместите его на сцену (рис. 7.11).

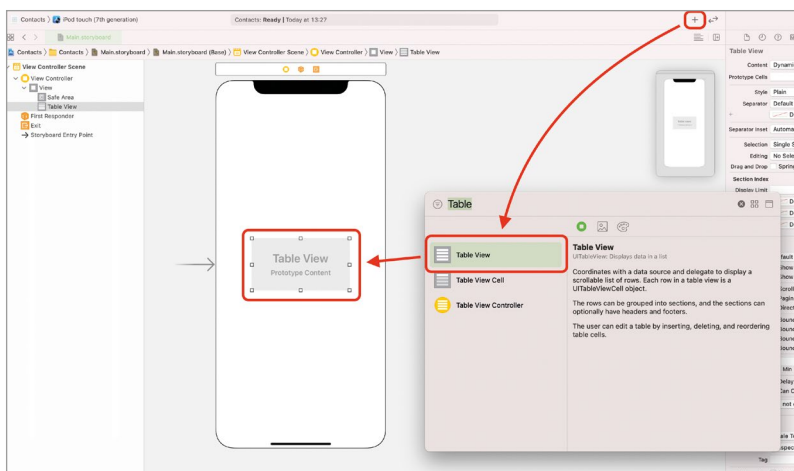


Рис. 7.11. Размещение табличного представления на сцене

Примечание В библиотеке объектов помимо **Table View** вы также увидите **Table View Cell** и **Table View Controller**. Не перепутайте необходимый элемент – нам нужен именно **Table View**.

Table View – это и есть, то самое табличное представление, в котором будут выводиться ячейки с данными о контактах. Сейчас Table View занимает лишь небольшое пространство на сцене. Нам же необходимо увеличить его на все доступное пространство. Решить эту задачу можно двумя путями:

1. Растянуть таблицу, выделив ее и потянув за углы (рис. 7.12).

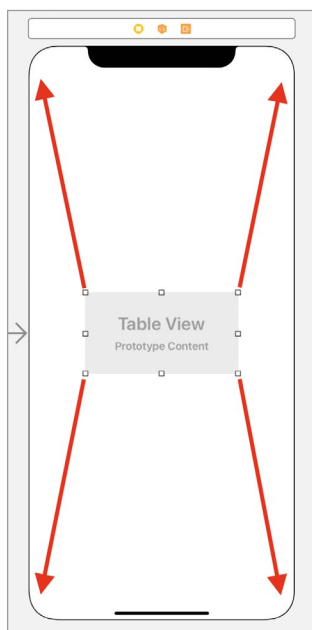



Рис. 7.12. Растягивание таблицы

Ранее мы уже подробно говорили о таком способе позиционирования элементов, тем не менее, я хочу в очередной раз заострить на нем ваше внимание. Если вы сделаете так, как описано выше, то интерфейс приложения будет выглядеть корректно только в случае, когда устройство (или симулятор), на котором будет запускаться приложение, будет соответствовать устройству, выбранному в **Interface Builder**. Очевидно, это не лучший способ решения задачи позиционирования элементов.

2. Использовать ограничения (constraints).

Мы можем определить правила, по которым отступы от внешних границ таблицы до краев экрана на любых устройствах будут равны нулю. И этот вариант будет наиболее верным, так как позволяет интерфейсу корректно отображаться на различных устройствах.

- ▶ Выделите элемент **Table View** на сцене.
- ▶ Нажмите на кнопку **Add New Constraints** , расположенную в нижней части **Interface Builder**.
- ▶ В верхней части всплывающего окна щелкните по четырем прерывистым красным линиям так, чтобы они стали закрашенными.
- ▶ Во всех текстовых полях, соответствующих линиям, установите значение 0 (рис 7.13).

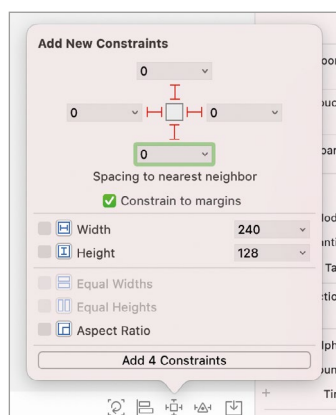


Рис. 7.13. Создание ограничений

Примечание Каждая из линий определяет отдельное правило для отступа данного элемента от ближайшего с одной из четырех сторон. В нашем случае на сцене расположен всего один элемент, и ближайшим для него является корневое представление сцены, значит отступы задаются между данным элементом (табличным представлением) и корневым представлением.

- ▶ Нажмите кнопку **Add 4 Constraints**, расположенную в нижней части всплывающего окна.

После создания ограничений **Table View** автоматически растянется на весь экран и будет выглядеть так на любом устройстве.

- ▶ Запустите проект на симуляторе.

Сейчас таблица выводит просто множество пустых строк. Это связано с тем, что на данный момент приложение и **Table View**, в частности, не работают с какими-либо данными.

- ▶ Попробуйте промотать таблицу вверх и вниз.

Таблица «из коробки» поддерживает обработку скроллов. При попытке прокрутки она двигается, но из-за отсутствия данных незамедлительно возвращается в исходное положение.

Откуда Table View получает данные

Теперь перейдем к наполнению таблицы данными. Но прежде, чем переходить к практической части реализации, поговорим о том, как именно в Swift решается данный вопрос.

Элемент Table View сам по себе ничего не знает о данных, которые он будет выводить. Для его наполнения используется шаблон проектирования Источник данных (Data Source), который мы рассматривали ранее в главе. Проще говоря, для того, чтобы в Table View появились данные, ему необходимо назначить источник данных, которому будут делегированы вопросы наполнения таблицы данными. Делается это либо средствами **Interface Builder**, либо с помощью программного кода. В процессе загрузки табличное представление отправляет своему источнику данных запросы о том, какую информацию и в каком виде выводить, примерно так, как показано на рисунке 7.14.

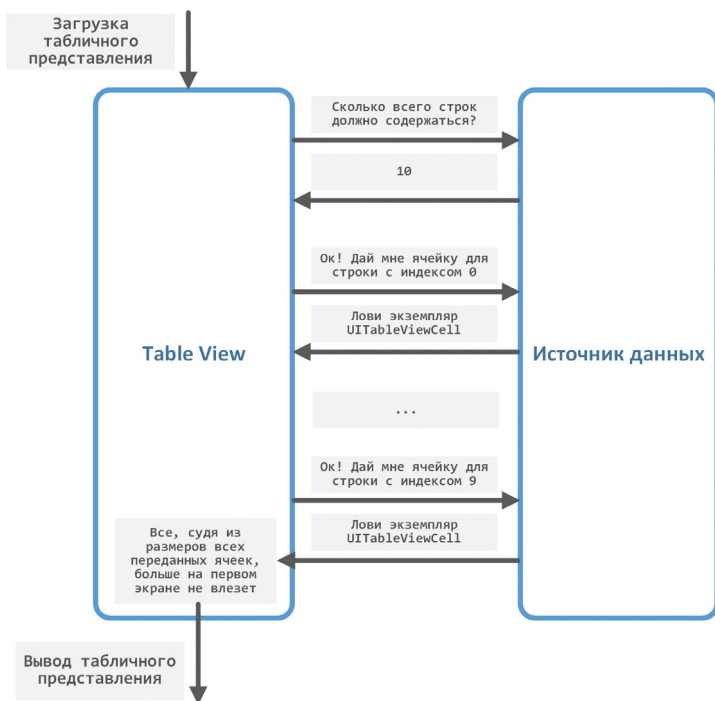


Рис. 7.14. Таблица и источник данных

Общение между таблицей и источником данных происходит с помощью вызова специальных методов (точно так же, как мы видели это в классе **AppDelegate**). В качестве источника таблицы обычно назначается View Controller, обеспечивающий работу сцены, на которой расположена таблица.

- ▶ На сцене выделите элемент **Table View**.
- ▶ Откройте панель **Connections Inspector**.
- ▶ Создайте связь между элементом **dataSource**, расположенным в разделе **Outlets**, и View Controller на сцене. Для этого перетяните соответствующий серый кружок на желтый кружок на сцене (рис. 7.15).

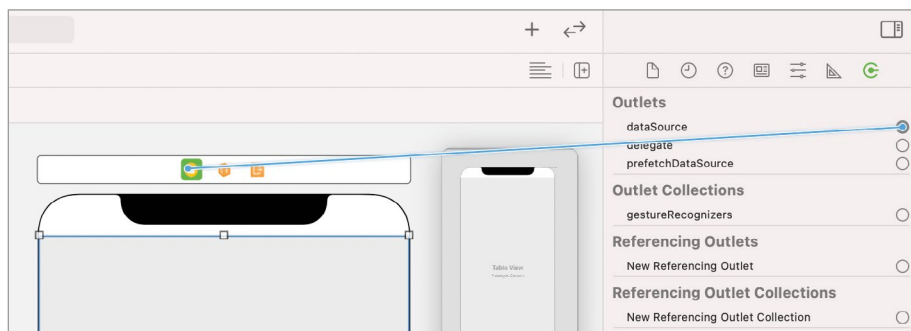


Рис. 7.15. Создание связи между таблицей и источником данных

Теперь класс View Controller назначен в качестве источника данных для табличного представления.

Отладка приложения

- ▶ Произведите запуск приложения на симуляторе.

Несмотря на то, что компиляция и сборка приложения завершается успешно, в ходе его загрузки возникает исключительная ситуация или, другими словами, приложение «падает»: в Xcode появится красная строка с текстом ошибки, на консоли будет представлен подробный лог, а на симуляторе отобразится белый экран (рис. 7.16).

Для того, чтобы разобраться с возникшей проблемой, найти и устранить ее причину, нам потребуется произвести дебаг.

Дебаг, отладка (от англ. debug) – поиск и устранение ошибок, возникающих в процессе функционирования программы.

При аварийном завершении приложения вся необходимая информация выводится в консоль. Зачастую, несмотря на большой объем отображаемых данных, наиболее ценная информация находится в самом верху:

```
2021-06-06 12:05:57.275234+0300 Contacts[63627:1725884] -[Contacts.View
Controller tableView:numberOfRowsInSection:]: unrecognized selector sent to
instance 0x7f8810c0bbf0
```

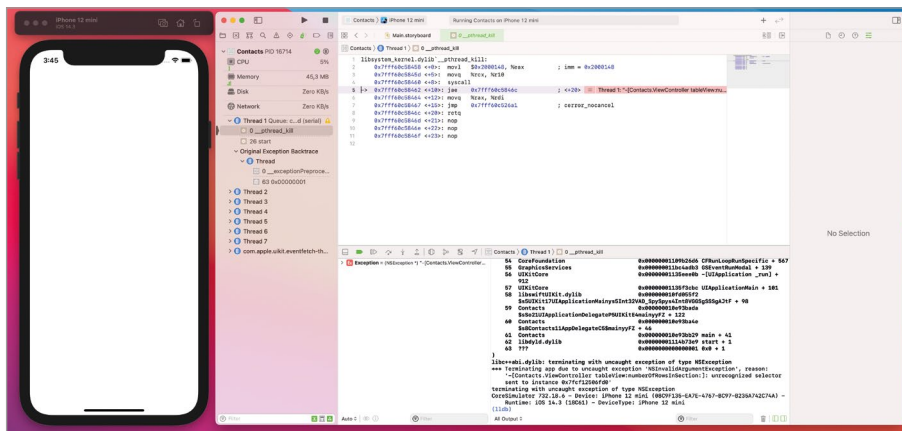


Рис. 7.16. Аварийное завершение приложения

ИЛИ

В приложении Contacts на сцене, работу которой обеспечивает класс `ViewController`, элемент `tableView` попытался вызвать метод `numberOfRowsInSection`, но эта попытка потерпела неудачу.

Попробуем разобраться с причинами такого поведения.

Метод **`numberOfRowsInSection`** должен быть определен в составе источника данных таблицы. Он вызывается автоматически в процессе формирования таблицы и сообщает общее количество строк таблицы. Соответственно, так как источник данных уже определен, а данный метод не реализован – попытка его вызова как раз и привела к ошибке.

Использование шаблонов проектирования «Делегирование» и «Источник данных» в Swift основано на применении протоколов. Так как класс **`ViewController`** является источником для **`Table View`**, его необходимо подписать на специальный протокол **`UITableViewDataSource`**, после чего реализовать несколько специальных методов.

- ▶ Остановите выполнение проекта.
- ▶ В составе проекта откройте файл **`ViewController.swift`**.

Подписать класс **`ViewController`** на протокол **`UITableViewDataSource`** можно двумя способами:

1. Непосредственно при объявлении класса **`ViewController`**:

```
class ViewController: UIViewController, UITableViewDataSource {
    // ...
}
```

2. Используя расширение (extension) класса **ViewController**:

```
extension ViewController: UITableViewDataSource {  
    // ...  
}
```

Оба способа полностью идентичны с точки зрения предоставляемой функциональности, но использование расширения позволит улучшить читабельность кода класса.

ПРИМЕЧАНИЕ Шаблон MVC, несмотря на все свои плюсы, медленно, но уверенно приводит к тому, что код контроллера разрастается до неприличных размеров. Эта проблема носит название Massive View Controller. Самое плохое, что может произойти – это ухудшение навигации по коду проекта. Именно по этой причине использование различных способов улучшения читабельности, например, использование расширений и разнесение по ним различных зон ответственности, является очень хорошей практикой.

- ▶ Создайте расширение для класса **ViewController** и подпишите его на протокол **UITableViewDataSource** (листинг 7.1).

ЛИСТИНГ 7.1

```
extension ViewController: UITableViewDataSource {}
```

Протокол **UITableViewDataSource** требует, чтобы в классе были реализованы два метода:

- **numberOfRowsInSection**, возвращающий общее количество строк в таблице;
 - **cellForRowAt**, возвращающий экземпляр ячейки для конкретной строки таблицы.
- ▶ В расширении класса **ViewController** реализуйте данные методы в соответствии с листингом 7.2.

ЛИСТИНГ 7.2

```
extension ViewController: UITableViewDataSource {  
    func tableView(_ tableView: UITableView, numberOfRowsInSection section:  
Int) -> Int {  
        return 50  
    }  
  
    func tableView(_ tableView: UITableView, cellForRowAt indexPath:  
IndexPath) -> UITableViewCell {  
        // получаем экземпляр ячейки  
        let cell = UITableViewCell(style: .default, reuseIdentifier: nil)  
  
        // конфигурируем ячейку
```

```
var configuration = cell.defaultContentConfiguration()
configuration.text = "Строка \(indexPath.row)"
cell.contentConfiguration = configuration

// возвращаем сконфигурированный экземпляр ячейки
return cell
}
}
```

► Запустите приложение.

Теперь наше приложение содержит 50 строк (рис. 7.17). Попробуйте переместиться по таблице вверх и вниз. Это действие не вызовет у вас каких-либо трудностей, так как **UITableView** по умолчанию поддерживает вертикальное перемещение.

Разберем каждый из методов более подробно.

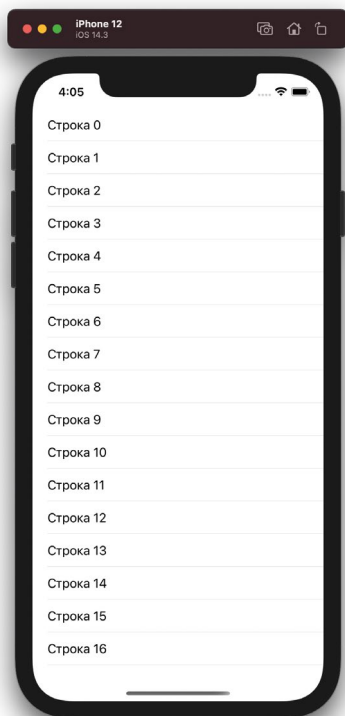


Рис. 7.17. Табличное представление с 50-ю строками

Метод `numberOfRowsInSection`

СИНАКСИС

Метод `UITableViewDataSource.tableView(_:numberOfRowsInSection:) -> Int`
Возвращает общее количество строк в таблице, которые потенциально могут быть отображены.

Аргументы

- `_`: `UITableView` – экземпляр табличного представления, для которого определяется количество строк.
- `numberOfRowsInSection: Int` – индекс секции, для которой определяется количество строк.

Возвращаемое значение

- `Int` – количество строк в данной секции таблицы

В текущей реализации табличное представление будет содержать всего 50 строк, так как метод `numberOfRowsInSection` возвращает число 50. Позже, когда мы займемся разработкой Модели, данный метод будет возвращать количество записанных контактов.

Обратите внимание, что во входном параметре `section` передается индекс секции. Это значение используется при работе с многосекционной таблицей, с которой мы еще столкнемся позднее. Если кратко, то секции позволяют визуально отделить группы строк друг от друга (рис. 7.18). Каждая секция может содержать произвольное количество строк, а сколько именно как раз и определяется в методе `numberOfRowsInSection`.



Рис. 7.18. Многосекционное табличное представление

Метод `cellForRowAt`

СИНТАКСИС

Метод `UITableViewDataSource.tableView(_:cellForRowAt:) -> UITableViewCell`
Возвращает ячейку, определяющую внешний вид данных, выводимых в конкретной строке.

Аргументы

- `_:` `UITableView` – экземпляр табличного представления, для которого определяется количество строк.
- `cellForRowAt:` `IndexPath` – специальное значение, описывающее путь к строке в таблице, для которой формируется ячейка.

Возвращаемое значение

- `UITableViewCell` – ячейка, выводимая в строке таблицы.

Метод **`cellForRowAt`** вызывается непосредственно перед тем, как будет отображена очередная строка таблицы. Причем для каждой строки таблицы происходит отдельный вызов метода **`cellForRowAt`**, а значит возвращается собственная ячейка (экземпляр типа **`UITableViewCell`**).

В ходе исполнения метода ячейка создается и наполняется данными. Взгляните на листинг 7.2 еще раз — там в первую очередь создается новая ячейка, а потом в свойство **`text`**, описывающее текст в ячейке, устанавливается значение с номером строки.

Для того, чтобы определить, для какой именно строки необходимо сформировать ячейку, используется значение параметра **`indexPath`** (его внешнее имя **`cellForRowAt`**) типа **`IndexPath`** (название и тип практически совпадают, разница лишь в первом символе). В нем описывается, для какой именно строки определенной секции необходимо сформировать ячейку. Для этого используются следующие свойства:

- **`indexPath.section: Int`** – индекс секции.

В нашем случае таблица является односекционной, и значение данного свойства всегда будет 0, поэтому свойство не используется в коде.

- **`indexPath.row: Int`** – индекс строки, для которой формируется ячейка.

В листинге 7.2 свойство **`row`** используется для вывода индекса строки прямо в тексте ячейки.

7.3 Создание и конфигурирование ячеек

При каждом вызове метода **`cellForRowAt`** создается, конфигурируется и возвращается ячейка таблицы (экземпляра типа **`UITableViewCell`**), которая определяет внешний вид данных, отображаемых в таблице.

Swift и Xcode предоставляют широкие возможности для оформления ячеек: помимо стандартных шаблонов, вы можете самостоятельно создавать собственные стили оформления, размещая в ячейке любые графические элементы (текст, картинки, переключатели и т.д.).

В текущей реализации метода **cellForRowAt** для создания ячейки используется инициализатор класса **UITableViewCell**:

```
let cell = UITableViewCell(style: .default, reuseIdentifier: nil)
```

В качестве аргумента **style** как раз и передается указатель на конкретный шаблонный стиль оформления. Всего вам доступны 4 таких стиля.

- **default** – стиль с основным и мелким дополнительным текстом снизу (может отсутствовать) (рис. 7.19).

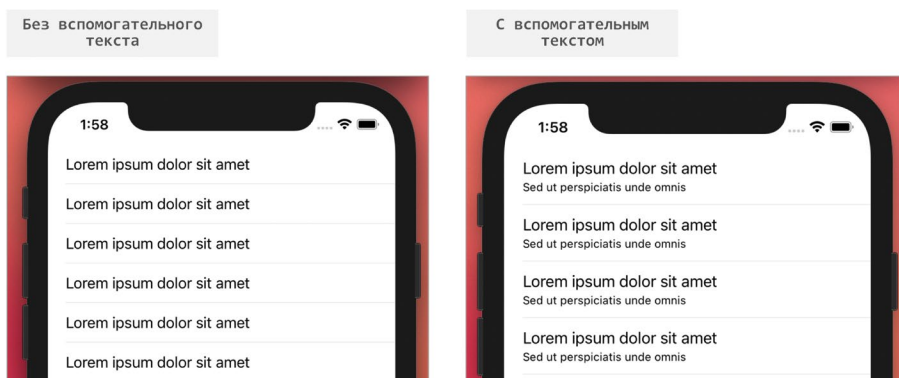


Рис. 7.19. Стиль default

- **value1** – стиль с черным основным и серым вспомогательным текстом. При этом, в зависимости от длины основного текста, вспомогательный может находиться либо в правой, либо в нижней части ячейки (рис. 7.20).
- **subtitle** – стиль с основным и мелким дополнительным текстом с уменьшенными отступами (рис. 7.21).



Рис. 7.20. Стиль value1

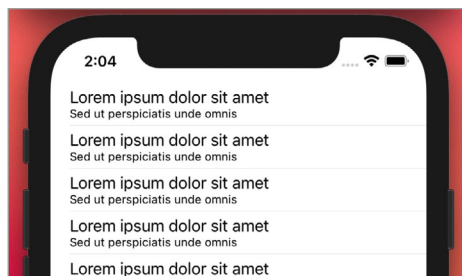


Рис. 7.21. Стиль subtitle

Примечание Так же существует стиль **value2**, но в iOS 14 при использовании новых механизмов конфигурации ячеек (их мы рассмотрим через несколько страниц) он будет выглядеть точно так же, как и **value1**. В iOS 13 же он выглядел так, как показано на рисунке 7.22.

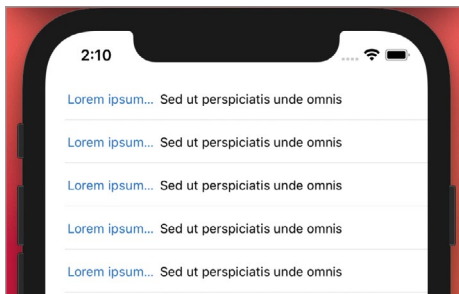


Рис. 7.22. Стиль value2 в iOS 13

Вы можете группировать различные стили оформления в соответствии с вашей задачей. Более того, предустановленные стили также позволяют использовать различные вспомогательные элементы, вроде картинок и графических элементов. На рисунке 7.23 показан пример оформления строк таблицы с применением различных стандартных стилей.

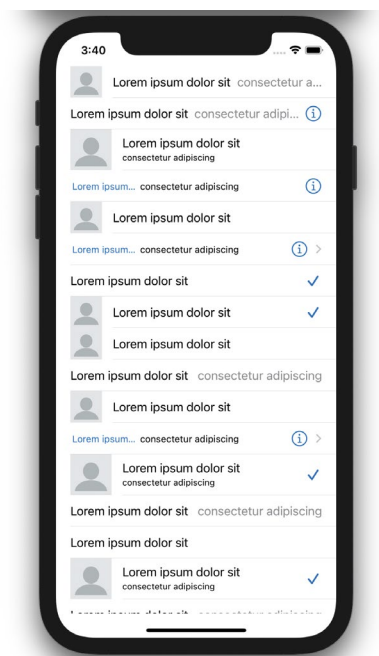


Рис. 7.23. Таблица с различными вариантами оформления ячеек

Примечание Показанный вариант интерфейса не имеет никакой практической ценности, так как был сгенерирован случайным образом исключительно с целью демонстрации возможностей применения различных стилей оформления.

Рассмотрим более подробно инициализатор типа **UITableViewCell**.

СИНТАКСИС

Инициализатор `UITableViewCell(style:reuseIdentifier:)`

Возвращает ячейку на основе предустановленного стиля оформления

Входные параметры

- `style: UITableViewCell.CellStyle` – значение перечисления, определяющее стиль оформления.
- `reuseIdentifier: String?` – текстовый идентификатор переиспользуемой ячейки.

Только что созданная ячейка не содержит каких-либо данных – она пуста. Для ее наполнения или, иными словами, конфигурирования, необходимо выполнить несколько операций.

Посмотрим на код, используемый в данный момент для конфигурирования ячейки:

```
var configuration = cell.defaultContentConfiguration()
configuration.text = "Строка \(indexPath.row)"
cell.contentConfiguration = configuration
```

Параметр **cell** содержит ячейку, созданную с помощью инициализатора класса **UITableViewCell**.

За оформление ячейки отвечает значение типа **UIListContentConfiguration**, которое может содержать в себе данные о тексте и картинках, выводимых в ячейке. В результате вызова метода **defaultContentConfiguration()** возвращается значение типа **UIListContentConfiguration**, которое является неким оформлением ячейки «по умолчанию», то есть таким оформлением, которое не содержит каких-либо данных.

Далее с помощью специальных свойств мы наполняем эту стандартную пустую конфигурацию данными и передаем свойству **contentConfiguration** ячейки.

Повторим еще раз.

1. Метод **defaultContentConfiguration()** возвращает пустую конфигурацию ячейки.
2. Данная пустая конфигурация наполняется данными.
3. Наполненная конфигурация передается ячейке.

После проделанных действий ячейка наполнена данными, что мы и видим при запуске приложения.

В нашем случае мы использовали свойства **text** параметра **configuration**, но тип **UICollectionContentConfiguration** позволяет использовать и другие свойства.

► Откройте документацию к типу данных **UICollectionContentConfiguration**.

В разделе **Customizing Content** перечислены доступные свойства, а в разделе **Customizing Appearance** — свойства, с помощью которых можно изменить оформление ячейки: шрифт, цвет текста, оформление картинки и т.д. (рис. 7.24).

Customizing Content	<pre>var image: UIImage? The image to display. var text: String? The primary text. var attributedText: NSAttributedString? An attributed variant of the primary text. var secondaryText: String? The secondary text. var secondaryAttributedText: NSAttributedString? An attributed variant of the secondary text.</pre>
Customizing Appearance	<pre>var imageProperties: UICollectionViewContentConfiguration.ImageProperties Properties for configuring the image. var textProperties: UICollectionViewContentConfiguration.TextProperties Properties for configuring the primary text. var secondaryTextProperties: UICollectionViewContentConfiguration.TextProperties Properties for configuring the secondary text. struct UICollectionViewContentConfiguration.ImageProperties Properties that affect the list content configuration's image. struct UICollectionViewContentConfiguration.TextProperties Properties that affect the list content configuration's text.</pre>

Рис. 7.24. Доступные свойства типа UICollectionViewContentConfiguration

Примечание Тип данных **UICollectionContentConfiguration** доступен в версиях iOS, начиная с 14. Если вам требуется обеспечить наполнение ячейки данными при работе приложения в более ранних версиях системы, для этого используются специальные свойства ячейки:

- **UITableViewCell.textLabel: UILabel?** – основная текстовая метка ячейки;
- **UITableViewCell.detailTextLabel: UILabel?** – вспомогательная текстовая метка ячейки;
- **UITableViewCell.imageView: UIImageView?** – изображение в ячейке.

В iOS 14 же данные свойства помечены как устаревшие (deprecated).

Вообще перед программистами нередко встает вопрос о реализации определенной функциональности для различных версий операционных систем. При этом, как в примере выше, в каждой системе могут/должны использоваться различные механизмы. В этом случае применяются специальные **атрибуты доступности** (Availability Attributes), позволяющие проверить текущую версию операционной системы и, в зависимости от этого, решать задачу различными способами.

```
if #available(iOS 14.0, *) {  
    // реализуем функциональность для iOS 14 и старше  
} else {  
    // реализуем функциональность для iOS 13 и младше  
}
```

Но как вы понимаете, такой подход требует от разработчика выполнения двойной работы, поскольку необходимо отдельно реализовать функциональность для старых и новых версий систем.

Переиспользуемые ячейки

Одной из важнейших функций табличных представлений является применение переиспользуемых ячеек, о которых мы уже говорили ранее. Как вы думаете, используется ли эта возможность в нашей текущей реализации метода **cellForRowAt**?

Нет!

В данный момент для каждой строки таблицы создается свой собственный экземпляр типа **UITableViewCell**, а значит ни о какой экономии ресурсов говорить не приходится. Для исправления ситуации нам необходимо внести некоторые правки в программный код.

В первую очередь — текстовый идентификатор переиспользуемых ячеек.

- В методе **cellForRowAt** в инициализаторе класса **UITableViewCell** для аргумента **reuseIdentifier** укажите текстовое значение **contactCellIdentifier** (листинг 7.3).

ЛИСТИНГ 7.3

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)  
-> UITableViewCell {  
    let cell = UITableViewCell(style: .default, reuseIdentifier:  
"contactCellIdentifier ")  
    var configuration = cell.defaultContentConfiguration()  
    configuration.text = "Строка \(indexPath.row)"  
    cell.contentConfiguration = configuration  
    return cell  
}
```

Какой именно идентификатор указать — это вопрос, который полностью лежит в вашей зоне ответственности. Важно, чтобы вы придерживались следующих правил:

1. идентификатор должен давать четкое представление о том, для чего именно предназначена ячейка.

2. идентификатор должен быть уникальным в пределах одного табличного представления; если в таблице используется несколько видов ячеек, необходимо указать различные идентификаторы для каждого из этих видов.

Теперь создаваемая ячейка имеет уникальный идентификатор, с помощью которого она может быть повторно использована. Но сам по себе идентификатор не приводит к тому, что для нее активируется функция повторного использования. Для этого перед созданием каждой новой ячейки необходимо осуществлять проверку условия, есть ли в памяти неиспользуемая ячейка, которая может быть повторно использована, точно так же, как это было изображено на рисунке 7.8:

- если такая ячейка есть, то необходимо использовать ее;
- если такой ячейки нет, необходимо создать новую, не забыв указать для нее соответствующий идентификатор.

Для решения этой задачи используется метод **UITableView.dequeueReusableCell(withIdentifier:)**, который осуществляет попытку загрузки переиспользуемой ячейки с определенным идентификатором и возвращает либо **nil** (если такая ячейка отсутствует), либо значение типа **UITableViewCell** (если такая ячейка найдена).

► Внесите изменения в метод **cellForRowAt** в соответствии с листингом 7.4.

ЛИСТИНГ 7.4

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
    // производим попытку загрузки переиспользуемой ячейки
    guard let cell = tableView.dequeueReusableCell(withIdentifier: "MyCell")
    else {
        print("Создаем новую ячейку для строки с индексом \(indexPath.
row)")
        let newCell = UITableViewCell(style: .default, reuseIdentifier:
"MyCell")
        var configuration = newCell.defaultContentConfiguration()
        configuration.text = "Строка \(indexPath.row)"
        newCell.contentConfiguration = configuration
        return newCell
    }
    print("Используем старую ячейку для строки с индексом \(indexPath.
row)")
    return cell
}
```

Теперь метод **cellForRowAt** работает по следующему принципу.

- В операторе **guard** производится попытка загрузки переиспользуемой ячейки.
- Если такая ячейка есть, она используется для текущей строки.
- Если такой ячейки нет, то создается новая.

В методе мы предусмотрели осуществление вывода специальных текстовых сообщений на консоль для определения того, была ли ячейка создана или загружена.

► Запустите проект на симуляторе.

При загрузке сцены в памяти нет еще ни одной переиспользуемой ячейки, поэтому для каждой строки создается своя собственная ячейка. Об этом, в том числе, говорит вывод на консоли. Поэтому мы видим ровно столько сообщений о создании новой ячейки, сколько строк поместилось на экран (рис. 7.25).

```
Создаем новую ячейку для строка с индексом 0
Создаем новую ячейку для строка с индексом 1
Создаем новую ячейку для строка с индексом 2
Создаем новую ячейку для строка с индексом 3
Создаем новую ячейку для строка с индексом 4
Создаем новую ячейку для строка с индексом 5
Создаем новую ячейку для строка с индексом 6
Создаем новую ячейку для строка с индексом 7
Создаем новую ячейку для строка с индексом 8
Создаем новую ячейку для строка с индексом 9
Создаем новую ячейку для строка с индексом 10
Создаем новую ячейку для строка с индексом 11
Создаем новую ячейку для строка с индексом 12
Создаем новую ячейку для строка с индексом 13
Создаем новую ячейку для строка с индексом 14
Создаем новую ячейку для строка с индексом 15
Создаем новую ячейку для строка с индексом 16
Создаем новую ячейку для строка с индексом 17
```

Рис. 7.25. Вывод на консоли

```
Создаем новую ячейку для строка с индексом 2
Создаем новую ячейку для строка с индексом 3
Создаем новую ячейку для строка с индексом 4
Создаем новую ячейку для строка с индексом 5
Создаем новую ячейку для строка с индексом 6
Создаем новую ячейку для строка с индексом 7
Создаем новую ячейку для строка с индексом 8
Создаем новую ячейку для строка с индексом 9
Создаем новую ячейку для строка с индексом 10
Создаем новую ячейку для строка с индексом 11
Создаем новую ячейку для строка с индексом 12
Создаем новую ячейку для строка с индексом 13
Создаем новую ячейку для строка с индексом 14
Создаем новую ячейку для строка с индексом 15
Создаем новую ячейку для строка с индексом 16
Создаем новую ячейку для строка с индексом 17
Используем старую ячейку для строка с индексом 19
```

Рис. 7.26. Вывод на консоли

► Начните медленно перемещать таблицу вверх и при этом смотрите, какие сообщения выводятся на консоль.

До тех пор, пока строка с индексом 0 не будет скрыта с экрана, вы будете видеть сообщения о создании новых ячеек. Но как только первая строка скроется, использованная для нее ячейка станет доступна для повторного применения, а на консоли отобразится уже новое сообщение (рис. 7.26). И сколько бы вы не прокручивали далее, все новые сообщения будут говорить о том, что ячейка используется повторно.

Вот она — сила переиспользуемых ячеек! Такое поведение намного более предпочтительно, так как потребляет меньше вычислительных ресурсов (загрузка из памяти проще и быстрее, нежели создание нового экземпляра).

Но наш текущий код содержит одну очень серьезную ошибку, с которой вы также будете периодически сталкиваться в процессе разработки: все новые ячейки содержат «старый» текст. К примеру, на рисунке 7.27 показано, что сра-

зу после строки с текстом «**Строка 18**» идет ячейка с текстом «**Строка 0**», хотя на консоли при этом указан верный индекс обрабатываемой строки.

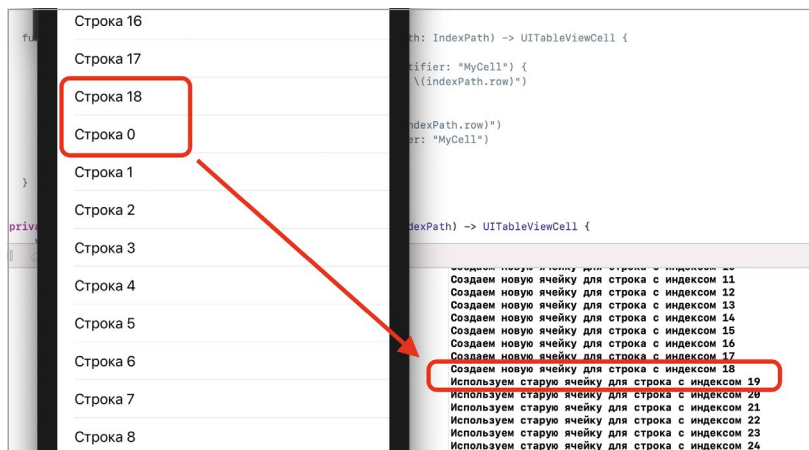


Рис. 7.27. Ошибка при конфигурировании ячеек

Как вы думаете, в чем причина такого поведения?

Решение этой проблемы кроется в том, как именно готовится к отображению переиспользуемая ячейка. Ячейка, хранящаяся в памяти и ожидающая повторного использования, наполнена теми данными, которые были записаны в нее ранее. В текущей реализации метода `cellForRowAt` при появлении переиспользуемой ячейки мы не производим никаких действий по обновлению информации в ней: она просто отправляется в таблицу. И это необходимо исправить. Таким образом, получается, что необходимо изменять текст ячейки в двух местах, и чтобы избежать дублирования, вынесем эту функциональность в отдельный метод.

- В расширении к классу `ViewController` реализуйте метод `configure` в соответствии с листингом 7.5.

ЛИСТИНГ 7.5

```
private func configure(cell: inout UITableViewCell, for indexPath:
IndexPath) {
    var configuration = cell.defaultContentConfiguration()
    configuration.text = "Строка \(indexPath.row)"
    cell.contentConfiguration = configuration
}
```

- Доработайте метод `cellForRowAt` таким образом, чтобы в нем происходил вызов метода `configure` вне зависимости от того, используется ли повторно ячейка, или создается новая (листинг 7.6).

ЛИСТИНГ 7.6

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
    guard var cell = tableView.dequeueReusableCell(withIdentifier: "MyCell")
    else {
        print("Создаем новую ячейку для строки с индексом \(indexPath.
row)")
        var newCell = UITableViewCell(style: .default, reuseIdentifier:
"MyCell")
        configure(cell: &newCell, for: indexPath)
        return newCell
    }
    print("Используем старую ячейку для строки с индексом \(indexPath.
row)")
    configure(cell: &cell, for: indexPath)
    return cell
}
```

► Произведите запуск приложения.

Теперь, как бы вы не перемещались по таблице, в ее строках всегда будут корректные данные (рис. 7.28).

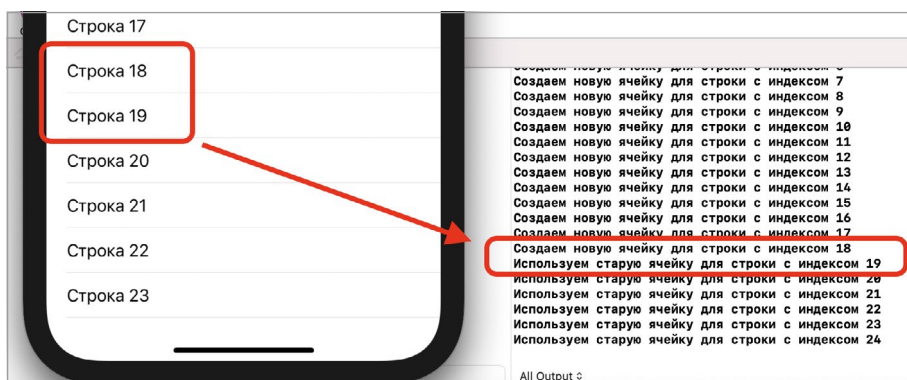


Рис. 7.28. Корректное конфигурирование ячеек

Обратите внимание на то, что хоть в таблице и выводится 50 строк, но индекс последней строки – 49, о чем говорит текст в соответствующей ячейке.

Несмотря на то, что код работает корректно, переработаем метод **cellForRowAt** таким образом, чтобы убрать двойное использование оператора **return** и двойной вызов метода **configure**.

► Измените метод **cellForRowAt** в соответствии с листингом 7.7.

ЛИСТИНГ 7.7

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
    var cell: UITableViewCell
    if let reuseCell = tableView.dequeueReusableCell(withIdentifier:
"MyCell") {
        print("Используем старую ячейку для строки с индексом \(indexPath.
row)")
        cell = reuseCell
    } else {
        print("Создаем новую ячейку для строки с индексом \(indexPath.
row)")
        cell = UITableViewCell(style: .default, reuseIdentifier: "MyCell")
    }
    configure(cell: &cell, for: indexPath)
    return cell
}
```

В результате мы получили очень качественный программный код, который может быть с легкостью преобразован для работы с сущностью «Контакт».

Примечание То, что мы вынесли конфигурирование ячейки в метод `configure`, в текущих условиях сыграло бы нам на руку в случае, если приложению потребовалась бы поддержка iOS 13 или более ранних версий системы. С помощью атрибутов доступности мы могли бы с легкостью реализовать оба варианта конфигурирования системы:

```
private func configure(cell: inout UITableViewCell, for indexPath:
IndexPath) {
    if #available(iOS 14, *) {
        var configuration = cell.defaultContentConfiguration()
        configuration.text = "Строка \(indexPath.row)"
        cell.contentConfiguration = configuration
    } else {
        cell.textLabel?.text = "Строка \(indexPath.row)"
    }
}
```

7.4 Разработка модели

Теперь табличные представления не являются для вас черным ящиком. Вы познакомились с основами их функционирования, а значит можете двигаться дальше. В этом разделе мы перейдем к разработке Модели и наполнению приложения данными.

Данными в приложении являются контакты, именно они будут отображаться в строках таблицы. Сущность «Контакт» реализуем посредством структуры **Contact**.

- ▶ В структуре проекта создайте папку **Model**.
- ▶ В папке **Model** создайте файл **Contact.swift**.
- ▶ В файле **Contact.swift** реализуйте протокол **ContactProtocol** и структуру **Contact** в соответствии с листингом 7.8.

ЛИСТИНГ 7.8

```
protocol ContactProtocol {  
    /// Имя  
    var title: String { get set }  
    /// Номер телефона  
    var phone: String { get set }  
}  
  
struct Contact: ContactProtocol {  
    var title: String  
    var phone: String  
}
```

Структура **Contact** содержит два свойства, определяющих имя контакта и его номер телефона. Значения этих свойств будут выводиться в ячейках таблицы.

Теперь свяжем сцену и созданную Модель.

- ▶ Откройте файл **ViewController.swift**.
- ▶ В классе **ViewController** создайте приватное свойство **contacts** (листинг 7.9).

ЛИСТИНГ 7.9

```
class ViewController: UIViewController {  
    private var contacts = [ContactProtocol]()  
    // ...  
}
```

Свойство **contacts** – это массив контактов, элементы которого будут выведены в табличном представлении. При загрузке сцены данное свойство будет наполняться данными, а впоследствии использоваться для наполнения ячеек таблицы данными.

Примечание Обратите внимание, что в качестве типа переменной мы используем протокол **ContactProtocol**, а не тип **Contacts**. Конечно, ничего страшного не будет, если мы будем ссылаться и на конкретный тип, но у такого подхода есть несколько преимуществ.

- Использование протоколов вместо конкретных типов позволяет снизить зависимость отдельных элементов программы друг от друга. Так класс **ViewController** не будет зависеть от типа **Contact** и наоборот. При изменении одного элемента нет необходимости вносить изменения в другой.
- Если появятся такие условия, вследствие которых нам потребуется создать новый тип, описывающий сущность **Contact**, мы сможем сделать это без каких-либо проблем и в дальнейшем использовать в приложении вместо **Contact**. Главное, чтобы новый тип соответствовал протоколу **ContactProtocol**.

Уменьшение зависимости элементов системы друг от друга считается хорошей практикой создания архитектуры приложений. Вы будете постигать эту тему постепенно, читая книги от таких авторов, как Роберт Мартин, Дэвид Томас и многих других. Совершенствование своих профессиональных навыков – это неизбежная судьба любого программиста, который хочет стать **Senior Swift Developer**.

Минусом таких больших примечаний является то, что они уведут вас в сторону от рассматриваемого материала. Но в каждом таком примечании я стараюсь дать важную для вас информацию. Старайтесь не игнорировать их и при необходимости делать заметки.

А теперь вернемся к работе с Моделью.

В момент инициализации сцены свойство **contacts** – это пустой массив, но с целью тестирования работы приложения нам необходимо сделать так, чтобы в нем содержались какие-либо данные.

- Создайте приватный метод **loadContacts**, наполняющий свойство **contacts** тестовыми данными (листинг 7.10).

ЛИСТИНГ 7.10

```
private func loadContacts() {
    contacts.append(
        Contact(title: "Саня Техосмотр", phone: "+799912312323"))
    contacts.append(
        Contact(title: "Владимир Анатольевич", phone: "+781213342321"))
    contacts.append(
        Contact(title: "Сильвестр", phone: "+7000911112"))
    contacts.sort{ $0.title < $1.title }
}
```

- Дополните метод **viewDidLoad** вызовом метода **loadContacts** (листинг 7.11).

ЛИСТИНГ 7.11

```
override func viewDidLoad() {
    super.viewDidLoad()
    loadContacts()
}
```

Теперь внесем изменения в методы **numberOfRowsInSection** (для корректного определения количества выводимых строк) и **configure** (для корректного заполнения ячейки данными).

- Внесите правки в методы **numberOfRowsInSection** и **configure** в соответствии с листингом 7.12.

ЛИСТИНГ 7.12

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    return contacts.count
}

private func configure(cell: inout UITableViewCell, for indexPath:
IndexPath) {
    var configuration = cell.defaultContentConfiguration()
    // имя контакта
    configuration.text = contacts[indexPath.row].title
    // номер телефона контакта
    configuration.secondaryText = contacts[indexPath.row].phone
    cell.contentConfiguration = configuration
}
```

Метод **numberOfRowsInSection** возвращает число, соответствующее количеству элементов в свойстве **contacts**. Тут все предельно ясно.

В методе **configure** для вывода текста используется конкретный элемент массива **contacts**, индекс которого соответствует индексу выводимой строки таблицы.

Для конфигурирования ячейки используется два свойства: **text** и **secondaryText**. Первое определяет основной текст в ячейке (имя контакта), а второе – вспомогательный (номер телефона).

- Запустите приложение (рис. 7.29).

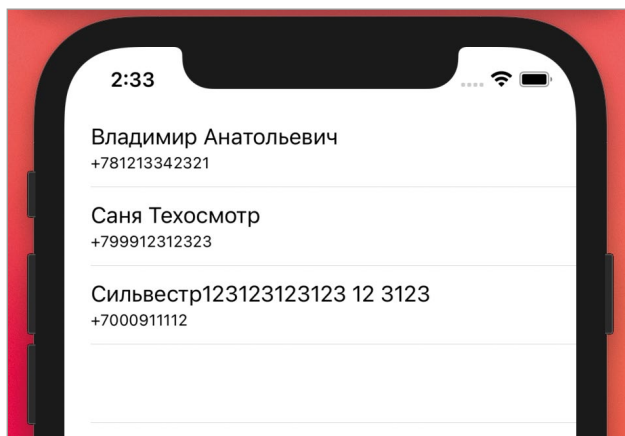


Рис. 7.29. Таблица с данными о контактах

Таблица выводит тестовые контакты! Если не учитывать теоретическую подготовку, согласитесь – это было довольно просто. Табличные представления дают нам поистине широкие возможности и позволяют очень гибко настраивать графический интерфейс приложений. На рисунке 7.30. показана одна из сцен приложения от магазина «ВкусВилл», созданная на основе Table View. Каждая ячейка таблицы имеет свой собственный стиль оформления и состоит из множества вложенных элементов.

Со временем вы освоите создание так называемых кастомных ячеек и научитесь оформлять свои приложения, ограничиваясь исключительно вашей фантазией.

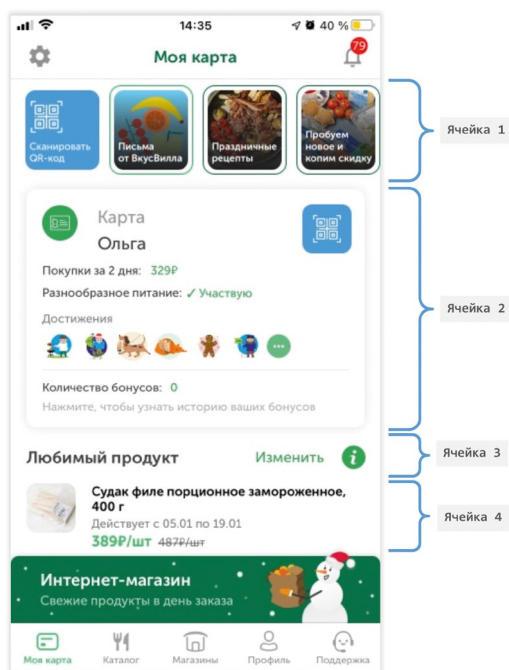


Рис. 7.30. Приложение на основе табличного представления

7.5 Удаление контактов

Сейчас список контактов в программе статичен, так как пользователь не может изменять его. В этом и следующем разделах мы займемся разработкой функциональности, позволяющей удалять, добавлять и редактировать контакты. При этом внесенные изменения пока будут сохраняться только в рамках текущего запуска приложения, а при каждой перезагрузке вы будете видеть все тот же первичный тестовый набор данных. Вопросом долговременного хранения мы займемся в следующей главе.

Вообще, редактирование списка контактов – довольно простая задача. Все, что нам нужно сделать, это:

1. изменить значение свойства **contacts**, удалив существующее или добавив новое значение;
2. обновить табличное представление, чтобы отобразить актуальный список контактов.

В первую очередь реализуем удаление и для этого воспользуемся стандартным для iOS способом: свайпом. Вы могли видеть такой подход во многих приложениях, например, в «Почте» (рис 7.31). При удалении сообщения все строки, находящиеся ниже, поднимаются вверх, обеспечивая таким образом отсутствие каких-либо пустых строк.

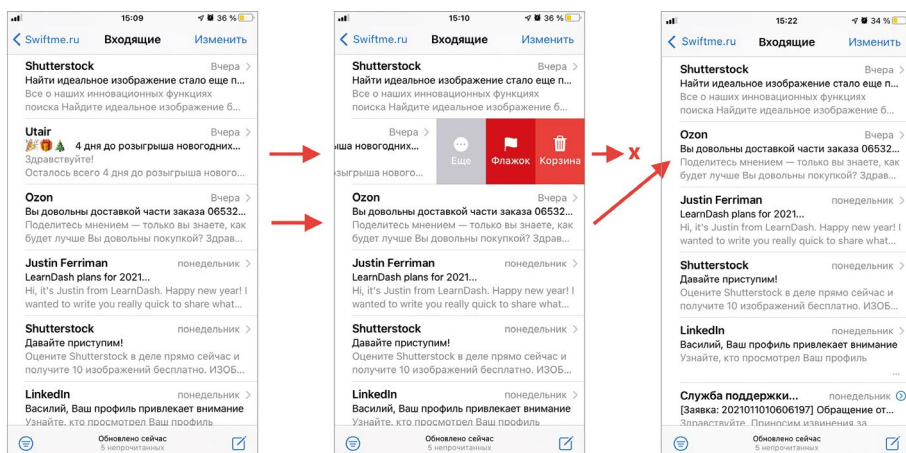


Рис. 7.31. Удаление письма с помощью свайпа по ячейке

Примечание **Свайп** (от англ. *swipe*) – это движение пальцем по экрану в одном направлении. Свайп завершается в момент завершения касания экрана. Соответственно, в процессе свайпа палец не должен отрываться.

Обработка свайпов по строкам табличного представления в iOS организована на основе шаблона проектирования «Делегирование». Таким образом, свайп обрабатывается не самой таблицей, а неким делегатом, которому назначена эта функциональность (точно так же, как это было с наполнением таблицы данными и шаблоном «Источник данных»).

В первую очередь, определимся с тем, какой элемент приложения будет делегатом Table View. На данном этапе наиболее оптимальным выбором (как и в случае с Data Source) станет View Controller.

- Подпишите класс **ViewController** на протокол **UITableViewDelegate**. Для этого создайте новое расширение в файле **ViewController.swift** (листинг 7.13)

ЛИСТИНГ 7.13

```
extension ViewController: UITableViewDelegate {}
```

Теперь необходимо связать View Controller и Table View, указав, что первый является делегатом второго.

- ▶ Откройте файл **Main.storyboard**.
- ▶ Выделите **Table view** на сцене.
- ▶ Откройте **Connections inspector**.
- ▶ В разделе **Outlets** свяжите поле **delegate** с элементом **View Controller** на сцене (рис. 7.32).

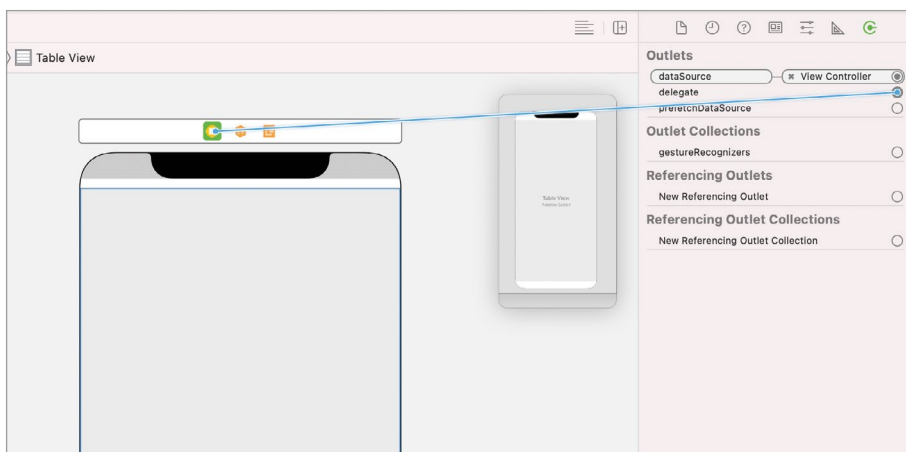


Рис. 7.32. Связь View Controller и Table View

Теперь **View Controller** является делегатом табличного представления, и мы можем использовать все возможности протокола **UITableViewDelegate**.

При свайпе по таблице вызывается один из двух методов делегата (в зависимости от того, в какую сторону происходит свайп):

- **tableView(_:trailingSwipeActionsConfigurationForRowAt:)** – при свайпе влево.
- **tableView(_:leadingSwipeActionsConfigurationForRowAt:)** – при свайпе вправо.

Примечание По устоявшейся традиции будем называть эти методы **trailingSwipeActionsConfigurationForRowAt** и **leadingSwipeActionsConfigurationForRowAt** по имени первого уникального аргумента.

Каждый из методов возвращает сконфигурированное значение, определяющее перечень доступных действий: кнопок, отображаемых при свайпе, и реакций на

нажатие этих кнопок. Обычно, для удаления используется свайп влево, поэтому мы реализуем только метод **trailingSwipeActionsConfigurationForRowAt**. При этом оба метода идентичны по своим возможностям, а значит вы сможете использовать **leadingSwipeActionsConfigurationForRowAt** аналогичным способом.

Примечание Теперь файл **ViewController.swift** содержит два расширения класса **ViewController**. Чтобы не запутаться, я буду называть их делегат-расширение и датасорс-расширение.

- В делегат-расширении объявите метод **trailingSwipeActionsConfigurationForRowAt** (листинг 7.14).

ЛИСТИНГ 7.14

```
func tableView(_ tableView: UITableView, trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {}
```

СИНТАКСИС

Метод `UITableViewDelegate.tableView(_: trailingSwipeActionsConfigurationForRowAt:)` -> `UISwipeActionsConfiguration?`

Возвращает объект, описывающий множество действий, доступных при свайпе по ячейке влево.

Входные параметры

- `_`: `UITableView` – табличное представление, в котором производится свайп.
- `trailingSwipeActionsConfigurationForRowAt: IndexPath` – индекс строки, по которой производится свайп.

Выходное значение

- `UISwipeActionsConfiguration?` – определяет множество доступных действий для конкретной строки таблицы.

Данный метод вызывается отдельно для каждой строки таблицы в тот момент, когда пользователь делает свайп по ней. С помощью параметра **trailingSwipeActionsConfigurationForRowAt** вы можете определить, для какой конкретно ячейки необходимо вернуть массив действий. Если вернуть **nil**, то ячейка останется на месте и никаких доступных действий не отобразится.

- Дополните метод **trailingSwipeActionsConfigurationForRowAt** в соответствии с листингом 7.15.

ЛИСТИНГ 7.15

```
func tableView(_ tableView: UITableView, trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {  
    print("Определяем доступные действия для строки \(indexPath.row)")  
    return nil  
}
```


- Запустите приложение и сделайте несколько свайпов по разным ячейкам.

Как только вы делаете свайп по определенной ячейке, на консоли появляется вывод, сообщающий об этом (рис. 7.33). При этом свайп вправо остается вообще без внимания, так как метод `leadingSwipeActionsConfigurationForRowAt` не объявлен.



Рис. 7.33. Свайп влево по ячейке таблицы

В текущий момент ничего кроме сообщения на консоли при свайпе мы не видим, удаление ячейки не происходит.

Параметр типа `UISwipeActionsConfiguration`, который возвращается в результате вызова метода описывает множество всех доступных действий, отображаемых при свайпе по ячейке. Каждое отдельное действие (в нашем случае оно будет всего одно) представлено с помощью значения типа `UIContextualAction`, которое нам также требуется создать внутри метода `trailingSwipeActionsConfigurationForRowAt`.

- Реализуйте тело метода `trailingSwipeActionsConfigurationForRowAt` в соответствии с листингом 7.16.

ЛИСТИНГ 7.16

```

func tableView(_ tableView: UITableView, trailingSwipeActionsConfigurationFo
rRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {
    // действие удаления
    let actionDelete = UIContextualAction(style: .destructive, title: "Уда-
лить") { _,_,_ in
  
```

```
// удаляем контакт
self.contacts.remove(at: indexPath.row)
// заново формируем табличное представление
tableView.reloadData()
}
// формируем экземпляр, описывающий доступные действия
let actions = UISwipeActionsConfiguration(actions: [actionDelete])
return actions
}
```

И это все, что нам нужно сделать!

- ▶ Запустите приложение.
- ▶ Попробуйте удалить все записи с помощью свайпа влево.

Контакты успешно удаляются, а после перезагрузки приложения восстанавливаются вновь, так как мы все еще не реализовали долговременное хранение.

Разберем подробнее, что именно было сделано для удаления записей.

1. В первую очередь создается значение типа **UIContextualAction**, описывающее одно действие, доступное при свайпе. В инициализатор типа **UIContextualAction** передаются три аргумента: **style**, **title** и замыкание.

Первые два значения описывают внешний вид отображаемого элемента (рис. 7.34).

Третий аргумент (замыкание) определяет конкретные операции, которые будут выполнены при активации действия. В нашем случае мы удаляем соответствующий строке контакт из хранилища

self.contacts.remove(at: indexPath.row)

и перезагружаем таблицу

tableView.reloadData()

Метод **reloadData()** применяется к табличному представлению и позволяет заново сформировать и отобразить его на экране.

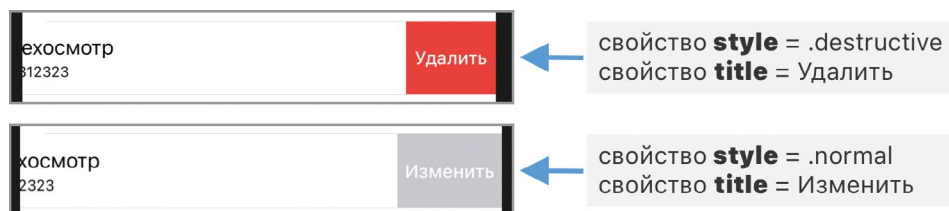


Рис. 7.34. Влияние свойств на оформление действий

2. Далее значение типа **UIContextualAction** передается в инициализатор класса **UISwipeActionsConfiguration**. При необходимости создания двух и более действий, каждое из них создается отдельно, после чего все действия передаются при создании значения типа **UISwipeActionsConfiguration**.

UISwipeActionsConfiguration(actions: [edit, share, delete])

Примечание Указанные выше действия (**edit**, **share**, **delete**) – это не какие-либо стандартные доступные действия: каждое из них должно быть создано с помощью класса **UIContextualAction**.

3. Наконец, мы возвращаем созданное значение типа **UISwipeActionsConfiguration**.

Таким образом, ячейки таблицы получают объект, описывающий доступные при свайпе действия. Напомню, что методы **trailingSwipeActionsConfigurationForRowAt** и **leadingSwipeActionsConfigurationForRowAt** вызываются независимо для каждой ячейки таблицы в момент свайпа.

На этом мы завершаем работу с удалением контактов и переходим к реализации функции создания новых контактов.

7.6 Создание контактов

Без функции добавления новых контактов наша программа не имеет никакой практической ценности. В этом разделе мы займемся ее созданием.

Перед началом реализации ответим на следующие вопросы.

1. Каким образом будет производиться переход к интерфейсу создания нового контакта?

Для этого мы разместим специальную панель в нижней части экрана (**Toolbar**) и расположим на ней кнопку «Создать контакт».

2. Как будет выглядеть интерфейс создания контакта?

На данном этапе я не хочу усложнять задачу, поэтому мы воспользуемся стандартным всплывающим окном (**UIAlertController**) с двумя текстовыми полями для ввода имени и телефона.

Размещение на сцене Toolbar

В данный момент сцена скомпонована таким образом, что табличное представление занимает все доступное на ней пространство. Это обеспечивается тем, что ранее нами были созданы ограничения (constraints), определяющие нулевой отступ Table View от корневого представления (рис. 7.35).

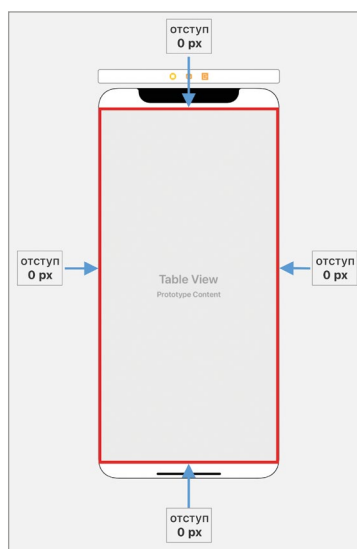


Рис. 7.35. Ограничения, определяющие отступ Table view от корневого представления

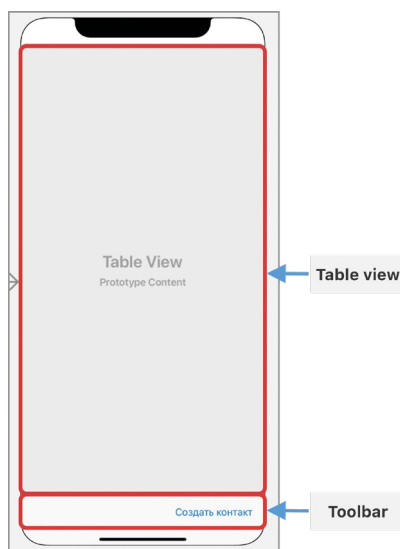


Рис. 7.36. Размещение элементов на сцене

Такое расположение не позволит разместить **Toolbar**, так как таблица всегда будет прижиматься к краям сцены, занимая все свободное пространство. Для решения этой задачи мы выполним ряд действий:

1. удалим нижнее ограничение для **Table view**;
2. поднимем нижний край **Table view**;
3. под **Table view** разместим графический элемент **Toolbar**;
4. создадим ограничение на отступ **Toolbar** слева, справа и снизу от краев сцены;
5. создадим ограничение, определяющее нулевой отступ нижней границы **Table view** от верхней границы **Toolbar**.

В результате проделанных действий мы получим требуемую компоновку элементов, при которой сверху находится **Table view**, а снизу – **Toolbar** (рис. 7.36).

У элемента **Table view** удалим нижний констрейнт (рис. 7.37).

- ▶ Откройте файл **Main.storyboard**.
- ▶ (1) Выделите **Table view**.
- ▶ (2) На панели **Inspectors** откройте **Size Inspector**.

В разделе **Constraints** (3) вы найдете все ограничения, созданные для табличного представления. Здесь вы можете редактировать или удалять их.

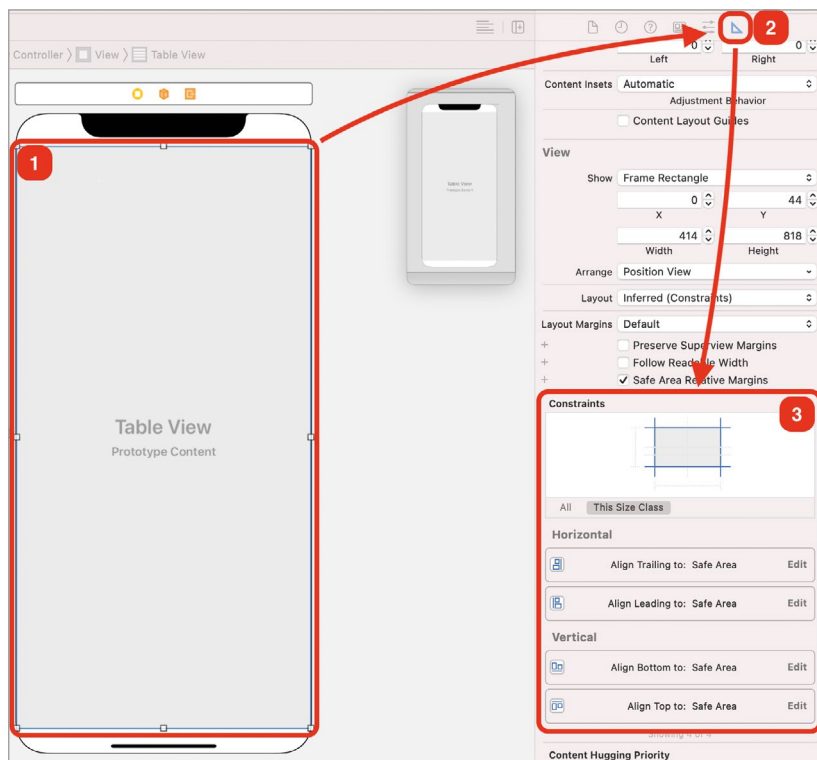


Рис. 7.37. Ограничения элемента

- Удалите ограничение **Align Bottom to: Safe Area**. Для этого выделите данное ограничение и нажмите **Backspace** на клавиатуре.

После проделанных действий на панели **Document Outline** отобразится красный кружок, сообщающий о том, что имеются критические проблемы с позиционированием элементов (рис. 7.38).

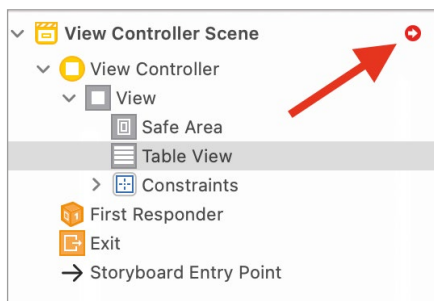


Рис. 7.38. Информационная пиктограмма о наличии проблем позиционирования элементов.

Задание Подумайте, почему теперь в **Document Outline** отображается пиктограмма, сообщающая о критической ошибке позиционирования элементов?

Решение

Дело в том, что теперь **Table view** не имеет привязки к какому-либо элементу снизу, а также не имеет созданных ограничений, определяющих его собственную высоту. Исходя из этого, **UIKit** не может однозначно определить, какого же размера должна быть таблица.

- ▶ Перетяните нижнюю границу табличного представления примерно на середину сцены (рис. 7.39).

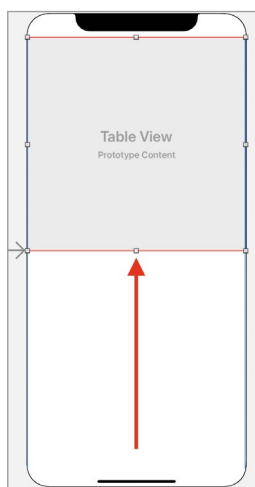


Рис. 7.39. Перемещение нижней границы Table view

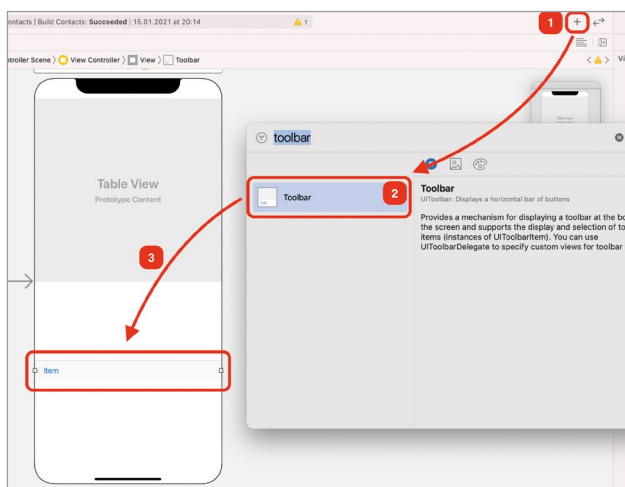


Рис. 7.40. Размещение Toolbar

Теперь разместим **Toolbar** на сцене (рис. 7.40):

- ▶ (1) Откройте библиотеку объектов.
- ▶ (2) Найдите элемент **Toolbar**.
- ▶ (3) Разместите его на сцене в произвольном месте ниже табличного представления.

Задание Создайте ограничения для элемента **Toolbar**, обеспечивающие нулевой отступ от элемента со всех четырех сторон.

Решение

Вы уже неоднократно создавали ограничения ранее, в данном случае задача будет решаться аналогичным способом (рис. 7.41).

- ▶ (1) Выделите **Toolbar** на сцене.
- ▶ (2) Нажмите кнопку **Add New Constraints** в нижней части **Interface Builder**.
- ▶ (3) Во всплывающем окне введите значение 0 во всех четырех полях, определяющих отступ элемента.
- ▶ Нажмите кнопку **Add 4 Constraints**.

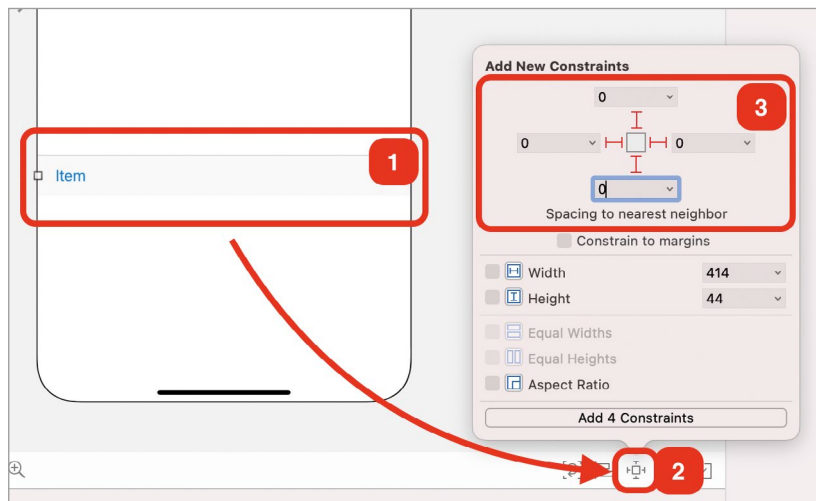


Рис. 7.41. Создание ограничений

Теперь элементы на сцене расположены именно так, как требуется. И это отличный повод поговорить подробнее о том, как работают ограничения.

Подробнее об ограничениях

Примечание Ограничения – это крайне полезная функция, позволяющая создавать адаптивные варианты интерфейса. Чаще всего в различных учебных курсах (или в видео на YouTube) для изучения ограничений выделяют отдельный урок, рассказывая и показывая все их возможности.

Я считаю, что к их рассмотрению необходимо подходить постепенно, именно поэтому я интегрирую работу с ограничениями в учебный материал. Таким образом, вы будете получать первые навыки работы с констрейнтами, излишне не углубляясь в эту тему.

Позже, когда вы будете готовы, мы подробно рассмотрим все возможности ограничений (а также других элементов, обеспечивающих позиционирование графических элементов на сцене).

Создавая ограничения, мы определяем отступ от краев элемента **Toolbar** до краев ближайших графических элементов, расположенных с соответствующей каждому ограничению стороны. Так, слева, справа и снизу ближайшим элементом является корневое представление, в связи с чем **Toolbar** автоматически расположился в нижней части сцены.

Примечание Как я уже говорил ранее, внутри корневого представления выделяется специальная область – **Safe Area**, которая исключает такие системные элементы экрана, как строка статуса сверху и функциональная линия снизу (на беснопочных версиях iPhone).

Верхний констрейнт **Toolbar**, в свою очередь, определяет отступ от этого элемента до **Table view**, так как **Table view** на сцене является ближайшим к **Toolbar** элементом сверху.

В связи с тем, что **Toolbar** имеет фиксированную высоту, он не может растянуться вверх, но **Table view** может спокойно изменять свою высоту, из-за чего он растягивается вниз и прижимается к **Toolbar**.

Таким образом, созданные ограничения позволяют системе однозначно позиционировать графические элементы, вследствие чего красная пиктограмма, сообщающая об ошибках, исчезла.

Ограничение, которое затрагивает несколько элементов, будет отображаться в списке ограничений каждого из них. Так, если вы выделите **Table view** и откроете **Size Inspector**, то увидите ограничение **Bottom space to Toolbar**, определяющее отступ снизу от **Table view** до **Toolbar** (рис. 7.42). При этом, если вы выделите **Toolbar** и откроете **Size Inspector**, то увидите ограничение **Top space to Table view**, определяющее отступ сверху от **Toolbar** до **Table view** (рис. 7.43).

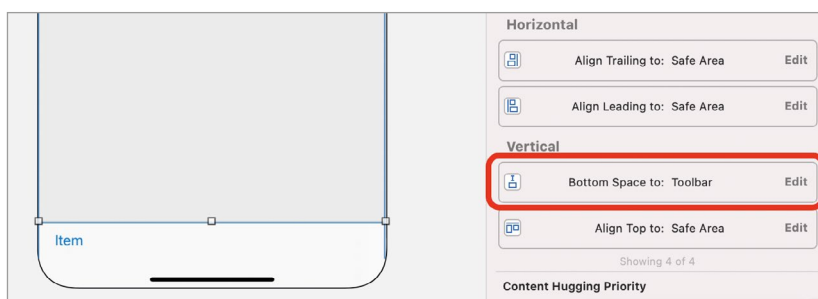


Рис. 7.42. Создание ограничений

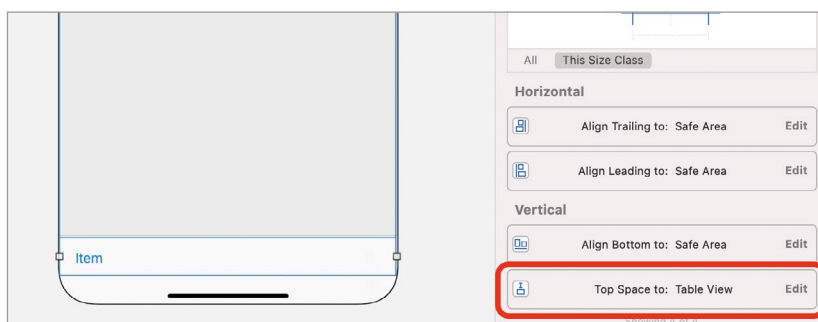


Рис. 7.43. Создание ограничений

Также все созданные ограничения на сцене можно посмотреть в **Document Outline** в разделе **Constraints** (рис. 7.44).

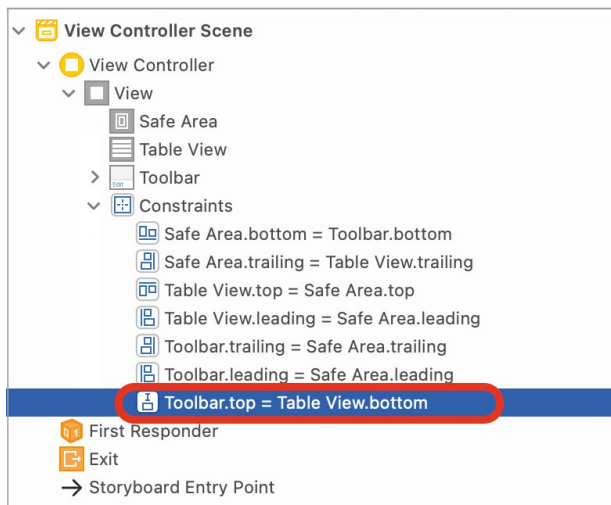


Рис. 7.44. Ограничение между Toolbar и Table view

Настройка Toolbar

Внесем финальные правки во внешний вид панели **Toolbar**.

- ▶ В **Document Outline** выделите элемент **Item**, расположенный в **Toolbar**.
- ▶ Откройте **Attributes Inspector**.
- ▶ Измените значение в поле **Title** на «Создать контакт».
- ▶ Откройте библиотеку объектов.
- ▶ Найдите объект **Flexible Space Bar Button Item** и перетяните его в **Toolbar**, расположив левее кнопки.

Элемент **Flexible Space Bar Button Item** предназначен для того, чтобы растягиваться, заполняя все доступное место пустым пространством. Благодаря этому кнопка сдвинулась до упора вправо.

На этом мы завершаем работу над внешним оформлением и переходим к реализации функциональности создания новых контактов.

Создание всплывающего окна

Для создания контактов будет использоваться всплывающее окно с двумя текстовыми полями (рис. 7.45). Вся прелесть такого метода в том, что он реализуется буквально в несколько строк кода.

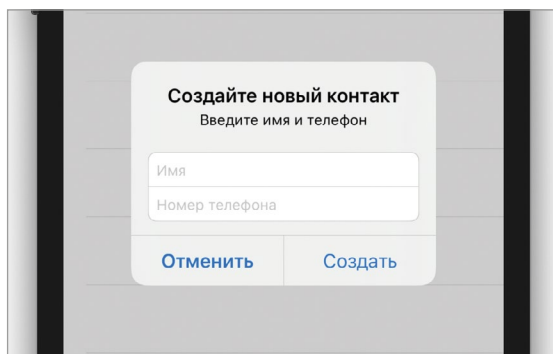


Рис. 7.45. Всплывающее окно

- ▶ В классе **ViewController** объявите новое аулет-свойство **tableView** (листинг 7.17).

ЛИСТИНГ 7.17

```
@IBOutlet var tableView: UITableView!
```

- ▶ В классе **ViewController** объявите новый action-метод **showNewContactAlert()** (листинг 7.18).

ЛИСТИНГ 7.18

```
@IBAction func showNewContactAlert() {  
    // создание Alert Controller  
    let alertController = UIAlertController(title: "Создайте новый кон-  
такт", message: "Введите имя и телефон", preferredStyle: .alert)  
  
    // добавляем первое текстовое поле в Alert Controller  
    alertController.addTextField { textField in  
        textField.placeholder = "Имя"  
    }  
  
    // добавляем второе текстовое поле в Alert Controller  
    alertController.addTextField { textField in  
        textField.placeholder = "Номер телефона"  
    }  
  
    // создаем кнопки  
    // кнопка создания контакта  
    let createButton = UIAlertAction(title: "Создать", style: .default) {  
_ in  
        guard let contactName = alertController.textFields?[0].text,
```

```

        let contactPhone = alertController.textFields?[1].text else {
            return
        }
        // создаем новый контакт
        let contact = Contact(title: contactName, phone: contactPhone)
        self.contacts.append(contact)
        self.tableView.reloadData()
    }

    // кнопка отмены
    let cancelButton = UIAlertAction(title: "Отменить", style: .cancel,
handler: nil)

    // добавляем кнопки в Alert Controller
    alertController.addAction(cancelButton)
    alertController.addAction(createButton)

    // отображаем Alert Controller
    self.present(alertController, animated: true, completion: nil)
}

```

- Свяжите аулет-свойство **tableView** с графическим элементом Table View, размещенным на сцене.

Разберем метод **showNewContactAlert()**.

В первую очередь создается экземпляр типа **UIAlertController**, который после проведения соответствующих настроек и будет использоваться в качестве всплывающего окна для создания контактов. Вы уже знакомы с этим типом данных по первой книге, так что не будем уделять ему внимание.

Метод **addTextField** предназначен для того, чтобы добавить во всплывающее окно текстовое поле. В качестве аргумента ему передается замыкание, конфигурирующее добавляемое текстовое поле. На этом моменте мы и остановимся более подробно.

Для работы с текстовыми полями в Swift предназначен класс **UITextField**. В его состав входит множество различных свойств и методов, позволяющих настраивать этот графический элемент. Так, например, свойство **placeholder**, используемое в нашем коде, изменяет плейсхолдер текстового поля. Таким образом, все изменения, которые вам необходимо произвести с текстовым полем, необходимо выполнять внутри переданного замыкания.

Примечание Плейсхолдер (placeholder) – это заполнитель текстового поля, отображаемый в том случае, когда в нем отсутствует какой-либо текст. Placeholder отображается светло-серым цветом, что воспринимается не как текстовое наполнение, а как подсказка. Используя его, разработчик может подсказать пользователю предназначение поля.

Возможно у вас остался вопрос, почему с помощью переданного замыкания удастся сконфигурировать добавляемое текстовое поле? Как вообще это работает?

В первой книге мы очень подробно рассматривали, чем отличаются классы от структур, и как работает ARC. Класс – это ссылочный тип данных, все экземпляры класса всегда передаются по ссылке. Замыкание как раз и использует эту особенность, настраивая переданное ей значение. Таким образом, все изменения значения, произведенные внутри замыкания, отобразятся и на внешнем значении, созданном вне замыкания (так как по сути это одно и то же значение). Для лучшего понимания ниже я привожу максимально упрощенный аналог метода **addTextField**.

```
func addTextField(_ configurationHandler: ((UITextField) -> Void)? ) {  
    // ...  
    let textField = UITextField()  
    configurationHandler?(textField)  
    self.textFields.append(textField)  
    // ...  
}
```

Внутри метода создается новый экземпляр текстового поля, после чего он передается в замыкание, где и конфигурируется требуемым образом. Получается, что при вызове метода **addTextField**, передавая замыкание, мы можем определить, как именно настроить этот графический элемент. А дальше, при отображении всплывающего окна на сцене, созданные текстовые поля будут также отображены.

После создания текстовых полей в окно добавляются кнопки создания новой записи и отмены. При создании новой записи с помощью свойства **textFields** получаются введенные пользователем значения. Далее они добавляются в массив контактов, после чего вызывается метод **reloadData()**.

Для того, чтобы метод **showNewContactAlert** был вызван по нажатию кнопки, необходимо связать его с этой кнопкой.

- ▶ Откройте файл **Main.storyboard**.
- ▶ Выделите кнопку «Создать контакт» в составе **Toolbar**. Для удобства вы можете использовать **Document Outline**.
- ▶ Откройте **Connections Inspector**.
- ▶ В разделе **Sent Events** перетяните серый круг (напротив поля **selector**) на значок **View Controller** на сцене.
- ▶ В выпадающем меню выберите **showNewContactAlert**.
- ▶ Запустите приложение и создайте несколько контактов.

Обратите внимание, что новые контакты всегда добавляются в конец списка, а не занимают место в соответствии с алфавитом.

- Добавьте наблюдатель к свойству **contacts**, сортирующий массив контактов при каждом его изменении (листинг 7.19).

ЛИСТИНГ 7.19

```
var contacts: [ContactProtocol] = [] {  
    didSet {  
        contacts.sort{ $0.title < $1.title }  
    }  
}
```

- Уберите из метода **loadContacts** вызов метода **sort**. Теперь в нем нет необходимости.

Теперь вы имеете полный контроль над списком контактов! Конечно, было бы идеально произвести несколько доработок, к примеру, прикрутив функциональность проверки корректности введенного номера телефона. Но всем этим вы займетесь при решении домашних заданий.

На текущий момент единственным существенным минусом приложения является то, что после каждой перезагрузки данные принимают первоначальное значение, но с этим мы разберемся в следующей главе.

Глава 8.

Долговременное хранение данных. User Defaults.

В этой главе вы:

- познакомитесь с доступными разработчику вариантами долговременного хранения данных доступны при разработке на iOS;
- узнаете, что такое User Defaults и как использовать его для хранения данных.

Практически любая программа, которую вы можете найти в App Store, пользуется возможностью долговременного хранения данных. Такие данные остаются доступными даже после перезагрузки, а иногда и переустановки приложения. Примеров подобных данных очень много: журнал звонков, статистика игр команды, настройки приложения, данные о лекарствах, личные данные пользователя и т.д.

В текущей версии приложения «**Contacts**» все изменения, внесенные в список контактов, отменяются после выхода из приложения. Каждый раз при новом входе в приложение мы видим все тот же тестовый набор данных. Это происходит по той причине, что данные хранятся в оперативной памяти (в переменных и константах), но не записываются в долговременные хранилища, а значит удаляются при завершении работы приложения.

В этой главе мы начнем изучать вопрос долговременного хранения, рассмотрим доступные элементы, реализующие эту задачу, а также реализуем сохранение данных в приложении «**Contacts**».

8.1 Варианты долговременного хранения данных

Операционная система iOS уделяет особое внимание вопросу долговременного хранения данных. При разработке приложений у вас есть целое множество доступных «из коробки» механизмов для решения этой задачи. Рассмотрим основные из них.

1. Property list – список настроек.

Это файл с расширением plist, содержащий список настроек, хранящий данные в виде элементов «ключ-значение». В любом вашем проекте изначально уже есть один такой файл – **Info.plist** (вы можете найти его в **Project Navigator**, рис. 8.1). С точки зрения внутренней структуры plist-файл – это текстовый файл, отформатированный по стандарту XML.

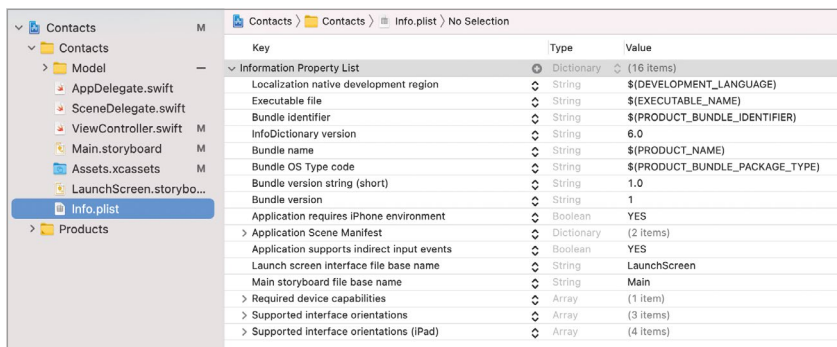


Рис. 8.1. Файл Info.plist в структуре проекта

Plist-файлы обычно используются для хранения небольших объемов данных. В частности, **Info.plist** содержит настройки приложения, используемые при его установке и запуске.

При необходимости вы можете создать произвольное количество plist-файлов и с помощью удобного интерфейса прямо в Xcode записывать в них необходимые данные различных типов. Так, к примеру, в одном из своих приложений вы бы могли создать файл **DemoData.plist** и хранить в нем демонстрационный набор данных, доступных пользователю сразу после установки приложения. Записав в него данные вам бы осталось лишь подгрузить их в процессе запуска приложения, привести к необходимому виду и использовать по назначению.

Обычно plist-файлы используются для хранения неизменяемых данных, то есть таких данных, которые меняются разработчиком только в процессе создания приложения. Примером таких данных может служить адрес удаленного сервера, с которым работает приложение. Для изменения сервера мы просто выпускаем новую версию приложения с обновленными данными в plist-файле.

Очень удобной возможностью plist-файлов является их распределение по targets внутри проекта.

Target – это, грубо говоря, набор настроек для компиляции проекта. При компиляции вы указываете, в соответствии с каким target производить компиляцию. Пока что во всех наших проектах был всего один target, поэтому мы не обращались к этому понятию.

При необходимости вы можете, например, создать дополнительный target для работы с тестовым сервером и создать два одинаковых plist-файла, в одном указать адрес рабочего сервера и привязать его к основному target, а в другом – адрес тестового сервера и привязать его к дополнительному target. И теперь, в зависимости от того, какой target выбран при компиляции, итоговое приложение будет обращаться либо к тестовому, либо к рабочему серверу.

2. User Defaults – интерфейс доступа к пользовательским данным.

Это специальный интерфейс, обеспечивающий хранение изменяемых данных в специальном пользовательском хранилище. Если взглянуть глубже, само пользовательское хранилище представляет из себя уже рассмотренные ранее plist-файлы, но в данном случае вы не видите их в составе проекта, а обращаетесь к ним в коде через специальный класс **UserDefaults**. Использование User Defaults имеет несколько особенностей.

- Класс **UserDefaults**, через который происходит работа, имеет очень удобный API. Например, для сохранения значения достаточно просто выполнить следующий код:

```
UserDefaults.standard.set("Some value", forKey: "Some Key")
```

а для чтения

```
UserDefaults.standard.object(forKey: "Some Key")
```

- User Defaults поддерживает кеширование, а это несколько ускоряет работу приложения.
- User Defaults позволяет хранить числовые, логические, строковые значения, массивы, даты и другие виды данных, но не умеет работать с опционалами.

Для хранения записей в приложении **Contacts** мы будем использовать именно User Defaults.

3. File Manager – файловый менеджер.

Это интерфейс, позволяющий работать с файловой структурой приложения. С помощью File Manager вы можете просматривать, создавать, удалять, изменять, перемещать и искать файлы в пределах вашего приложения. К примеру, с его помощью вы можете сохранить фотографии, полученные из сети для их последующего использования.

4. SQLite – система управления базами данных (СУБД).

Это легковесная подсистема, обеспечивающая хранение данных в реляционных базах данных. Отличительной особенностью SQLite является чрезвычайно высокая скорость работы.

Если у вас уже есть опыт программирования на других языках, то возможно ранее вы работали с SQLite или другой подобной СУБД, работающей посредством SQL-запросов (например, MySQL или PostgreSQL).

5. Core Data – фреймворк.

Это специальный фреймворк, предоставляющий удобный интерфейс для долговременного хранения данных. В базовом варианте Core Data хранит данные с помощью описанного ранее SQLite, но при этом предоставляет больше возможностей, включая облачную синхронизацию данных на разных устройствах с помощью iCloud.

Несмотря на то, что по умолчанию Core Data использует SQLite-базы, вы можете настроить его на работу с другими типами хранилищ: Binary, In-Memory и даже XML.

Зачастую разработчики избегают работы с Core Data, так как это не самая простая задача. Но хорошо разобравшись с этим фреймворком вы сможете с высоким уровнем удобства начать использовать его.

6. Keychain – шифрованное хранилище.

Это специализированное хранилище, предназначенное для хранения конфиденциальной информации, такой как пароли, хэши, секретные фразы и т.д.

7. Сторонние сервисы, вроде Firebase и Realm.

Это не просто сервисы хранения данных, а так называемые mobile-backend-as-a-service (MBaaS), предоставляющие множество возможностей, включая ведение баз пользователей, их аутентификацию, синхронизацию данных и многое другое. Умение работать с такими сервисами очень ценится работодателями.

Примечание Еще одним вариантом хранения данных можно было бы назвать файлы с исходным кодом, примерно так, как сейчас хранятся тестовые записи контактов в классе **ViewController**. А что, записали в коде массив на 1000 элементов – и чем это не хранилище?

Мы с вами не зря касаемся вопросов создания архитектуры приложения. В качественно построенном приложении каждый элемент должен выполнять строго определенную задачу.

Какое предназначение у файла с исходным кодом?

Хранить исходный код. Но не хранить данные.

Именно по этой причине я не выделяю хранение данных в коде в качестве варианта долговременного хранилища. А то, что в текущей реализации класса **ViewController** мы все же храним данные в коде – это лишь временное решение, от которого мы избавимся в скором времени.

Каждый из описанных механизмов требует от вас времени и сил на его изучение, так как правильное и глубокое понимание темы позволит вам верно выбирать наиболее подходящий вариант хранилища в зависимости от ситуации. В этой главе мы рассмотрим только User Defaults.

8.2 User Defaults

User Defaults обеспечивает долговременное хранение дат, числовых, логических, строковых и других типов данных. Когда вы сохраняете данные с помощью User Defaults, то всегда можете быть уверены, что они останутся доступными при следующем запуске приложения.

Примечание Удаление приложения из устройства приводит к уничтожению всех данных в User Defaults.

Интерфейс, который предоставляет User Defaults для работы с данными, очень удобен. Со временем у вас может возникнуть желание хранить там все больше и больше различной информации. Однако вам следует понимать, что это очень плохая идея, так как в конечном счете это приведет к увеличению времени загрузки приложения. User Defaults хорош для небольших объемов, но при увеличении размера стоит рассмотреть другие доступные механизмы долговременного хранения. Например, не стоит хранить там большие текстовые заметки, объекты Модели (если вы используете MVC) или статьи.

Основное предназначение User Defaults – хранение различных параметров пользователя (имя, почтовый адрес, возраст, пол), настроек приложения (выбранный язык, валюта, цветовая схема) и различные флаги, содержащие логические значения («была ли показана инструкция при входе», «были ли синхронизированы данные с сервером»).

User Defaults загружает все сохраненные данные при первом обращении к нему в процессе работы приложения, и чем больший объем данных хранится в нем, тем больше времени займет загрузка. При этом все последующие операции чтения чрезвычайно быстры.

Стоп! Если User Defaults предназначен для хранения различных настроек и флагов, то почему мы собираемся записывать туда данные из Модели? Да, это в некотором роде нарушение и так лучше не делать в ваших последующих проектах. Но в нашем случае:

1. вы пока еще не умеете работать вообще с какими-либо хранилищами;
2. все наши проекты еще не настолько сложные, чтобы у них появились какие-либо настройки;
3. модель приложения «**Contacts**» максимально проста, и это отличный способ продемонстрировать возможности User Defaults.

В своей карьере разработчика для хранения данных Модели старайтесь использовать другие типы хранилищ (например, Core Data).

Хранение данных

User Defaults хранит данные в plist-файлах. И, как в случае с Property list, данные представляют из себя пары «ключ-значение», примерно так, как вы видели это при работе со словарями:

```
var myDict = [  
    "isAdmin": "true",  
    "age": "39",  
    "mail": "mail@swiftme.ru"]
```

В левой части каждого элемента словаря находится ключ, а в правой – значение. Для того, чтобы получить или изменить данные словаря, вам необходимо обратиться к ним используя ключ:

```
myDict["isAdmin"] // true  
myDict["isAdmin"] = "false"
```

Примерно так же происходит работа и с User Defaults.

Доступ к User Defaults

Для работы с User Defaults используется класс **UserDefaults**, который работает на основе паттерна «Одиночка» (Singleton). Для того, чтобы получить экземпляр класса, необходимо выполнить следующий код:

```
UserDefaults.standard
```

В результате вам вернется объект, который запишет данные в стандартное хранилище.

Примечание У вас есть возможность создавать собственные хранилища в User Defaults, но на данном этапе это будет совершенно лишняя информация. Пока что старайтесь работать именно со стандартным хранилищем.

Где бы в коде вы не обратились к **UserDefaults.standard**, вы всегда получите один и тот же экземпляр, а значит в результате будете работать с одними и теми же данными. Например, вы можете сохранить имя пользователя на экране логина, а на экране редактирования профиля загрузить и изменить его.

- В классе **ViewController** создайте новое свойство **userDefaults** в соответствии с листингом 8.1

ЛИСТИНГ 8.1

```
var userDefaults = UserDefaults.standard
```

Теперь внутри класса **ViewController** нам не придется использовать длинный синтаксис **UserDefaults.standard**. Вместо этого будем обращаться к свойству **userDefaults**.

Запись данных

Для записи данных в User Defaults используется метод `set`. В качестве первого аргумента передается значение, а второго – ключ. Если вызвать метод `set` свойства `userDefaults`, в окне автодополнения вы увидите несколько его вариантов (рис. 8.2).

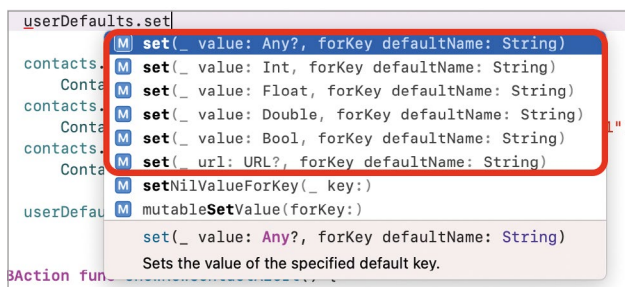


Рис. 8.2. Автодополнение для метода `set`

Примечание Не путайте метод `set(_:forKey:)` с методом `setValue(_:forKey:)`. Второй наследуется от класса `NSObject` и не относится к `UserDefaults`. Для записи значений используйте исключительно `set`.

По всплывающему окну автодополнения видно, что в User Defaults могут быть сохранены значения типов `Int`, `Float`, `Double`, `Bool`, `URL` и `Any?`. С первыми пятью все понятно, но что значит `Any?`, и какие значения могут быть переданы в этом случае?

Ключевое слово `Any` (с ним мы уже знакомы по материалу первой книги) определяет любой тип данных. Другими словами, вместо `Any` можно передать совершенно любое значение. Но попробуйте написать в методе `viewDidLoad` следующий код

```
userDefaults.set(self, forKey: "View Controller current state")
```

и получите критическую ошибку во время исполнения программы. При этом, пока вы пишете этот код. Среда разработки будет молчать, так как формально `self` имеет тип данных `UIViewController` и соответствует требованиям `Any`.

Дело в том, что несмотря на использование `Any` в качестве типа аргумента, все же есть ограничения на то, какие значения могут быть переданы. Помимо указанных выше `Int`, `Float`, `Double`, `Bool` и `URL`, вы можете сохранить значения типов `Array`, `Dictionary`, `String`, `Date` (дата) и `Data` (двоичные данные). Если вам требуется записать значение другого типа, его предварительно необходимо привести к одному из указанных выше.

Примечание Хранение в User Defaults значений кастомных типов, например, `Currency`, `Person` и т.д. также возможно, однако это требует определенных навыков работы с протоколами `NSCoding` и `Codable`.

Попробуем записать в User Defaults несколько значений.

- Добавьте в метод **viewDidLoad** код из листинга 8.2.

ЛИСТИНГ 8.2

```
userDefaults.set("Some random text", forKey: "Some key")
```

- Запустите приложение на симуляторе.

В процессе запуска приложения, когда будет вызван метод **viewDidLoad**, в User Defaults будет произведена запись, которая останется там, пока вы не удалите или не измените ее (или пока не удалите приложение целиком).

Примечание В некоторых руководствах вы можете встретить указание на необходимость использовать метод **synchronize** после изменения данных в User Defaults. На самом деле, Apple уже несколько лет рекомендует не делать этого.

Чтение данных

Для получения данных, ранее записанных в User Defaults, используется специальный метод **object(forKey:)**, например:

```
UserDefaults.standard.object(forKey: "Some key")
```

Данный метод вернет:

- значение типа **Any?**, если запрошенный ключ **forKey** существует;
- **nil**, если запрошенный ключ **forKey** не существует.

Примечание Обратите внимание, что класс **UserDefaults** также позволяет вам вызвать метод **value(forKey:)**, который с первого взгляда очень похож на **object(forKey:)**. Данный метод наследуется от протокола **NSObject**, не имеющего никакого отношения к **User Defaults**!

Не используйте **value(forKey:)** для получения значений, используйте **object(forKey:)** или один из его аналогов.

После того, как метод **object** вернет значение типа **Any?**, вам необходимо привести его к типу данных, соответствующему этому значению. Например, если в User Defaults было записано строковое значение

```
UserDefaults.standard.set("Some random text", forKey: "Some key")
```

то при его получении потребуются провести приведение к строковому типу (тайпкастинг)

```
UserDefaults.standard.object(forKey: "Some key") as? String,
```

или сразу же безопасно извлечь значение из опционала

```
UserDefaults.standard.object(forKey: "Some key") as? String ?? ""
```

Помимо **object**, в состав класса **UserDefaults** входят несколько вспомогательных методов, позволяющих сразу же получить значение требуемого типа данных (таблица 8.1).

Таблица 8.1. Методы класса UserDefaults

Название метода	Возвращает, если ключ существует, и полученное значение может быть приведено к требуемого типу	Возвращает, если ключ не существует, или существует, но полученное значение не может быть приведено к требуемому типу
<code>array(forKey:)</code>	<code>Array<Any>?</code>	<code>nil</code>
<code>bool(forKey:)</code>	<code>true</code>	<code>false</code>
<code>data(forKey:)</code>	<code>Data?</code>	<code>nil</code>
<code>dictionary(forKey:)</code>	<code>Dictionary<String, Any>?</code>	<code>nil</code>
<code>double(forKey:)</code>	<code>Double</code>	<code>0</code>
<code>float(forKey:)</code>	<code>Float</code>	<code>0</code>
<code>integer(forKey:)</code>	<code>Int</code>	<code>0</code>
<code>string(forKey:)</code>	<code>String?</code>	<code>nil</code>
<code>stringArray(forKey:)</code>	<code>[String]?</code>	<code>nil</code>
<code>url(forKey:)</code>	<code>URL?</code>	<code>nil</code>

Все перечисленные методы по сути являются оберткой над **object(forKey:)**, просто они дополнительно проводят тайпкастинг. То есть, вы всегда можете использовать метод **object**, но для удобства можете использовать один из указанных методов.

При загрузке данных из User Defaults вы должны знать, значение какого типа вы получаете.

Попробуем получить ранее записанное в методе **viewDidLoad** значение.

► Удалите вызов метода **set** в методе **viewDidLoad**.

Несмотря на то, что вызов **set** был удален, текстовое значение по ключу «**Some key**» было сохранено при предыдущем запуске, а значит продолжит быть доступным. Помните, что User Defaults обеспечивает долговременное хранение данных.

► Добавьте в метод **viewDidLoad** код из листинга 8.3.

ЛИСТИНГ 8.3

```
print( userDefaults.object(forKey: "Some key") )  
print( userDefaults.string(forKey: "Some key") )
```

- ▶ Произведите запуск приложения.

После того, как приложение будет загружено, на консоли будет отображено два сообщения. Первое указывает на тип данных **Optional<Any>**, а второе – **Optional<String>**. Это говорит о том, что хоть мы и не записывали данные в User Defaults в текущей сессии работы приложения, они все равно остались там. Примерно таким образом вы можете организовывать долговременное хранение требуемых данных в любой программе, которую разрабатываете. User Defaults прост в использовании и практически бесплатен (в отношении ресурсов) для операционной системы (при условии хранения малых объемов данных).

Доступ к plist-файлу User Defaults

В процессе разработки вам может потребоваться визуально оценить, что в данный момент содержится в User Defaults. Возможно, это потребуется в целях отладки или оптимизации, но факт остается фактом – периодически вам нужно открывать plist-файл, в который записываются данные.

Очень важно отметить, что это доступно только при запуске приложения на реальном устройстве, не на симуляторе.

- ▶ Добавьте удаленный ранее вызов метода **set** в метод **viewDidLoad**:

```
userDefaults.set("Some random text", forKey: "Some key")
```

- ▶ Произведите запуск приложения на реальном устройстве.

Примечание Для запуска приложения на реальном устройстве необходимо подключить его к компьютеру с помощью кабеля (в случае, если вы не настраивали беспроводное подключение ранее). Далее в списке симуляторов выбрать ваше устройство и нажать кнопку запуска проекта.

- ▶ В главном меню Xcode выберите пункт **Window > Devices and Simulators**.

Перед вами появится окно, отображающее список всех подключенных мобильных устройств, а также установленных симуляторов. Теперь необходимо скачать пакет с данными установленного приложения (рис. 8.3).

- ▶ (1) В разделе **Devices** выберите свой iPhone, на котором запущено приложение.
- ▶ (2) В разделе **Installed Apps** щелкните по приложению «**Contacts**».
- ▶ (3) Ниже списка приложений щелкните по иконке «**Шестеренка**» и в выпадающем окне выберите пункт «**Download Container**».

- Сохраните контейнер в произвольном месте, но там, где вы сможете его найти. Сохраненный пакет данных представляет из себя файл с расширением **xcappdata**.

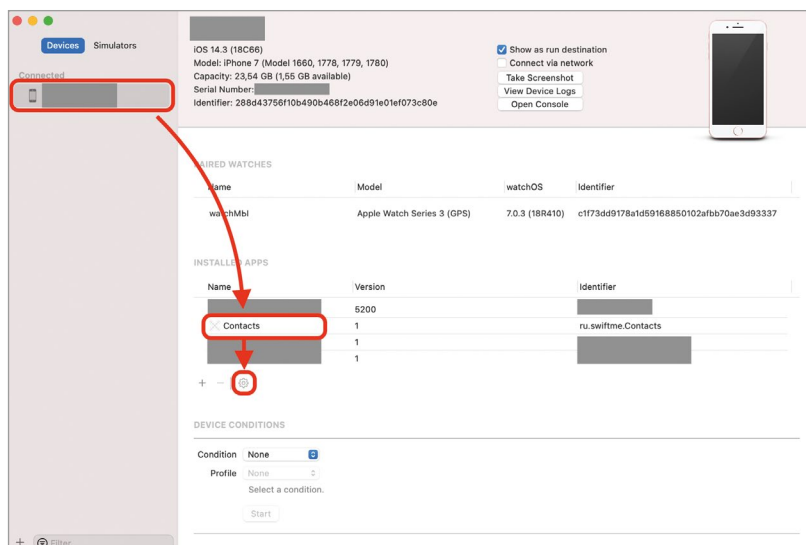


Рис. 8.3. Окно Devices and Simulators

- В Finder щелкните правой кнопкой мыши по файлу и выберите пункт «**Показать содержимое пакета**».
- Далее перейдите по пути **AppData/Library/Preferences**.

Файл с расширением plist, который хранится в папке **Preferences** – это и есть то самое хранилище пользовательских данных, доступ к которому мы получали посредством User Defaults (рис. 8.4). При открытии вы увидите в нем всего одно сохраненное значение «**Some key**» с текстовым значением «**Some random text**» (рис. 8.5).

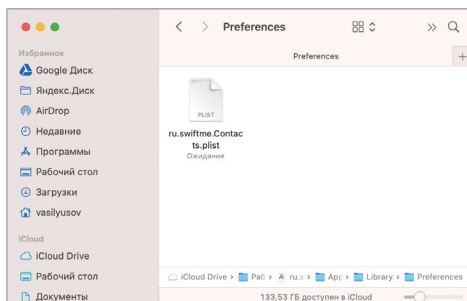


Рис. 8.4. Файл с данными



Рис. 8.5. Содержимое файла

Вот таким несложным способом вы в любой момент сможете проверить состав и структуру данных, хранящихся в текущий момент в User Defaults.

► Удалите метод **set** из **viewDidLoad**.

8.3 Хранение контактов в User Defaults

Теперь перейдем к организации долговременного хранения данных в нашем приложении. Для решения этой задачи нам потребуется немного доработать Модель и View Controller.

Во-первых, выделим такую сущность, как «Хранилище данных о контактах». Она будет составной частью Модели, а в ее задачи будет входить решение вопроса сохранения и загрузки списка контактов.

Во-вторых, View Controller будет иметь ссылку на хранилище и при необходимости обновления данных будет обращаться к нему, вызывая соответствующие методы.

В-третьих, вся работа внутри View Controller будет основана на свойстве **contacts**, в котором в любой момент времени (после первоначальной загрузки) будет храниться актуальный список контактов. С помощью наблюдателя **didSet** (так, как это сделано сейчас для сортировки массива) реализуем сохранение данных в «Хранилище» при любом изменении значения этого свойства (при создании и удалении элементов). Такой подход позволит нам не дорабатывать какие-либо другие элементы View Controller и выделить единую точку взаимодействия с «Хранилищем».

► Откройте файл **Contact.swift**.

► Добавьте в файл код из листинга 8.4

ЛИСТИНГ 8.4

```
protocol ContactStorageProtocol {  
    // Загрузка списка контактов
```

```
func load() -> [ContactProtocol]
// Обновление списка контактов
func save(contacts: [ContactProtocol])
}

class ContactStorage: ContactStorageProtocol {
    // Ссылка на хранилище
    private var storage = UserDefaults.standard
    // Ключ, по которому будет происходить сохранение хранилища
    private var storageKey = "contacts"

    // Перечисление с ключами для записи в User Defaults
    private enum ContactKey: String {
        case title
        case phone
    }

    func load() -> [ContactProtocol] {
        var resultContacts: [ContactProtocol] = []
        let contactsFromStorage = storage.array(forKey: storageKey) as?
[[String:String]] ?? []
        for contact in contactsFromStorage {
            guard let title = contact[ContactKey.title.rawValue],
                  let phone = contact[ContactKey.phone.rawValue] else {
                continue
            }
            resultContacts.append(Contact(title: title, phone: phone))
        }
        return resultContacts
    }

    func save(contacts: [ContactProtocol]) {
        var arrayForStorage: [[String:String]] = []
        contacts.forEach { contact in
            var newElementForStorage: Dictionary<String, String> = [:]
            newElementForStorage[ContactKey.title.rawValue] = contact.title
            newElementForStorage[ContactKey.phone.rawValue] = contact.phone
            arrayForStorage.append(newElementForStorage)
        }
        storage.set(arrayForStorage, forKey: storageKey)
    }
}
```

Реализацию сущности «Хранилище» мы начинаем, как и всегда, с создания протокола. Для работы с хранилищем нам потребуется всего два метода: один для загрузки, второй для сохранения данных.

Как вы думаете, какая практическая ценность от протокола в данном случае?

Вот представьте, прошло время, и вы решили доработать свою программу, а в качестве долговременного хранилища использовать Core Data. В этом случае вам потребуется создать новый класс, скажем **ContactStorageCoreData**, и интегрировать его во **View Controller**.

Использование протокола позволит вам быть уверенным, что новый класс будет иметь тот же самый набор доступных методов, а это значительно упростит интеграцию.

Более того, **View Controller** будет связан с «Хранилищем» также на основе протокола. Таким образом, создавая новый класс мы просто подпишем его под протокол **ContactStorageProtocol** и с легкостью подставим вместо **ContactStorage** во вью контроллер.

Протоколы (а также наследование) – это основа полиморфизма, одной из особенностей объектно-ориентированного программирования. Благодаря полиморфизму вы можете подменять одни типы и объекты другими, не вызывая никаких сбоев со стороны программного обеспечения.

Обратите внимание на реализацию методов **load** и **save**. User Defaults позволяет хранить в себе значения конкретного перечня типов, которые мы рассматривали ранее. Иначе говоря, мы не можем просто взять и сохранить значение типа **Contacts**! Прежде, чем сделать это, нам потребуется привести его к такому типу данных, который может быть записан в User Defaults.

В нашем случае в методе **save** мы сперва приводим его к допустимому типу **[[String:String]]**, то есть к массиву словарей, а уже потом производим операцию записи.

В методе **load** происходит обратная операция — загруженное значение типа **[[String:String]]** преобразуется к массиву контактов **[Contacts]**.

При этом в качестве ключей используется перечисление **ContactKey** — такой подход исключит случайную ошибку в названии ключа.

Теперь внесем правки в класс **ViewController**, связав его с «Хранилищем» и обеспечив загрузку и сохранение данных.

- ▶ Удалите свойство **userDefaults**.
- ▶ Создайте свойство **storage**, которое будет держать в себе ссылку на «Хранилище» (листинг 8.5).

ЛИСТИНГ 8.5

```
var storage: ContactStorageProtocol!
```

- Дополните наблюдатель свойства **contacts** в соответствии с листингом 8.6.

ЛИСТИНГ 8.6

```
var contacts: [ContactProtocol] = [] {  
    didSet {  
        contacts.sort{ $0.title < $1.title }  
        // сохранение контактов в хранилище  
        storage.save(contacts: contacts)  
    }  
}
```

Теперь при каждом изменении значения свойства **contacts** массив контактов будет незамедлительно передан в «Хранилище» для записи в User Defaults.

- Дополните метод **viewDidLoad** в соответствии с листингом 8.7.

ЛИСТИНГ 8.7

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    storage = ContactStorage()  
    loadContacts()  
}
```

В процессе загрузки сцены будет произведено создание экземпляра типа **ContactStorage** и его инициализация свойству **storage**. С этого момента мы можем загружать данные из User Defaults.

- Измените тело метода **loadContacts**, убрав оттуда тестовый набор данных и обеспечив загрузку данных из хранилища (листинг 8.8).

ЛИСТИНГ 8.8

```
private func loadContacts() {  
    contacts = storage.load()  
}
```

Это все правки, который нам требуется внести во view контроллер для организации его работы с хранилищем контактов.

- Протестируйте работу приложения, несколько раз запустив его. При этом создавайте и удаляйте контакты в произвольном порядке.

При каждом новом запуске список контактов в таблице будет в точности соответствовать тому, каким он был при предыдущем запуске.

Примечание При работе на симуляторе данные User Defaults могут сохраняться с небольшими задержками. Так, например, иногда, создав несколько записей и тут же перезапустив приложение, вы можете не увидеть последний созданный контакт. Но при этом, если выполните ту же операцию, но подождете несколько секунд, данные будут сохранены.

При работе на реальном устройстве этой проблемы не будет.

Вот так, без особого труда вы можете добавить долговременное хранение данных в любое свое приложение. В дальнейшем вы познакомитесь и с другими механизмами, а затем уже на основе имеющегося опыта сможете выбирать между ними.

8.4 Распределение элементов проекта по папкам

Последний вопрос, о котором я хотел бы поговорить в данной главе, не имеет никакого отношения к долговременному или кратковременному хранению данных. Тем не менее, я считаю, что он достаточно важен для того, чтобы обсудить его с вами. Я хотел бы рассказать о распределении файлов в составе проекта и о том, где хранить ваши файлы с исходным кодом, ресурсы, plist-файлы и т.д.

В первую очередь, стоит понять, что используемая архитектура (в нашем случае это MVC) определяет, какие сущности должны существовать в проекте, но она не говорит о том, как эти сущности хранить в структуре проекта. Несмотря на это для хранения модели мы создавали папку с именем **Model**, а для представлений — **View**. Но даже, если вы разместите файлы иначе, проект все равно будет создан с учетом принципов MVC. Тем не менее, отражение некоторых черт и структуры используемого паттерна в структуре файлов — это хорошая идея, которая упрощает работу с проектом.

Вы можете размещать папки и файлы так, как вам захочется, но в общем случае можно следовать следующей схеме (если конечно вы используете MVC):

/Корневая папка проекта

 /AppDelegate.swift

 /SceneDelegate.swift

/Info.plist

/View — папка для хранения всех представлений в составе проекта.

 /Storyboards — папка для хранения сторибордов.

 /Cells — папка для хранения кастомных ячеек (с ними познакомимся в следующей части).

 /Xibs — папка для хранения xib-файлов (с ними познакомимся в следующей части).

/... — другие папки, определяющие типы используемых в проекте представлений.

/Model — папка, содержащая Модель.

/Storage — папка, содержащая типы, обеспечивающие доступ к хранилищам. Несмотря на то, что это часть Модели, я стараюсь хранить их отдельно.

/Network — папка, содержащая типы, обеспечивающие доступ к сети. Ситуация та же самая, что и со Storage.

/Controller — папка, содержащая Контроллер (классы выю контроллеров).

/Helpers — папка, содержащая файлы с исходным кодом с различными вспомогательными функциями (например, реализованная вами функция перевода градусов в радианы).

/Resources — папка для хранения различных ресурсов, вроде ассетов, картинок и т.д.

Это лишь один из вариантов. Со временем вы сами определите удобную для вас структуру, более того, она будет изменяться от проекта к проекту. Так, например, в одном из своих приложений я храню класс контроллера и соответствующий ему storyboard или xib в одной папке, то есть совмещаю View и Controller.

Самое важное, чтобы реализованная структура была удобна для вас и позволяла быстро искать требуемые ресурсы.

ИТОГИ ВТОРОЙ ЧАСТИ КНИГИ

Вторая часть книги была посвящена знакомству с возможностями iOS по отображению данных в виде таблиц, а также долговременному хранению информации. Полученные знания будут использоваться вами, в том числе и в следующей части, где мы продолжим рассмотрение табличных представлений и разработаем приложение «**To-Do Manager**», позволяющее вести учет планируемых и выполненных дел.

Часть III

ПРОДВИНУТЫЕ ТАБЛИЧНЫЕ ПРЕДСТАВЛЕНИЯ

ПРОЕКТ «TO-DO MANAGER»

Процесс обучения разработке под iOS, на самом деле, не имеет конца. Это связано не только с регулярным выходом новых версий Swift, Xcode и операционных систем, но и с тем, что профессиональный разработчик никогда не должен прекращать свое обучение. Для того, чтобы разрабатывать простые приложения на должности Junior Swift Developer, вам, возможно, и хватит материала первых двух книг и пары видео-курсов, но если вы амбициозны и хотите достичь вершин в нашем деле, вам предстоит узнать еще очень многое, начиная от уже упомянутых Swift, Xcode и операционных систем, до сторонних библиотек, computer science, теории программирования, математического анализа, теории алгоритмов и т.д. Каждая изученная тема позволит вам по-новому взглянуть на ваш предыдущий опыт, и даст сильный толчок к дальнейшему развитию.

Но путь к великим знаниям делается небольшими шагами. И в этой главе мы продолжим создавать крепкий фундамент, на котором будет строиться ваша будущая карьера.

В ходе следующих семи глав мы разработаем приложение **«To-Do Manager»**, предназначенное для ведения списка задач. Его интерфейс показан на рисунке 1.

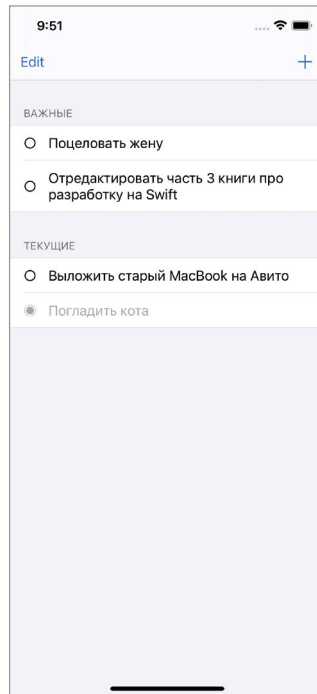


Рис. 1. Интерфейс приложения «To-Do Manager»

Основной задачей этого приложения будет отслеживание ваших текущих и выполненных задач. При этом у задач будет несколько свойств, определяющих их положение в списке. Например, задачи могут быть «важными» и «текущими», «выполненными» и «запланированными».

В ходе изучения материала вы продолжите рассмотрение табличных представлений, познакомитесь с навигационным контроллером, а также научитесь передавать данные между сценами в приложении.

Глава 9. Навигационный контроллер. Класс **UINavigationController**

Глава 10. Передача данных между контроллерами

Глава 11. Контроллер табличного представления. Класс **UITableViewController**

Глава 12. Табличные представления на основе прототипов ячеек

Глава 13. Изменение элементов табличного представления

Глава 14. Создание и изменение задач

Глава 15. Завершение разработки приложения

Глава 9.

Навигационный контроллер.

Класс UINavigationController.

В этой главе вы:

- научитесь работать с навигационным контроллером (Navigation Controller);
- узнаете, что такое навигационный стек (Navigation Stack);
- рассмотрите, какие возможности предоставляет класс UINavigationController;
- научитесь осуществлять навигацию между сценами с помощью навигационного контроллера и навигационного стека;
- попытаетесь изменить визуальное оформление навигационного контроллера.

Если у вас есть iPhone или iPad, ответьте на вопрос, как много приложений, состоящих всего из одной сцены, установлено на них? На ум приходит разве что стандартный «**Калькулятор**». Так и в вашем случае, подавляющее большинство будущих проектов будут включать два и более экранов, а это значит, что необходимость удобной навигации между ними сложно переоценить.

У вас уже есть небольшой опыт в этом вопросе: вы знаете, что такое переходы (segue) (рис. 9.1). Это довольно удобный способ перемещения между сценами приложения.

В этой главе мы познакомимся еще с одним элементом, обеспечивающим навигацию внутри приложения – навигационным контроллером (Navigation Controller), а также связанным с ним навигационным стеком (Navigation Stack).

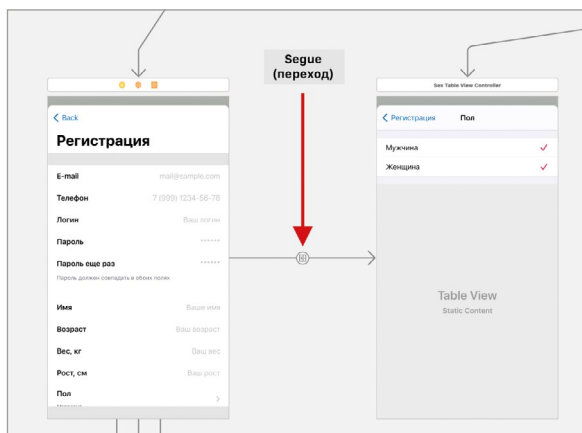


Рис. 9.1. Пример segue на storyboard

9.1 Навигационный контроллер

Навигационный контроллер (Navigation Controller) – это контейнерный View Controller, способный управлять одним или несколькими дочерними вью контроллерами, отображая их интерфейс внутри себя и обеспечивая навигацию между ними.

В общем случае навигационный контроллер состоит из контейнерного представления (Container View) и панели навигации (Navigation Bar). На рисунке 9.2 навигационный контроллер показан схематично (слева) и в интерфейсе приложения «Заметки» (справа).

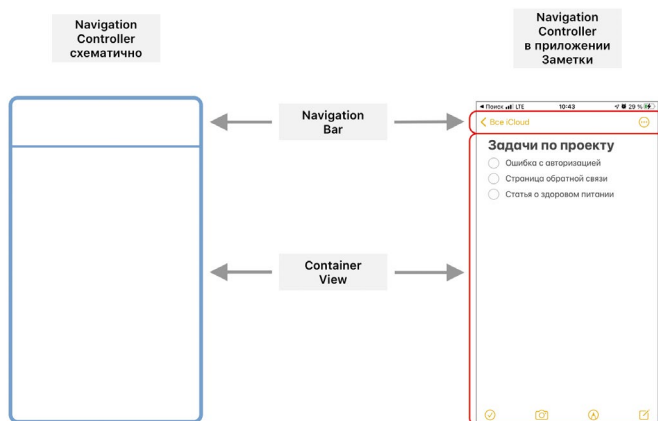


Рис. 9.2. Навигационный контроллер

Панель навигации (Navigation Bar) подстраивается под текущую сцену, отображая ее название, вспомогательные графические элементы (например, кнопки), а также обеспечивает возврат к предыдущей сцене (на рисунке 9.2 это кнопка «**Все iCloud**», предназначенная для возврата к списку заметок).

Контейнерное представление (Container View) – это особое представление, способное отображать внутри себя интерфейс одной из дочерних сцен (одного из дочерних вью контроллеров).

Навигационный контроллер – это своего рода обертка для других контроллеров, способная отображать внутри себя сцены дочерних вью контроллеров. Для Navigation Controller могут быть определены один или несколько дочерних контроллеров (рис. 9.3), и в один момент времени в нем может быть отображена только одна конкретная сцена. При переходе от одной сцены к другой панель навигации изменяется в соответствии с настройками отображаемой сцены, обеспечивая отображение кнопки для возврата, названия сцены, а также вспомогательных элементов.



Рис. 9.3. Пример дочерних контроллеров

- ▶ Откройте приложение «**Заметки**» на своем iPhone или iPad и попробуйте несколько раз произвести переходы между различными экранами (заметками, списками заметок).

Обратите внимание, что при навигации внутри приложения «**Заметки**» (а также любого другого приложения, созданного с использованием навигационного контроллера) верхняя панель подстраивается под текущую сцену, отображая различные элементы (кнопки, название).

Navigation Controller в Swift представлен классом **UINavigationController** (потомок **UIViewController**), который входит в состав фреймворка **UIKit**.

9.2 Создание навигационного контроллера

Посмотрим, как выглядит Navigation Controller в Xcode.

- ▶ Создайте новый проект с названием **NavigationApp**. В качестве шаблона, как и ранее, выберите **App**.
- ▶ Перейдите к файлу **Main.storyboard** и удалите стандартную сцену.
- ▶ Найдите **Navigation Controller** в библиотеке объектов и разместите его на сториборде.

В результате на storyboard появятся две сцены, соединенные между собой стрелкой (рис. 9.4): левая – навигационный контроллер, а правая – его дочерний контроллер, который включает в себя табличное представление.



Рис. 9.4. Navigation Controller и его дочерний View Controller

По вашему мнению, что сейчас будет отображено на экране устройства, если собрать и запустить проект?

- ▶ Выделите **Navigation Controller** на сториборде.
- ▶ Перейдите к **Attributes Inspector** и активируйте пункт **Is Initial View Controller**.

Примечание Если бы мы не проделали указанные действия, проект был бы запущен с черным экраном, так как у него отсутствовала стартовая сцена.

- ▶ Произведите запуск приложения.

В результате вы увидите сцену с пустой таблицей, обернутую в навигационный контроллер (рис. 9.5), в навигационной панели которой будет выведено «**Root View Controller**», что соответствует названию сцены (рис. 9.6).

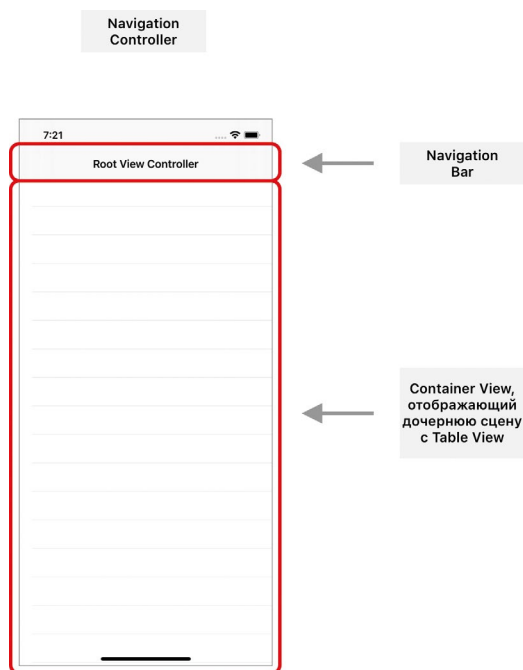


Рис. 9.5. Сцена на основе навигационного контроллера

Навигационный контроллер предоставляет вам полный доступ к сцене, отображаемой в данный момент внутри Container View. Таким образом, работа со сценой, а также связанной с ней классом, практически ничем не отличается от того, что вы делали ранее: создавайте аутлеты и экшены, размещайте графические элементы – все это будет отображено на сцене, но вся сцена при этом целиком будет обернута в UINavigationController.

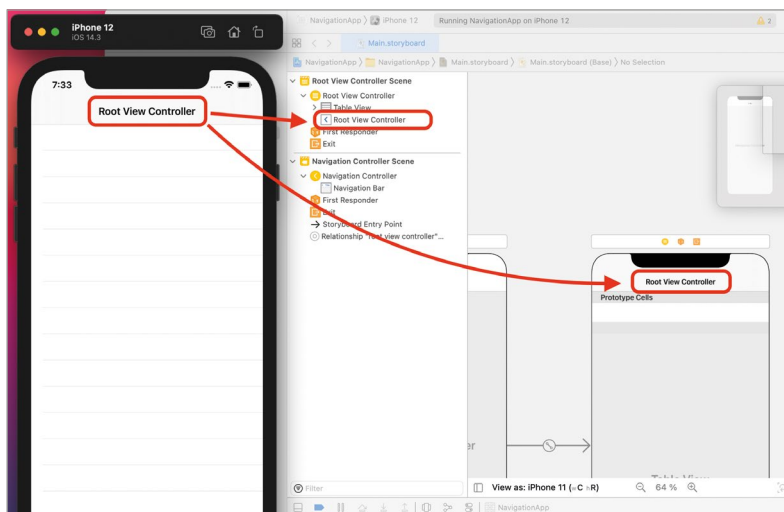


Рис. 9.6. Название сцены в навигационной панели

- Выделите и удалите **Navigation Controller** со сториборда.

Обратите внимание, что вместе с навигационным контроллером со второй сцены (Root View Controller) пропала и навигационная панель. Если теперь установить данную сцену как стартовую, а после этого запустить проект, вы увидите уже знакомое вам табличное представление (рис. 9.7). Сцена больше не будет обернута в Navigation Controller.



Рис. 9.7. Интерфейс приложения без навигационного контроллера

Добавление Navigation Controller к существующей сцене

Не всегда получается досконально продумать структуру будущего приложения и порой требуется встроить навигационный контроллер в уже существующий проект. Xcode позволяет вам сделать это без каких-либо усилий. Добавим Navigation Controller к сцене с табличным представлением:

- ▶ Выделите **Root View Controller** на сториборде.
- ▶ Выберите пункт главного меню **Editor > Embed In > Navigation Controller**.
- ▶ Установите **Navigation Controller** в качестве стартового (пункт **Is Initial View Controller**).

Теперь storyboard вновь содержит два связанных между собой контроллера: навигационный и с табличным представлением.

Описанным способом вы можете с легкостью добавить навигационный контроллер в любой уже существующий проект.

Корневой View Controller

Сцена Root View Controller является корневой для размещенного на сториборде навигационного контроллера. Корневая сцена будет отображена первой внутри Navigation Controller.

1. При запуске приложения формируется экземпляр **UIWindow**.
2. В соответствии с настройками в файле **Info.plist** определяется storyboard-файл, который будет загружен первым в составе приложения (в нашем случае это **Main.storyboard**). Именно из данного файла получается стартовая сцена.
3. Определяется стартовый View Controller (тот, у которого отмечен пункт **Is Initial View Controller**, в нашем случае — это **Navigation Controller**).
4. Для **Navigation Controller** определяется корневая сцена (в нашем случае — это **Root View Controller**).

Стрелка (вернитесь к рис. 9.4), идущая от Navigation Controller к Root View Controller, как раз и определяет, какая сцена является корневой. Каждый Navigation Controller должен иметь корневую сцену, и при необходимости вы можете изменять ее с помощью **Interface Builder** или программного кода.

Примечание В отношении навигационного контроллера обычно говорят не о корневой сцене, а о корневом вью контроллере (root View Controller), который эту сцену обслуживает.

При необходимости вы можете с легкостью изменить корневой View Controller.

- ▶ Добавьте на сториборд пустой вью контроллер.

- ▶ Измените его фоновый цвет на красный.
- ▶ Выделите Navigation Controller.
- ▶ Зажмите клавишу **Control** и перетяните его на красный вью контроллер.
- ▶ В появившемся окне выберите пункт **root view controller** (рис. 9.8).

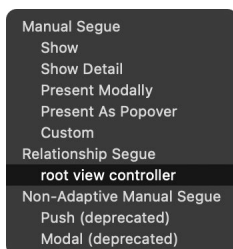


Рис. 9.8. Выбор корневого вью контроллера

После проделанных действий на сториборде отобразится связь между навигационным и красным вью контроллерами, а вот линия к контроллеру с табличным представлением автоматически удалится (рис. 9.9).

Примечание Изменить корневой View Controller можно, в том числе, и с помощью панели **Connection inspector**.

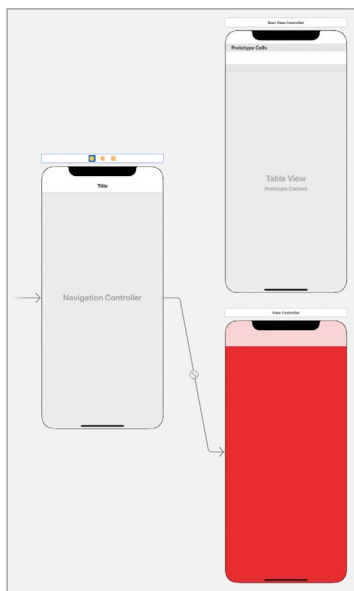


Рис. 9.9. Новый корневой вью контроллер

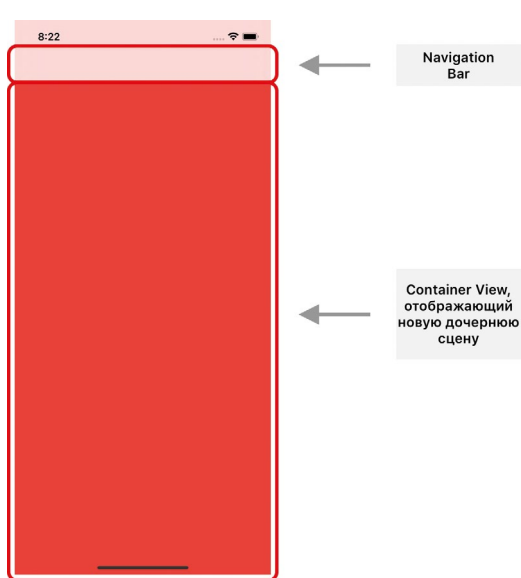


Рис. 9.10. Красная сцена внутри навигационного контроллера

- Произведите запуск приложения.

Теперь вместо таблицы внутри навигационного контроллера будет отображена сцена с красным фоном (рис. 9.10). Навигационная панель также отображается в верхней части экрана, но на этот раз на ней нет названия, но с этим мы разберемся чуть позже.

- Удалите со сториборда неиспользуемый контроллер с табличным представлением.

9.3 Навигационный стек

Все сцены (а точнее вью контроллеры), которые отображаются в навигационном контроллере, помещаются в специальный навигационный стек, являющийся неотъемлемой частью навигационного контроллера. При этом у разработчика есть возможность свободно перемещаться назад по элементам этого стека, отображая необходимые сцены внутри Navigation Controller.

Обратите внимание, что говоря о перемещении по стеку, я использовал слово назад. Это связано с принципом его действия.

Стек – это структура данных, функционирующая по принципу «первый пришел – последний вышел» (First Input – Last Output, FILO). Каждый новый элемент, попадая в стек, помещается в самый верх, и пока он не выйдет из стека, мы не имеем доступа к лежащим ниже элементам. Чтобы добраться до какого-либо элемента в стеке, нам потребуется по одному перебирать и удалять его элементы до тех пор, пока не найдем требуемый.

Примечание Пример работы стека мы рассматривали еще в первой книге в главах о последовательности и управлении памятью в Swift.

В случае с навигационным стеком происходит то же самое: каждая новая сцена, которую отображает Navigation Controller, добавляется в самый верх стека и отображается на экране. При переходе к предыдущей сцене верхний элемент стека удаляется из него, после чего новый верхний выводится на экран.

Чтобы лучше понять, как именно работает навигационный стек, рассмотрим один пример.

- Добавьте на сториборд два новых вью контроллера, разместив их в ряд правее красного контроллера.
- Измените цвет первого добавленного вью контроллера на зеленый, а второго на желтый.

В результате проделанных действий у вас должна получиться картина, похожая на ту, что изображена на рисунке 9.11.

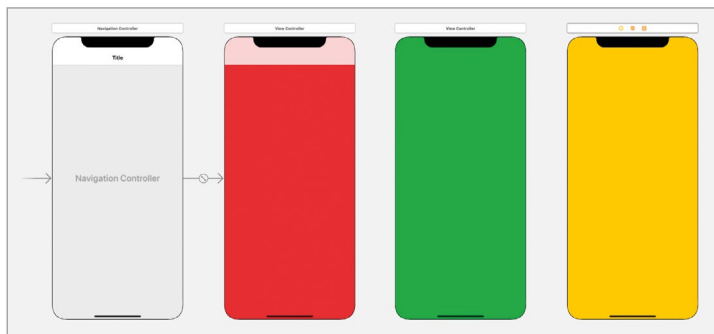


Рис. 9.11. Несколько View Controller на storyboard

Организуем навигацию между сценами, создав несколько переходов (segue).

- ▶ Разместите в центре красной сцены новую кнопку с текстом «**К зеленой сцене**».
- ▶ Зажмите клавишу **Control** и перетяните кнопку на зеленую сцену (точно так, как мы делали это при создании segue). При этом должна отображаться уже знакомая вам синяя линия.
- ▶ Во всплывающем окне выберите пункт **Show**.

Теперь красная сцена связана с зеленой с помощью сиквея, который сработает при нажатии на кнопку.

- ▶ Добавьте в центр зеленой сцены новую кнопку с текстом «**К желтой сцене**».
- ▶ Зажмите клавишу **Control** и перетяните кнопку на желтую сцену.
- ▶ Во всплывающем окне выберите пункт **Show**.

Теперь все сцены на storyboard связаны между собой и создают единую последовательность (рис. 9.12). Обратите внимание, что у зеленой и желтой сцен появилась кнопка «**Back**», с помощью которой можно вернуться к предыдущей сцене.

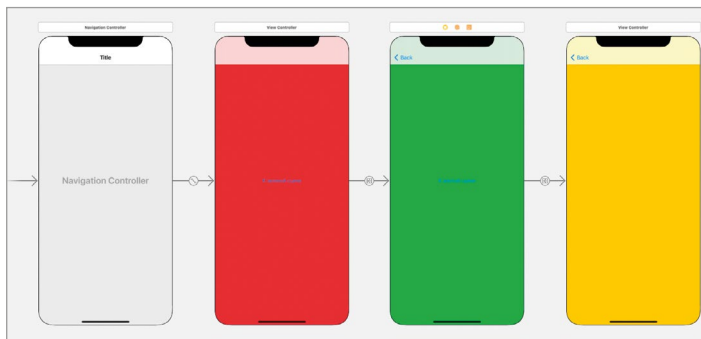


Рис. 9.12. Несколько View Controller с созданными переходами

Ранее в книге мы уже рассматривали использование слайда. При его вызове новая сцена выезжала снизу, а старая затемнялась и отдалялась. Но в данном случае segue будут работать несколько иначе, так как сработают внутри Navigation Controller.

► Запустите приложение.

Сейчас в навигационном стеке находится один вью контроллер (красный), и именно он отображается внутри навигационного контроллера (рис. 9.13). Красный контроллер в данном случае является корневым.

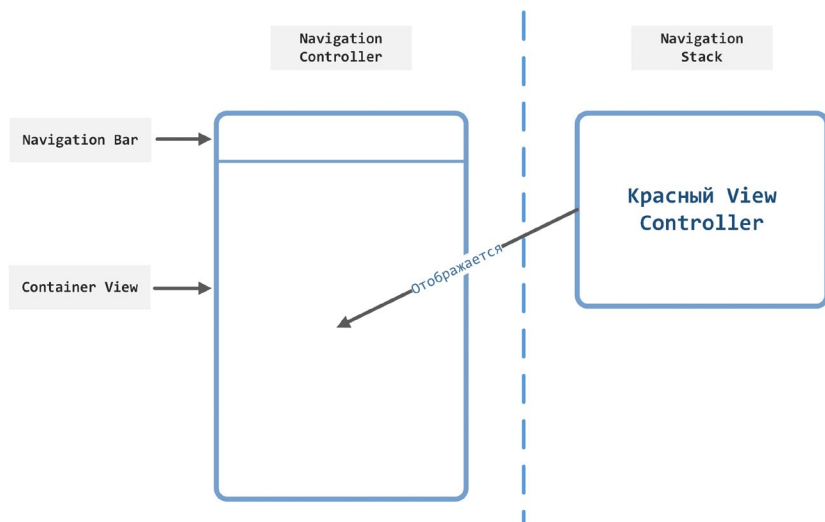


Рис. 9.13. Один View Controller в навигационном стеке

Теперь осуществим переход к следующей сцене.

► Нажмите кнопку «**К зеленой сцене**».

Обратите внимание на то, как именно произошел переход: вместо того, чтобы появиться снизу, зеленая сцена плавно выехала справа, а в навигационной панели отобразилась кнопка для возврата. Переход осуществляется внутри Navigation Controller, который, в свою очередь, переопределяет анимацию появления сцены.

Взглянем на текущее состояние навигационного стека: теперь в нем находятся два вью контроллера (рис. 9.14). При этом в Container View отображается та сцена, которая пришла в стек последней (то есть зеленая).

Продолжим наполнение навигационного стека новыми элементами и произведем переход к последней сцене.

► Нажмите кнопку «**К желтой сцене**».

И вновь уже знакомая нам анимация, и на экране отобразилась желтая сцена, также обернутая в навигационный контроллер. Теперь навигационный стек содержит три элемента (рис. 9.15).

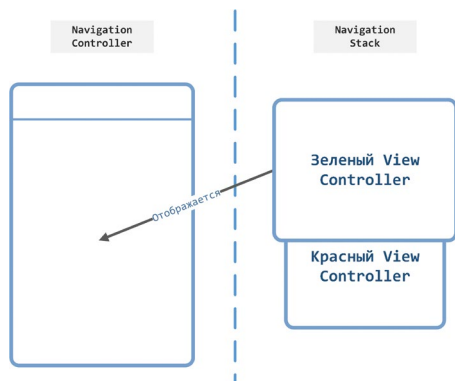


Рис. 9.14. Два View Controller в навигационном стеке

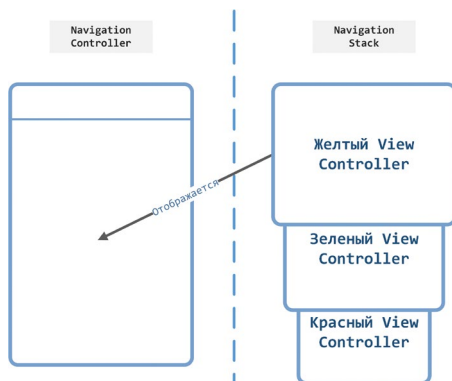


Рис. 9.15. Три View Controller в навигационном стеке

Чем больше сцен будет содержать приложение, тем глубже мы сможем «зарыться» в них, и тем больше элементов будет находиться в навигационном стеке.

Теперь попробуем вернуться к предыдущей сцене.

► Находясь на желтой сцене нажмите кнопку «Назад».

Красивая анимация, сдвигающая желтую сцену вправо, и перед нами вновь отобразилась зеленая сцена. Так как мы совершили переход назад, верхний элемент стека (желтый вью контроллер) был удален и больше не доступен. Теперь навигационный стек опять состоит из двух элементов (рис. 9.16), а на экране внутри Navigation Controller отображается верхний элемент стека – зеленая сцена.

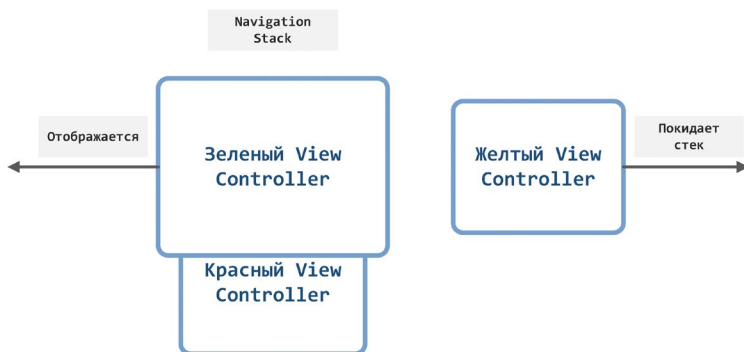


Рис. 9.16. Состав навигационного стека

Если снова нажать кнопку «Назад», мы совершим переход к корневому контроллеру, а стек вновь обновится, и теперь будет состоять всего из одного элемента (красного вью контроллера). На этом все – ниже корневой сцены опуститься нельзя.

9.4 Навигация с помощью программного кода

Класс **UINavigationController**, который и представляет собой навигационный контроллер, является дочерним по отношению к **UIViewController**. Данный класс предоставляет вам все необходимые свойства и методы для управления навигационным стеком и порядком отображения сцен. В этом разделе мы попрактикуемся в работе с классом **UINavigationController** и навигационным стеком с помощью кода: мы программно реализуем несколько вариантов перехода и посмотрим, как при этом изменяется состав стека, и какие возможности при этом доступны вам для его редактирования.

- Удалите со сториборда все переходы (segue), вызываемые по нажатию на кнопки на красной и зеленой сценах.

Примечание Будьте осторожны, не удалите связь между Navigation Controller и его корневой сценой (рис. 9.17).

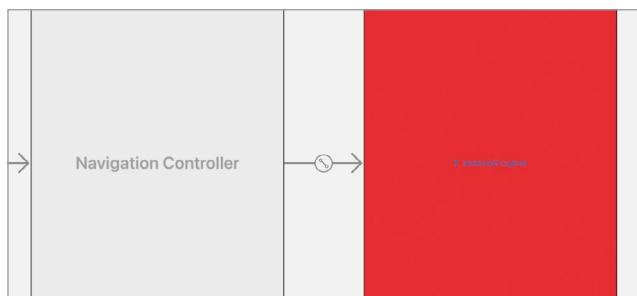


Рис. 9.17. Связь между Navigation Controller и корневой сценой

Переходы удалены, теперь нажатие на любую из кнопок, размещенных на сценах, не приводит к какому-либо результату.

Далее требуется средствами кода реализовать навигацию между сценами по нажатию на кнопки, размещенные на сценах. Для этого напишем несколько экшн-методов и свяжем их с кнопками.

В данный момент ни одна из сцен не связана с каким-либо программным классом. Поскольку наш проект является тестовым, мы не станем применять особое оформление к каждой сцене, размещать на них какие-либо уникальные элемен-

ты и наделять их особой функциональностью. По этой же причине для упрощения работы свяжем каждую сцену с одним и тем же классом **ViewController**, который уже есть в проекте.

- ▶ Выделите красный вью контроллер на сториборде.
- ▶ Откройте панель **Identity Inspector**.
- ▶ В поле **Class** введите **ViewController**.
- ▶ Прodelайте то же самое для зеленого и желтого вью контроллеров.

Теперь за работу каждой сцены будет отвечать один и тот же класс **ViewController**.

Примечание В реальных проектах стоит использовать отдельный класс для каждой отдельной сцены. Помните, что основная задача применения любого шаблона проектирования – это уменьшение связности. Элементы программы должны быть настолько независимы друг от друга, насколько это возможно.

Теперь для каждого контроллера на сториборде требуется определить уникальный идентификатор Storyboard ID.

- ▶ Выделите красный вью контроллер.
- ▶ Откройте панель **Identity Inspector**.
- ▶ В поле **Storyboard ID** укажите «**redViewController**».
- ▶ Прodelайте то же самое для зеленого и желтого вью контроллеров, указав соответственно «**greenViewController**» и «**yellowViewController**».

Примечание Мы уже работали с идентификаторами Storyboard ID ранее, тем не менее, вспомним еще раз, в каких случаях они используются.

Вью контроллер на сцене всегда связан с некоторым классом в программном коде. По умолчанию это **UIViewController**, но изменяя значение в поле **Class** (на панели **Identity Inspector**), эту связь можно переопределить. Разрабатывая интерфейс средствами **Interface Builder**, то есть размещая с помощью визуального редактора на сцене графические элементы, связанный класс совершенно ничего не знает об этих элементах. На этом этапе класс существует отдельно от сцены на сториборде.

Если при этом мы создадим экземпляр связанного класса с помощью программного кода, напрямую вызвав инициализатор (например, **ViewController()**), то в результате не получим никакой информации о размещенных на нем графических элементах (ведь они были размещены с помощью **Interface Builder**).

Для решения данной проблемы как раз и используются Storyboard ID. С его помощью, вызвав специальный метод, вы можете получить экземпляр связанного класса (например, **ViewController**) с информацией обо всех графических элементах, размещенных на сцене средствами **Interface Builder**. То есть, вы получите экземпляр класса, отталкиваясь не от программной реализации этого класса, а от визуальной структуры сцены на storyboard.

Добавим на желтую сцену кнопку, обеспечивающую переход к корневой сцене навигационного стека.

- Разместите на желтой сцене кнопку с текстом «**К корневой сцене**».

Перейдем к реализации программного кода.

Для загрузки контроллеров с использованием Storyboard ID необходимо воспользоваться возможностями класса **UIStoryboard**, получив с его помощью программный вариант файла **Main.storyboard**. Так как обращаться к этому файлу мы будем не один раз, создадим отдельное свойство.

- В классе **ViewController** реализуйте свойство из листинга 9.1.

ЛИСТИНГ 9.1

```
class ViewController: UIViewController {  
  
    // Ссылка на сториборд, где размещен данный ViewController  
    let storyboardInstance = UIStoryboard(name: "Main", bundle: nil)  
  
    // ...  
}
```

Обращаясь к свойству **storyboardInstance**, вы всегда будете получать один и тот же экземпляр, описывающий все элементы, размещенные на сториборде (в файле **Main.storyboard**).

- В классе **ViewController** реализуйте методы из листинга 9.2.

ЛИСТИНГ 9.2

```
// перейти к зеленой сцене  
@IBAction func toGreenScene(_ sender: UIButton) {  
    // получаем ссылку на следующий контроллер  
    // в данном случае следующий - это зеленый  
    let nextViewController = storyboardInstance.instantiateViewController(w  
ithIdentifier: "greenViewController")  
    // обращаемся к Navigation Controller  
    // и вызываем метод перехода к новому контроллеру  
    self.navigationController?.pushViewController(nextViewController,  
animated: true)  
}  
  
// перейти к желтой сцене  
@IBAction func toYellowScene(_ sender: UIButton) {  
    let nextViewController = storyboardInstance.instantiateViewController(w  
ithIdentifier: "yellowViewController")  
    self.navigationController?.pushViewController(nextViewController,  
animated: true)  
}
```

```
// перейти к корневой сцене
@IBAction func toRootScene(_ sender: UIButton) {
    // обращаемся к Navigation Controller
    // и вызываем метод перехода к корневому контроллеру
    self.navigationController?.popToRootViewController(animated: true)
}
```

- ▶ Свяжите реализованные методы и кнопки, размещенные на сценах:
 - метод **toGreenScene** с кнопкой на красной сцене;
 - метод **toYellowScene** с кнопкой на зеленой сцене;
 - метод **toRootScene** с кнопкой на желтой сцене.
- ▶ Произведите запуск приложения и протестируйте реализованную систему навигации.

Нажатия кнопок осуществляют переходы между сценами внутри навигационного контроллера. Более подробно разберем, что именно было нами реализовано.

1. В свойстве **storyboardInstance** хранится ссылка на программное представление файла **Main.storyboard**. При обращении к нему с помощью метода **instantiateViewController** происходит загрузка зеленого и желтого контроллеров. В результате этого возвращается экземпляр класса **ViewController** (все цветные сцены связаны с одним классом), содержащий в себе все графические элементы.

Передавая тот или иной Storyboard ID, вы получите экземпляр класса **ViewController**, включающий соответствующую структуру графических элементов.

2. Для программной реализации навигации в первую очередь необходимо получить доступ к навигационному контроллеру, в который обернута текущая сцена. Для этого используется свойство **navigationController** типа **UINavigationController?**. То есть, если сцена выводится внутри Navigation Controller, вы можете обратиться к свойству **navigationController** и получить текущий навигационный контроллер.

При этом совершенно неважно, обернут ли выю контроллер в Navigation Controller (как в нашем случае) или нет. Если обернут, то в данном свойстве хранится ссылка на данный экземпляр, если нет –там находится **nil**.

3. Класс **UINavigationController** включает в себя несколько специальных методов, позволяющих производить навигацию между сценами внутри навигационного контроллера:

- a. **pushViewController(_:animated:)** добавляет View Controller в навигационный стек и отображает соответствующую ему сцену на экране (внутри навигационного контроллера);

- b. `popToRootViewController(animated:)`** удаляет все контроллеры из навигационного стека и производит переход к корневой сцене.
- c. `popViewController(animated:)`** удаляет из навигационного стека верхний элемент и осуществляет переход к предыдущей сцене;
- d. `popToViewController(_:animated:)`** производит переход к конкретной сцене в навигационном стеке, удаляя все вышележащие элементы.

Примечание Обращаясь к свойству `navigationController`, вы получаете доступ к навигационным методам, описанным выше. Использование знака вопроса `?` при доступе к данному свойству позволяет безопасно работать с опциональными значениями. То есть, даже если сцена не будет обернута в навигационный контроллер (а такое вполне может быть, если она используется в нескольких местах в приложении), экстренного завершения не произойдет.

В коде выше мы попробовали в действии два из четырех описанных навигационных метода.

- В классе `ViewController` реализуйте метод из листинга 9.3.

ЛИСТИНГ 9.3

```
// перейти к предыдущему экрану
@IBAction func toPreviousScene(_ sender: UIButton) {
    self.navigationController?.popViewController(animated: true)
}
```

С помощью метода `toPreviousScene` будет осуществляться переход к предыдущему элементу навигационного стека. Это программный аналог кнопки возврата к предыдущей сцене, автоматически размещаемой на навигационной панели.

- На каждой из трех сцен разместите кнопку с текстом «**К предыдущему экрану**».
- Свяжите нажатие каждой из этих кнопок с вызовом метода `toPreviousScene`.
- Запустите проект, дойдите до последнего экрана и попробуйте совершить обратные переходы с помощью новых кнопок.

Обратите внимание, что нажатие кнопки «**К предыдущему экрану**» при нахождении на корневой красной сцене ни к чему не приводит. И в этом нет ничего удивительного, так как к этому моменту в навигационном стеке находится всего один элемент, и двигаться назад просто некуда.

Использование методов класса `UINavigationController` развязывает вам руки — вы можете создать такую систему навигации, которая требуется для реализации вашей идеи.

Ошибки при работе с навигационным стеком

Что, по вашему мнению, произойдет, если на желтой сцене создать кнопку, при нажатии на которую будет вызван метод `toGreenScene`?

Очевидно, что произойдет переход к зеленой сцене. Но к какой именно: той, что уже содержится в навигационном стеке, или в стек будет добавлен новый экземпляр класса **ViewController**, описывающий зеленую сцену?

Для ответа на этот вопрос вам необходимо внимательно посмотреть на те две строчки кода, что представляют собой тело метода **toGreenScene**. Все очевидно – при каждом срабатывании метода будет создан новый экземпляр класса **ViewController**, описывающий собой зеленую сцену.

- ▶ Разместите на желтой сцене кнопку с текстом «**К зеленой сцене**».
- ▶ Свяжите нажатие на нее с вызовом метода **toGreenScene**.
- ▶ Запустите проект на симуляторе и попробуйте произвести навигацию с использованием новой кнопки.

Нажимая кнопки «**К желтой сцене**» на зеленой сцене и «**К зеленой сцене**» на желтой сцене, на первый взгляд вы попадаете в петлю, когда две одинаковые сцены сменяют друг друга. Визуально это именно так, но функционально каждая «повторяющаяся» сцена – это совершенно новый и независимый от других экземпляр. Если в процессе разработки вы реализовали такой подход не умышленно, то это самая настоящая утечка памяти. При определенном стечении обстоятельств ваш навигационный стек будет содержать большое количество однотипных элементов, которые вы будете считать одним и тем же значением (рис. 9.18).

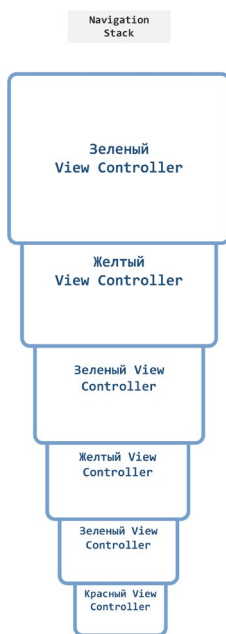


Рис. 9.18. Навигационный стек

Вам нужно быть осторожным и без необходимости не допускать такого поведения, так как каждый элемент стека занимает оперативную память, а ее перерасход может плохо сказаться на производительности приложения и системы в целом.

Доступ к навигационному стеку

В процессе разработки вам может потребоваться доступ к навигационному стеку, например, для того, чтобы найти в нем необходимую сцену и осуществить к ней переход. Для получения доступа к навигационному стеку используется свойство **viewControllers** класса **UINavigationController**:

```
self.navigationController?.viewControllers
```

Свойство **viewControllers** возвращает массив значений типа **UIViewController**, содержащий в себе ссылки на все контроллеры навигационного стека. Данное значение является изменяемым, а значит вы можете произвольным образом модифицировать его (удалять и вставлять элементы). При этом, если вы изменяете верхний элемент стека (последний элемент массива **viewControllers**), например, удаляете его или вставляете новый, изменения тут же отобразятся на экране (без какой-либо анимации).

Вот несколько примеров работы с навигационным стеком, которые могут вам понадобиться:

1. Возврат к определенной сцене

```
// перебираем все элементы стека
self.navigationController?.viewControllers.forEach { viewController in
    // определяем требуемый контроллер
    // SomeViewController – это класс, соответствующий искомому контроллеру
    if viewController is SomeViewController {
        // производим возврат к нему
        self.navigationController?.popToViewController(viewController,
animated: true)
    }
}
```

2. Работа с корневой сценой

```
// изменяем корневую сцену
self.navigationController?.viewControllers[0] = someViewController
```

В этой книге мы не будем делать настолько сложные проекты, в которых требуется работа с навигационным стеком напрямую. Но в будущих книгах и в вашей профессиональной карьере вы будете с завидной регулярностью пользоваться описанным материалом.

9.5 Визуальное оформление Navigation Controller

Navigation Controller – это контейнер, отображающий внутри себя дочерние сцены, который позволяет изменять значения некоторых параметров, влияющих на визуальное отображение этих сцен. Разберем некоторые из них.

Текстовое наполнение Navigation Bar

В Navigation Bar отображается название текущей сцены, правда сейчас ни для одной из сцен названия не определены. Поэтому кроме кнопки возврата на панели ничего нет.

Для того, чтобы указать название сцены, в ее состав должен входить элемент Navigation Item. Сейчас он есть у каждой сцены на storyboard в нашем проекте (рис. 9.19). Но если его там нет, просто найдите его в библиотеке и разместите на сцене (рис. 9.20).

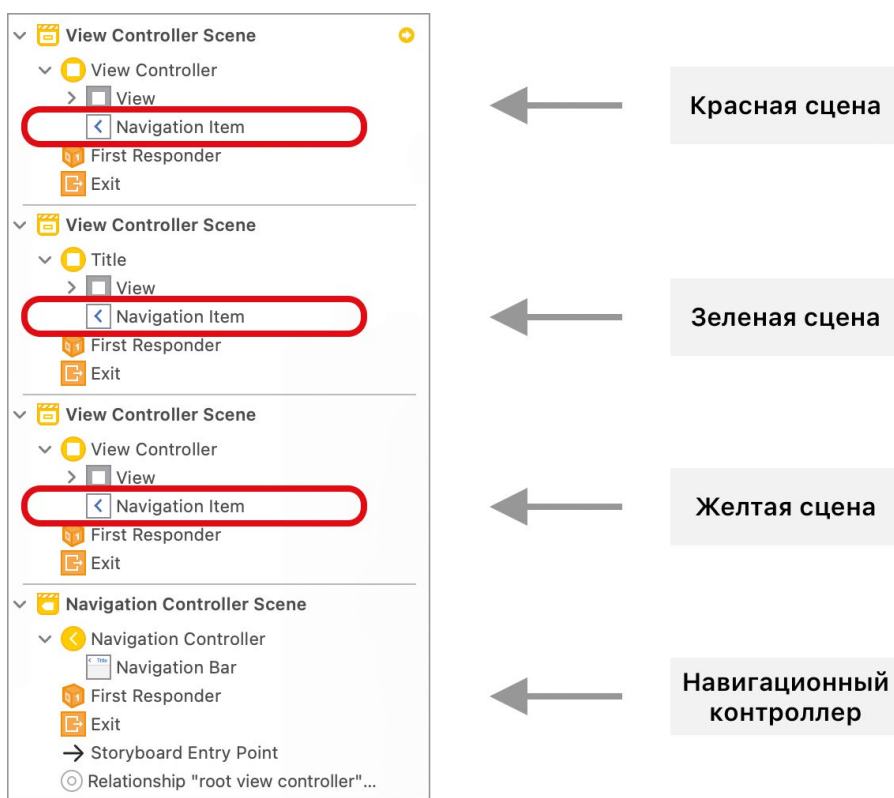


Рис. 9.19. Navigation Item в составе сцен

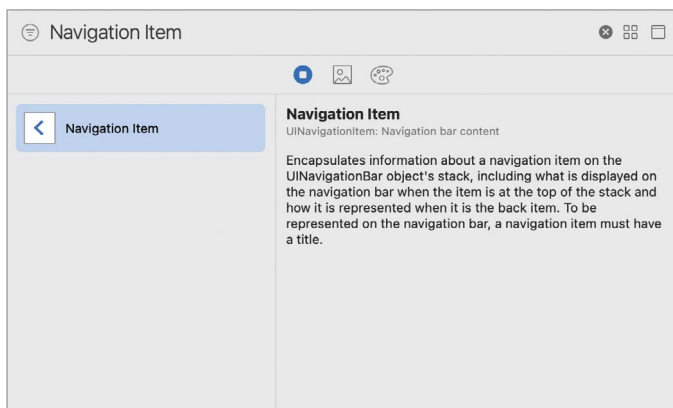


Рис. 9.20. Navigation Item в библиотеке объектов

Примечание В составе каждой сцены в нашем проекте уже есть Navigation Item. Это связано с тем, что ранее все сцены были добавлены в качестве дочерних к навигационному контроллеру прямо на storyboard с помощью переходов (segue). Позже переходы были удалены, а вот Navigation Item в составе остались.

Выделив Navigation Item на панели **Attributes Inspector**, можно настроить заголовок сцены, вспомогательный текст, а также текст кнопки возврата, которая будет показана на следующей сцене (рис. 9.21).

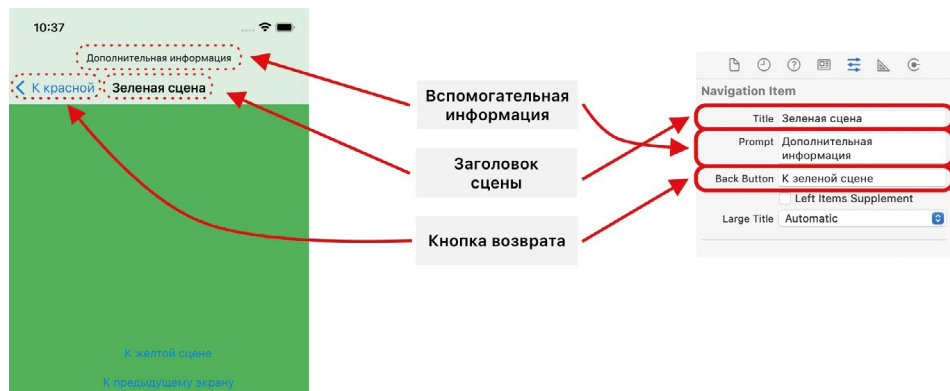


Рис. 9.21. Параметры Navigation Item

Примечание Обратите особое внимание на то, что поле Back Button определяет текст, который будет отображен на кнопке возврата на следующей сцене в стеке, а не на текущей.

Данные параметры могут быть определены также с помощью программного кода, например, в методе **viewDidLoad**:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.navigationItem.prompt = "Подсказка"
    self.navigationItem.title = "Заголовок сцены"
    self.navigationItem.backBarButtonItem = "Назад"
}
```

Стиль заголовка Navigation Bar

Операционная система iOS поддерживает два стиля оформления заголовка сцены в Navigation Bar: стандартный и увеличенный (рис. 9.22). По умолчанию используется стандартный стиль оформления. Для включения увеличенного стиля необходимо активировать пункт **Prefers Large Title** на панели **Attributes Inspector** для Navigation Bar в составе навигационного контроллера (рис. 9.23). После этого, изменяя у Navigation Item каждого выю контроллера значение поля **Large Title** (панель **Attributes Inspector**), вы сможете менять стиль необходимым вам образом (рис. 9.24).

Помимо описанных выше, вам доступны и другие параметры, например, цвет текста и фона в Navigation Bar, его прозрачность и многое-многое другое. Я рекомендую вам самостоятельно поэкспериментировать, попробовав все из доступных настроек.

На этом мы завершаем наше знакомство с навигационным контроллером. Полученных вами знаний вполне хватит для того, чтобы начать создавать приложение на основе Navigation Controller.

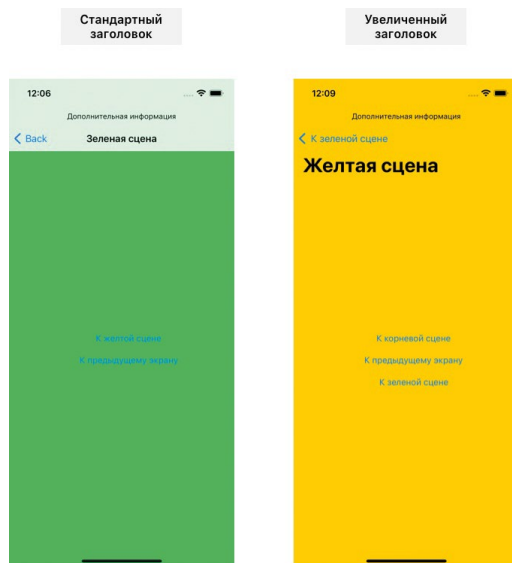


Рис. 9.22. Стили заголовка в Navigation Bar

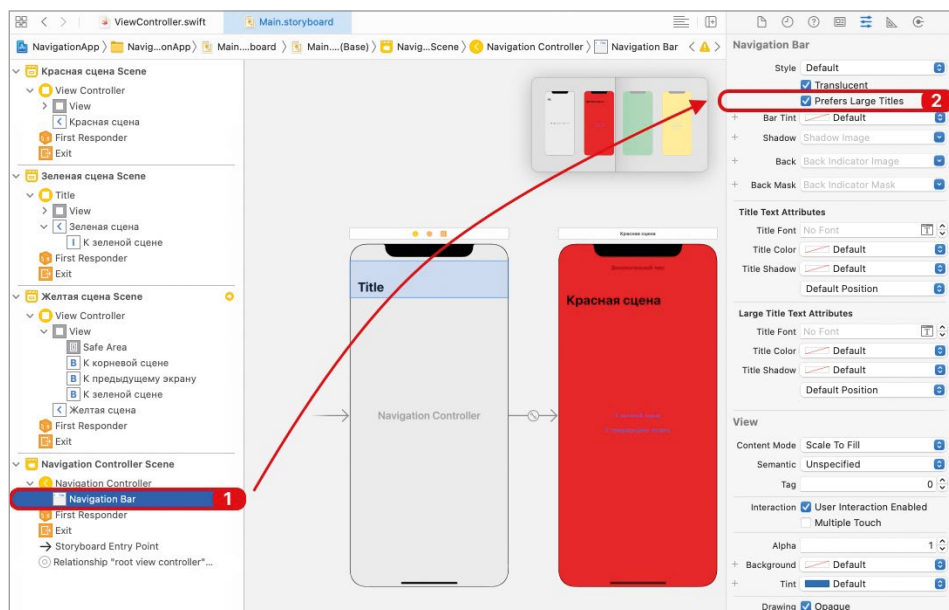


Рис. 9.23. Установка увеличенного стиля оформления заголовка сцены

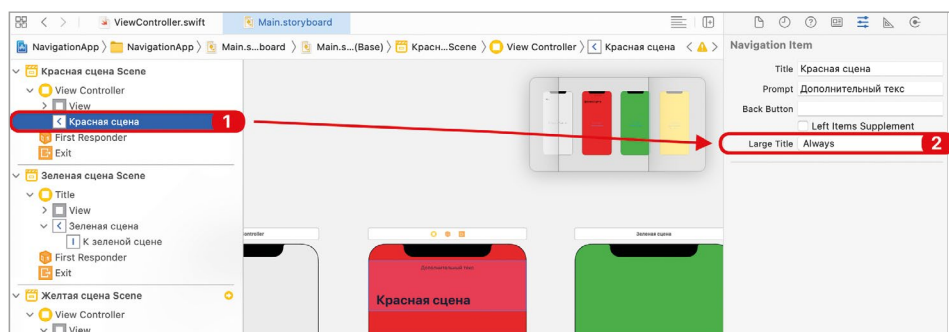


Рис. 9.24. Установка увеличенного стиля сцены

Глава 10.

Передача данных между контроллерами

В этой главе вы:

- узнаете основные способы передачи данных между вью контроллерами в приложении.

Приложение – это множество взаимосвязанных между собой экранов. Одни экраны просто сменяют друг друга, а другие подразумевают односторонний или двусторонний обмен данными. Вы уже довольно неплохо разбираетесь в вопросах навигации между сценами – segue и Navigation Controller для вас уже не темные лошадки. Но пока что сцены всех разработанных вами проектов функционировали независимо друг от друга, а это значит, что их совершенно ничего не объединяло. Однако, чем глубже вы будете погружаться в iOS-разработку, тем острее будет вставать вопрос взаимодействия отдельных сцен посредством обмена данными.

В качестве примера рассмотрим страницу профиля в медицинском приложении «Голдлайн». На рисунке 10.1 показан процесс выбора диагноза путем перехода между сценами. Изначально диагноз не выбран, но при нажатии на соответствующую ячейку происходит переход к экрану выбора, а после выбора измененные данные отображаются на экране профиля.

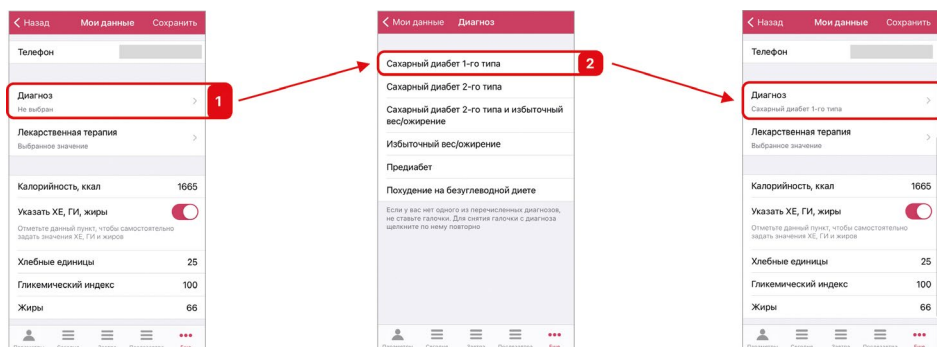


Рис. 10.1. Передача выбранного диагноза между экранами

Какое бы приложение вы не разрабатывали (простое или сложное), перед вами обязательно встанет вопрос реализации обмена данными между вью контроллерами. В этой главе мы рассмотрим некоторые из доступных в Swift и Xcode способов решения этой задачи.

10.1 Создание проекта

На рисунке 10.2 показана типовая схема взаимодействия контроллеров внутри приложения.



Рис. 10.2. Схема взаимодействия контроллеров в приложении

Контроллер А отображает некоторые данные. При необходимости их изменения будет происходить переход к контроллеру Б. После изменения обновленные данные будут вновь передаваться в контроллер А для отображения.

Руководствуясь приведенной схемой, создадим проект, на основе которого и будем изучать вопрос обмена данными между контроллерами. В составе проекта будут созданы две сцены, а для навигации между ними воспользуемся навигационным контроллером. На первом экране (контроллер А) будет размещена текстовая метка для отображения, а на втором (контроллер Б) – текстовое поле для изменения. Значения в текстовой метке и текстовом поле всегда должны совпадать. При переходе с контроллера А будут передаваться данные для заполнения текстового поля, а при его изменении и сохранении – обратно, для вывода в текстовой метке.

В этой главе мы рассмотрим несколько способов передачи данных между сценами. Для каждого из них будем создавать свой набор кнопок на сцене, а также порассуждаем об особенностях каждого подхода.

Начнем с создания нового проекта.

- ▶ Создайте проект **TransferApp**. В качестве шаблона выберите **App**.
- ▶ Откройте файл **Main.storyboard**.

В составе проекта уже есть один выю контроллер. Будем называть его **контроллер А** или **сцена А**.

В качестве базы для навигации между сценами будем использовать Navigation Controller.

- ▶ Оберните контроллер А в Navigation Controller (пункт меню **Editor > Embed In > Navigation Controller**).

Изменим внешний вид сцены.

- ▶ Измените стиль оформления заголовка в Navigation Bar на **Large**:
 - в Navigation Controller на панели **Attributes Inspector** активируйте пункт **Prefers Large Titles**;
 - в Navigation Item контроллера А на панели **Attributes Inspector** измените значение поля **Large Title** на **Always**.
- ▶ Измените заголовок контроллера А, выводимый в Navigation Bar, на «**Сцена А**».
- ▶ Измените фоновый цвет сцены А на фиолетовый.

Далее на сцене разместим и настроим текстовую метку (Label), в которой будет выводиться актуальное строковое значение.

- ▶ Разместите в центре сцены А текстовую метку. Для позиционирования самостоятельно создайте необходимые констрейнты.

Примечание Даже если вы не сможете создать необходимые ограничения для размещаемых на сцене элементов, это никак не повлияет на функциональную составляющую проекта. Тем не менее, я настоятельно рекомендую вам постараться и (при возникновении сложностей) используя уже изученный ранее материал, а также помощь в нашем чате в Telegram, создать требуемые ограничения.

- ▶ Измените цвет текста метки на белый, а размер шрифта на 30.
- ▶ Свяжите сцену А с классом **ViewController**.
- ▶ В классе **ViewController** создайте аутлет **dataLabel** и свяжите его с текстовой меткой на сцене А (листинг 10.1).

ЛИСТИНГ 10.1

```
class ViewController: UIViewController {
    @IBOutlet var dataLabel: UILabel!
    // ...
}
```

На этом подготовка контроллера А завершена (рис. 10.3).

Теперь добавим на storyboard дополнительный контроллер (контроллер Б), отвечающий за смену строкового значения.



Рис. 10.3. Storyboard проекта

- ▶ Разместите на сториборде новый вью контроллер.
- ▶ Измените фоновый цвет сцены на зеленый.
- ▶ Измените заголовок сцены, выводимый в Navigation Bar, на «Сцена Б». Для этого потребуется добавить на сцену Navigation Item.
- ▶ Разместите в центре сцены Б текстовое поле (Text Field). Для позиционирования используйте ограничения.
- ▶ Определите отступы слева и справа от текстового поля в 30 точек.
- ▶ Выворняйте содержимое внутри текстового поля по центру.

Визуальная составляющая сцены готова (рис. 10.4). Обратите внимание, что пока мы не создадим segue между первым и вторым контроллерами, Navigation Bar не будет отображаться на storyboard.

Теперь создадим класс, который будет управлять сценой Б.

- ▶ Создайте новый файл **SecondViewController.swift** с классом **SecondViewController**, который является потомком **UIViewController** (листинг 10.2).

ЛИСТИНГ 10.2

```
class SecondViewController: UIViewController {
    // ...
}
```

- ▶ Свяжите контроллер Б с классом **SecondViewController**.
- ▶ В классе **SecondViewController** создайте аутлет **dataTextField** и свяжите его с текстовым полем на сцене Б (листинг 10.3).

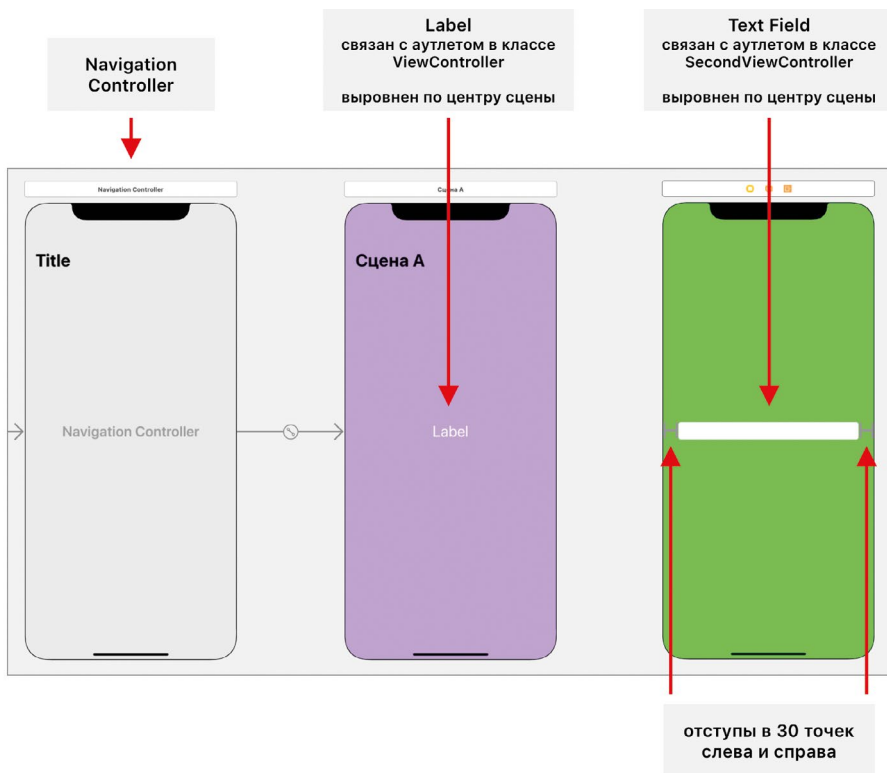


Рис. 10.4. Storyboard проекта

ЛИСТИНГ 10.3

```
class SecondViewController: UIViewController {
    @IBOutlet var dataTextField: UITextField!
    // ...
}
```

Теперь укажем Storyboard ID для каждого из контроллеров на сториборде.

- ▶ Для контроллера А в поле **Storyboard ID** укажите **ViewController**.
- ▶ Для контроллера Б в поле **Storyboard ID** укажите **SecondViewController**.

Примечание Не перепутайте Storyboard ID для каждого из контроллеров, он идентичен имени класса, связанного со сценой.

Весь дальнейший материал, описываемый в главе, будет основан на использовании данного проекта. В последующих разделах мы рассмотрим несколько способов передачи данных между контроллерами в каждую из сторон.

10.2 Передача данных от А к Б с помощью свойств

Наиболее простым способом передать данные из одного контроллера в другой является использование свойств. Всякий раз при создании экземпляра вью контроллера, к которому производится переход, данные инициализируются в предназначенное для них свойство в этом экземпляре. При выводе сцены на экран данные в этом свойстве могут быть использованы для наполнения графических элементов (например, для заполнения текстового поля).

Примечание Прошу обратить внимание, что в этой главе мы рассматриваем вопросы передачи данных между контроллерами, а не последующего использования этих данных для обновления элементов интерфейса.

- ▶ На сцене А под текстовой меткой разместите кнопку **«Изменить с помощью свойства»**. С помощью ограничений укажите отступ в 30 пикселей сверху (от текстовой метки), слева и справа.
- ▶ Измените фоновый цвет кнопки на синий, цвет шрифта на белый, а размер текста на 20 (рис. 10.5).

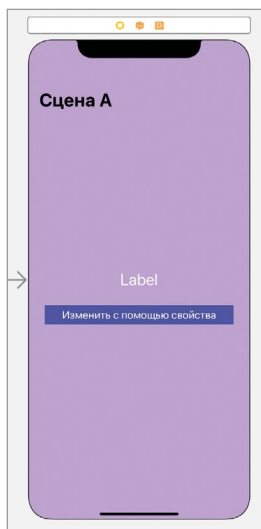


Рис. 10.5. Кнопка на сцене А

При нажатии созданной кнопки будет происходить передача данных в контроллер Б с последующим переходом к его сцене. Данные будут записываться в специальное свойство **updatingData**.

- ▶ В классе **SecondViewController** объявите свойство **updatingData** типа **String** (листинг 10.4).

ЛИСТИНГ 10.4

```
var updatingData: String = ""
```

Свойство **updatingData** будет использоваться для заполнения данными текстового поля на сцене Б. Каждый раз при отображении сцены значение текстового поля будет обновляться в соответствии со значением свойства.

► Дополните код класса **SecondViewController** методами из листинга 10.5.

ЛИСТИНГ 10.5

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    updateTextFieldData(withText: updatingData)
}

// обновляем данные в текстовом поле
private func updateTextFieldData(withText text: String) {
    dataTextField.text = text
}
```

Метод **viewWillAppear** относится к жизненному циклу вью контроллера. Напомним, он вызывается при каждом отображении сцены на экране, а не только при первом, как **viewDidLoad**.

Теперь реализуем непосредственно передачу данных и переход к сцене Б.

► В классе **ViewController** создайте экшн-метод **editDataWithProperty** (листинг 10.6).

ЛИСТИНГ 10.6

```
@IBAction func editDataWithProperty(_ sender: UIButton) {
    // получаем вью контроллер, в который происходит переход
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    let editScreen = storyboard.instantiateViewController(withIdentifier:
"SecondViewController") as! SecondViewController

    // передаем данные
    editScreen.updatingData = dataLabel.text ?? ""

    // переходим к следующему экрану
    self.navigationController?.pushViewController(editScreen, animated:
true)
}
```

► Свяжите вызов метода **editDataWithProperty** с нажатием кнопки «**Изменить с помощью свойства**» на сцене А.

► Запустите приложение.

При нажатии на кнопку «**Изменить с помощью свойства**» значение текстовой метки контроллера А передается в текстовое поле контроллера Б. Это очень простой, но наиболее очевидный и понятный способ передачи данных между контроллерами.

На что стоит обратить внимание

Работая над программным кодом ваших приложений, всегда старайтесь думать о том, а не навредит ли принятое решение архитектуре проекта. В частности, в использованной реализации создается излишняя связанность между вью контроллерами, так как внутри одного контроллера происходит непосредственная работа с другим контроллером. Если вдруг потребуется изменить название или тип свойства `updatingData`, или подставить вместо `SecondViewController` совершенно иной контроллер, это приведет к необходимости внесения изменений и в класс `ViewController`. Излишняя связанность не очень хорошо сказывается на архитектуре приложения, так как она приводит к необходимости порой неожиданных изменений.

Можно избежать подобной ситуации, создав между контроллерами прослойку в виде протокола, описывающего требования к наличию свойству `updatingData`. Подписав на протокол класс `SecondViewController` внутри метода `editDataWithProperty`, мы сможем производить тайпкастинг (приведение) не к типу `SecondViewController`, а к созданному протоколу:

```
protocol UpdatingDataController: class {
    var updatingData: String { get set }
}

class SecondViewController: UIViewController, UpdatingDataController {
    var updatingData: String = ""
    // ...
}

class ViewController: UIViewController {
    // ...
    @IBAction func editDataWithProperty (_ sender: UIButton) {
        // ...
        var editScreen = storyboard.instantiateViewController(withIdentifier:
"SecondViewController") as! UpdatingDataController
        // ...
    }
}
```

Такой подход позволяет скрыть за протоколом внутреннюю реализацию и конкретный тип контроллера, а значит при необходимости использовать любой вью контроллер, соответствующий протоколу **UpdatingDataController**.

10.3 Передача данных от Б к А с помощью свойств

Данные успешно передаются от контроллера А в контроллер Б и отображаются в текстовом поле. Теперь рассмотрим вопрос обратной передачи измененных данных в контроллер А.

В первую очередь, мы реализуем обновление текстовой метки на сцене А при ее отображении на экране. То есть, каждый раз, когда сцена появляется на экране, текст в метке должен измениться на актуальный.

► Добавьте в класс **ViewController** код из листинга 10.7.

ЛИСТИНГ 10.7

```
var updatedData: String = "Test data"

override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    updateLabel(withText: updatedData)
}

// Обновляем данные в текстовой метке
private func updateLabel(withText text: String) {
    dataLabel.text = updatedData
}
```

Принцип тот же самый, что был ранее использован в классе **SecondViewController**.

Для передачи обновленного значения из контроллера Б в контроллер А нам необходимы два элемента: измененное значение и контроллер назначения.

Примечание Контроллер назначения – это вью контроллер, в который осуществляется переход.

Измененное значение может быть получено с помощью свойства **dataTextField**. А вот для получения контроллера назначения (в нашем случае — контроллера А) есть несколько вариантов исполнения.

Вариант 1. Так как отображение сцен производится внутри навигационного контроллера, можно обратиться к навигационному стеку, получить ссылку на предыдущий контроллер и обновить значение свойства **updatedData**.

Вариант 2. В классе **SecondViewController** создать свойство, хранящее ссылку на вью контроллер. При переходе от сцены А к сцене Б инициализировать этому свойству ссылку на класс **ViewController** и с его помощью обновлять значение свойства **updatedData**.

Каждый из указанных вариантов имеет свои положительные и отрицательные стороны, о которых мы поговорим ниже. Мы же воспользуемся первым.

- ▶ На сцене Б ниже текстового поля разместите кнопку «**Сохранить с помощью свойства**». Укажите ограничение на отступ в 30 точек сверху, слева и справа.
- ▶ Оформите кнопку в соответствии с использованным ранее стилем на сцене А (измените фон, размер текста и цвет шрифта) (рис. 10.6).



Рис. 10.6. Оформление сцены Б

- ▶ В классе **SecondViewController** реализуйте экшн-метод **saveDataWithProperty** из листинга 10.8.

ЛИСТИНГ 10.8

```
@IBAction func saveDataWithProperty(_ sender: UIButton) {
    self.navigationController?.viewControllers.forEach { viewController in
        (viewController as? ViewController)?.updatedData = dataTextField.
text ?? ""
    }
}
```

- ▶ Свяжите вызов метода **saveDataWithProperty** с нажатием кнопки «**Сохранить с помощью свойства**».
- ▶ Запустите проект.

После перехода на сцену Б, изменения данных в текстовом поле и нажатия на кнопку сохранения данные в контроллере А обновляются, а по возвращению на сцену А мы видим уже обновленное значение.

Примечание Вместо создания свойства **updatedData** можно было обновлять значение в текстовой метке вызовом метода **updateLabel** класса **ViewController**.

На что стоит обратить внимание

Выше я описал два варианта доступа к экземпляру класса **ViewController**: через навигационный стек и с помощью свойства. Хочу обратить ваше внимание, что при определенном уровне невнимательности использование свойства могло привести к утечке памяти! Если созданное свойство будет держать сильную ссылку на **ViewController**, контроллер Б не будет удален до тех пор, пока не будет удален контроллер А. В результате, возвращаясь со сцены Б к сцене А, экземпляр класса **SecondViewController** будет продолжать занимать ценную оперативную память.

Примечание Если данный материал вызвал у вас затруднения, вернитесь к главе про управление памятью и ARC в первой книге.

Выходом из этой ситуации является хранение внутри **SecondViewController** слабой (**weak**) ссылки на экземпляр **ViewController** вместо сильной. Очень важно следить за тем, чтобы ссылающиеся друг на друга объекты при появлении соответствующих условий могли быть удалены из памяти средствами ARC.

Также вновь вернусь к вопросу сильной связанности классов, так как мы получили ту же самую проблему, что и ранее. И вновь решить ее можно с помощью протоколов. Достаточно создать протокол, требующий наличия свойства **updatedData**, подписать на него класс **ViewController** и использовать его при тайпкастинге вместо конкретного типа **ViewController**:

```
protocol UpdatableDataController: class {
    var updatedData: String { get set }
}

class ViewController: UIViewController, UpdatableDataController {
    var updatedData: String = ""
    // ...
}

class SecondViewController: UIViewController {
    // ...
    @IBAction func saveDataWithProperty(_ sender: UIButton) {
```

```
        self.navigationController?.viewControllers.forEach {  
viewController in  
            (viewController as? UpdatableDataController)?.updatedData =  
dataTextField.text ?? ""  
        }  
    }  
}
```

10.3 Передача данных от А к Б с помощью segue

Segue уже довольно хорошо вам известны. С их помощью в графическом интерфейсе **Interface Builder** можно с большим уровнем удобства решить вопрос навигации между сценами. При срабатывании segue пытается вызвать специальный метод **prepare** вью контроллера, из которого происходит переход. С его помощью можно выполнить требуемый код, в том числе, передать данные в контроллер назначения.

- ▶ На сцене А разместите новую кнопку «**Изменить с помощью segue**». Она должна находиться ниже на 30 точек уже существующей кнопки с отступами в 30 точек слева и справа. Не забывайте для этого использовать механизм ограничений.
- ▶ Измените стиль кнопки в соответствии со стилем других кнопок.
- ▶ Создайте segue от созданной кнопки к контроллеру Б. Для этого нажмите клавишу **Control** и перетяните кнопку на вторую сцену, а в выпадающем окне выберите «**Show**».

Сразу после создания на сцене Б отобразится Navigation Bar и заголовок сцены (рис. 10.7).

- ▶ На storyboard выделите созданный segue, откройте панель **Attributes Inspector** и в поле **Identifier** введите значение «**toEditScreen**».

Вью контроллер на storyboard может иметь множество переходов (segues). Если они используются только для презентации сцен, никаких сложностей не создается. Но если в рамках перехода также требуется выполнить какие-либо дополнительные действия (например, передать данные в контроллер назначения), необходимо использовать механизм, позволяющий отличить один segue от другого. И таким механизмом является идентификатор перехода (он был только что указан в поле **Identifier**).

Идентификатор – это уникальное, в пределах контроллера, строковое значение, позволяющее однозначно идентифицировать segue. При осуществлении

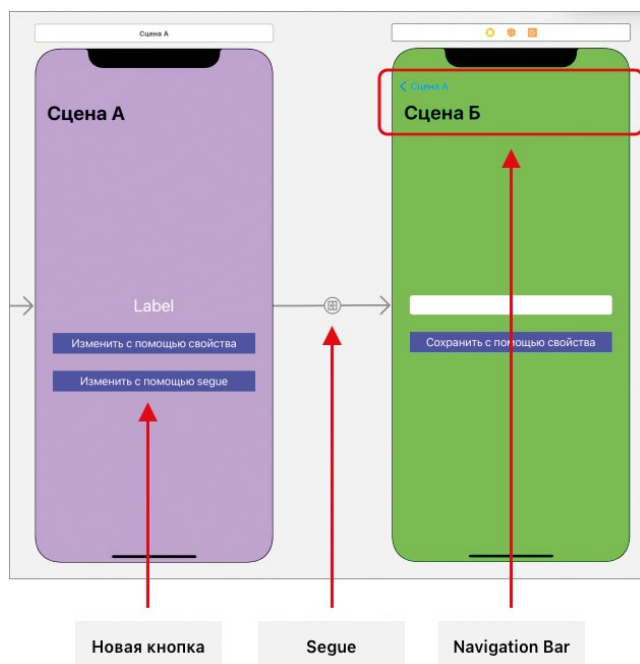


Рис. 10.7. Новые элементы на storyboard

перехода к новому контроллеру, используя идентификатор, можно определить какой именно segue сработал, и в зависимости от этого выполнить необходимые действия.

Для обработки перехода используется метод **prepare** класса **UIViewController**.

СИНТАКСИС

Метод `UIViewController.prepare(for:sender:)`

Вызывается в момент срабатывания segue непосредственно перед осуществлением перехода к новой сцене.

Аргументы

- **for:** `UIStoryboardSegue` – описывает сработавший segue. Используется для получения идентификатора, источника перехода (контроллера, с которого происходит переход) и контроллера назначения.
- **sender:** `Any?` – описывает элемент, вызвавший segue. Например, если segue привязан к нажатию кнопки, то в **sender** будет находиться экземпляр `UIButton`, соответствующий этой кнопке на сцене.

Особое внимание стоит обратить на аргумент **for** типа `UIStoryboardSegue`. С его помощью происходит идентификация segue, а также получение контроллеров источника и назначения.

СИНТАКСИС

Тип `UIStoryboardSegue`

Описывает segue (переход) между сценами.

Важные свойства

- `identifier: String` – идентификатор перехода.
- `destination: UIViewController` – контроллер назначения (к которому происходит переход).
- `source: UIViewController` – контроллер-источник, с которого происходит переход.

Реализуем метод **prepare**, позволяющий обработать переход и передать данные во вью контроллер.

► Добавьте в класс **ViewController** код из листинга 10.9.

ЛИСТИНГ 10.9

```
// Передача данных с помощью segue
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // определяем идентификатор segue
    switch segue.identifier {
    case "toEditScreen":
        // обрабатываем переход
        prepareEditScreen(segue)
    default:
        break
    }
}

// подготовка к переходу на экран редактирования
private func prepareEditScreen(_ segue: UIStoryboardSegue) {
    // безопасно извлекаем опциональное значение
    guard let destinationController = segue.destination as?
    SecondViewController else {
        return
    }
    destinationController.updatingData = dataLabel.text ?? ""
}
```

► Запустите проект.

При нажатии на кнопку «**Изменить с помощью segue**» происходит переход к сцене редактирования, при этом значение из текстовой метки передается в текстовое поле. Сохранить данные (то есть передать измененные данные обратно) можно нажатием на созданную ранее кнопку «**Сохранить с помощью свойства**».

Огромным плюсом использования segue является их визуализация на storyboard. Стоит вам взглянуть на созданные связи между сценами, как становится ясно, каким образом происходит навигация внутри приложения, и как передаются данные. При этом реализация передачи данных, как вы только что убедились, очень проста.

10.4 Передача данных от Б к А с помощью unwind segue

Unwind segue – это особый вид перехода (segue), с помощью которого можно вернуться к одной из сцен, отображенных ранее. При этом у unwind segue есть несколько особенностей.

- Они не ограничены последней показанной сценой. Вы можете вернуться к любой сцене, показанной ранее, пропустив все промежуточные. Например, если в навигационном стеке четыре контроллера, вы с легкостью сможете создать unwind segue, который одним нажатием вернет вас к любому из этих контроллеров.
- Не имеет значения, каким образом выводится текущая сцена: с помощью метода **present** или внутри Navigation Controller – unwind segue способен произвести обратный переход.

Воспринимайте unwind segue, как обычный segue, но производящий переход в обратную сторону.

Для использования unwind segue необходимо реализовать метод с произвольным именем, принимающий всего один параметр типа **UIStoryboardSegue**. Самое важное, что данный метод необходимо реализовывать в том контроллере, в который будет производиться обратный переход, а не в том из которого производится возврат. То есть, если нам необходимо вернуться из контроллера **SecondViewController** к первой сцене, то метод реализуется в классе **ViewController** (а не **SecondViewController**).

► В классе **ViewController** реализуйте метод **unwind** из листинга 10.10.

ЛИСТИНГ 10.10

```
@IBAction func unwindToFirstScreen(_ segue: UIStoryboardSegue) {}
```

Для реализации unwind segue важен сам факт наличия данного метода, при этом нет необходимости наполнять его тело каким-либо кодом.

- На сцене Б разместите кнопку «**Сохранить с помощью unwind**». Она должна находиться ниже на 30 точек уже существующей кнопки с отступами в 30 точек слева и справа. Не забывайте для этого использовать механизм ограничений.

- Измените стиль кнопки в соответствии со стилем другим кнопок.

В результате на сцене появится новая кнопка (рис. 10.8).

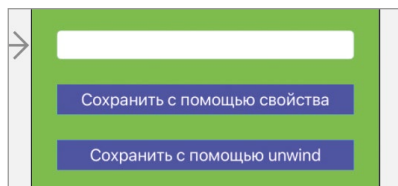


Рис. 10.8. Новая кнопка на сцене Б

- Нажмите клавишу **Control** и перетяните данную кнопку на элемент **Exit** в составе сцены. Найти его можно либо в **Document Outline**, либо прямо над сценой на сториборде (рис. 10.9).

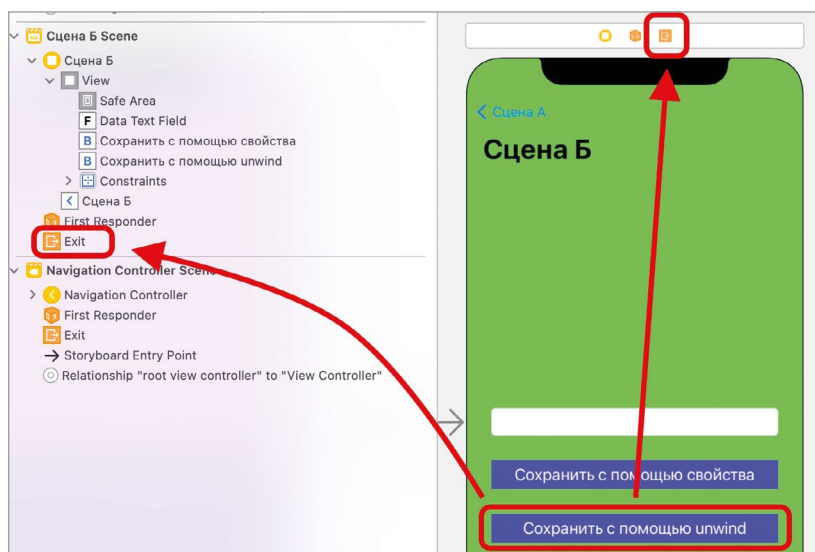


Рис. 10.9. Создание связи между кнопкой и элементом Exit

- Во всплывающем окне выберите пункт **unwindToFirstScreen** (рис. 10.10). Название пункта соответствует названию метода в первом контроллере.

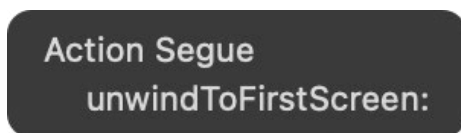


Рис. 10.10. Выбор unwind segue

Вы только что создали свой первый unwind segue. Обратите внимание, что в **Document Outline** в составе сцены Б появился новый элемент «**Unwind segue to unwindToFirstScreen**» (рис. 10.11).

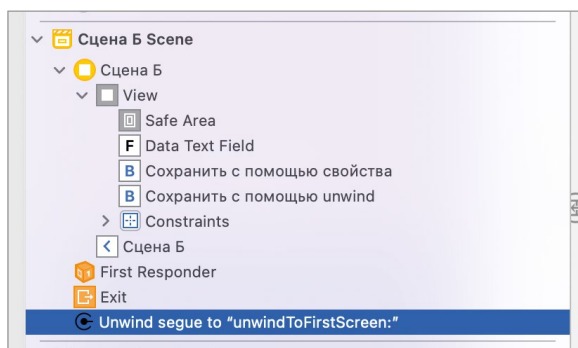


Рис. 10.11. Созданный unwind segue

- ▶ Запустите приложение.

При нажатии кнопки «**Сохранить с помощью unwind**» происходит переход к корневой сцене навигационного стека, а это говорит о том, что созданный unwind segue работает. Переход производится, но данные пока еще не передаются.

Реализуем обратную передачу данных. Для этого необходимо указать идентификатор unwind segue, и, используя его в методе **prepare**, передавать данные, как мы уже делали это в предыдущем разделе.

- ▶ В составе сцены Б в **Document Outline** выделите элемент «**Unwind segue to unwindToFirstScreen**».
- ▶ Откройте панель **Attributes Inspector**.
- ▶ В поле **Identifier** укажите «**toFirstScreen**».
- ▶ В классе **SecondViewController** реализуйте методы из листинга 10.11.

ЛИСТИНГ 10.11

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // определяем идентификатор segue
    switch segue.identifier {
    case "toFirstScreen":
        // обрабатываем переход
        prepareFirstScreen(segue)
    default:
        break
    }
}
```



```
// подготовка к переходу на первый экран
private func prepareFirstScreen(_ segue: UIStoryboardSegue) {
    // безопасно извлекаем опциональное значение
    guard let destinationController = segue.destination as? ViewController
    else {
        return
    }
    destinationController.updatedData = dataTextField.text ?? ""
}
```

- Осуществите запуск приложения.

Теперь после нажатия кнопки «Сохранить с помощью unwind» происходит не только переход к корневой сцене, но и передача данных от контроллера Б контроллеру А. Сейчас у вас в запасе есть несколько способов передачи данных между контроллерами, и при этом они не конфликтуют между собой: вы можете осуществлять переходы и передачу данных любыми нажатиями на любые из кнопок в любых комбинациях.

10.5 Передача данных от Б к А с помощью делегирования

Вы уже хорошо знакомы с шаблоном «Делегирование». Напомню, его суть заключается в том, что объект делегирует (передает) ответственность за решение какой-либо задачи другому объекту. Данный подход прекрасно вписывается в контекст решения задачи передачи данных между контроллерами.

На рисунке 10.12 схематично показан принцип обмена данными на основе паттерна «Делегирование». Его суть заключается в том, что контроллер, изменяющий данные (контроллер Б), хранит ссылку на делегата (которым является контроллер А) и после изменения вызывает определенный метод делегата, передавая ему обновленные данные. А дальше, как использовать полученные данные, как с их помощью обновить интерфейс, как их сохранить – это проблема и задача делегата.

При этом наиболее верным подходом является скрытие делегата за протоколом, то есть когда контроллеры взаимодействуют не напрямую, а на основе протокола.

- В составе проекта создайте новый файл **DataUpdate.swift**.
- В созданном файле реализуйте протокол **DataUpdateProtocol** (листинг 10.12).



Рис. 10.12. Схема передачи данных

ЛИСТИНГ 10.12

```
protocol DataUpdateProtocol {
    func onDataUpdate(data: String)
}
```

Протокол **DataUpdateProtocol** содержит требование к наличию метода **onDataUpdate**, который будет вызываться в контроллере А при изменении его данных в контроллере Б.

- Подпишите класс **ViewController** на протокол **DataUpdateProtocol** (листинг 10.13) и реализуйте метод **onDataUpdate**.

ЛИСТИНГ 10.13

```
class ViewController: UIViewController, DataUpdateProtocol {

    // ...

    func onDataUpdate(data: String) {
        updatedData = data
        updateLabel(withText: data)
    }
}
```

Ничего сложного в этом методе нет: при его срабатывании обновленное значение инициализируется свойству **updatedData** и обновляется содержимое текстовой метки.

Теперь необходимо, чтобы один класс стал делегатом другого. Хотя мы говорили об этом выше, как по вашему мнению — кто должен быть чьим делегатом и в каких вопросах? Так как обновление данных происходит на сцене Б, становится очевидно, что контроллер А должен быть делегатом контроллера

Б в вопросах обработки измененных данных. То есть, когда данные изменятся, контроллер Б вызовет метод **onDataUpdate** контроллера А, тем самым передавая ответственность.

- ▶ В классе **SecondViewController** объявите свойство **handleUpdatedDataDelegate** (листинг 10.14).

ЛИСТИНГ 10.14

```
var handleUpdatedDataDelegate: DataUpdateProtocol?
```

Примечание Довольно часто в учебных материалах для хранения делегата используют свойство с именем **delegate**. Но тут стоит заметить, что у контроллера может быть несколько делегатов, каждый из которых решает собственную задачу. По этой причине использование свойства с таким общим именем может привести к путанице.

- ▶ На сцене А создайте кнопку с текстом «**Изменить с помощью делегата**». Вновь, используя констрейнты, разместите ее в 30 точках ниже.
- ▶ Оформите кнопку в соответствии со стилем других кнопок.

При нажатии на кнопку «**Изменить с помощью делегата**» (рис. 10.13) контроллер, с которого происходит переход, будет устанавливаться в качестве делегата контроллера назначения.



Рис. 10.13. Новая кнопка на сцене А

- ▶ В классе **ViewController** реализуйте экшн-метод **editDataWithDelegate** (листинг 10.15).

ЛИСТИНГ 10.15

```
// переход от А к Б
// передача данных с помощью свойства и установка делегата
@IBAction func editDataWithDelegate(_ sender: UIButton) {
    // получаем вью контроллер
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    let editScreen = storyboard.instantiateViewController(withIdentifier:
"SecondViewController") as! SecondViewController
```

```
// передаем данные
editScreen.updatingData = dataLabel.text ?? ""

// устанавливаем текущий класс в качестве делегата
editScreen.handleUpdatedDataDelegate = self

// открываем следующий экран
self.navigationController?.pushViewController(editScreen, animated:
true)
}
```

Тело метода **editDataWithDelegate** практически идентично методу **editDataWithProperty** за тем исключением, что помимо передачи данных в нем также устанавливается ссылка на класс-делегат.

- ▶ Свяжите метод **editDataWithDelegate** с кнопкой «**Изменить с помощью делегата**».

Теперь доработаем контроллер Б, реализовав вызов метода делегата при изменении данных.

- ▶ На сцене Б разместите новую кнопку «**Сохранить с помощью делегата**». Разместите ее ниже в ряд и оформите в соответствии со стилем остальных кнопок (рис. 10.14).

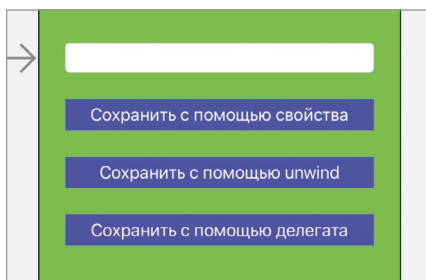


Рис. 10.14. Новая кнопка на сцене Б

- ▶ В классе **SecondViewController** реализуйте экшн-метод **saveDataWithDelegate** (листинг 10.16).

ЛИСТИНГ 10.16

```
// Переход от Б к А
// Передача данных с помощью делегата
@IBAction func saveDataWithDelegate (_ sender: UIButton) {
    // получаем обновленные данные
    let updatedData = dataTextField.text ?? ""
    // вызываем метод делегата
```

```
handleUpdatedDataDelegate?.onDataUpdate(data: updatedData)
// возвращаемся на предыдущий экран
navigationController?.popViewController(animated: true)
}
```

- ▶ Свяжите нажатие кнопки «**Сохранить с помощью делегата**» с вызовом метода **saveDataWithDelegate**.
- ▶ Запустите приложение и попробуйте в действии реализованную функциональность.

Использование делегата – это очень эффективный способ передачи данных между контроллерами. Его самым важным плюсом является то, что контроллеры связываются на основе протокола и вся ответственность за обработку данных ложится на заинтересованный в этом контроллер (А). То есть на самом деле контроллеру Б вообще не важно, что будет с данными – он их обновил и далее передал контроллеру А, мол делай с ними все, что считаешь нужным.

Такой подход снижает связанность контроллеров и позволяет многократно повторно использовать их в различных сценариях. Так, например, мы с легкостью можем указать в качестве делегата другой контроллер, и это не вызовет каких-либо ошибок. Главное, чтобы он соответствовал протоколу.

Также вам стоит обратить внимание на то, что кнопка «**Сохранить с помощью делегата**» приводит к изменению данных только в том случае, если сцена Б была показана по нажатии кнопки «**Изменить с помощью делегата**». Это связано с тем, что в остальных случаях свойство **handleUpdatedDataDelegate** не будет содержать ссылку на делегат.

10.6 Передача данных от Б к А с помощью замыкания

Последним способом передачи данных между контроллерами, который мы реализуем в этой главе, является подход с использованием замыкания. Этот вариант очень похож на использование делегата, но вместо ссылки на делегат в контроллере Б будет храниться замыкание, вызываемое после изменения данных (рис. 10.15). Замыкание способно хранить ссылку на контроллер А, а значит и производить в нем работу по обработке обновленных данных, даже если сцена А в данный момент не видна на экране.

Значительный плюс такого подхода состоит в том, что замыкание может быть изменено в зависимости от контекста использования контроллера Б, в то время, как использование делегата подразумевает вызов одного и того же метода.

- ▶ В классе **SecondViewController** объявите свойство **completionHandler** (листинг 10.17).



Рис. 10.15. Схема обмена данными

ЛИСТИНГ 10.17

```
var completionHandler: ((String) -> Void)?
```

Примечание Для хранения замыканий, вызываемых при завершении какой-либо задачи, очень часто используют параметры с именами **completionHandler**, **completionClosure**, просто **completion** или **handler**. Собственно, «completion handler» с английского переводится, как «обработчик завершения».

Тип замыкания **((String) -> Void)?** подразумевает, что если замыкание существует (значение свойства может быть **nil**), в него будет передано обновленное текстовое значение.

Теперь создадим на сцене А все необходимые для передачи замыкания элементы.

- ▶ На сцене А создайте кнопку с текстом «Изменить с помощью замыкания». Вновь, используя констрейнты, разместите ее в 30 точках ниже.
- ▶ Оформите кнопку в соответствии со стилем других кнопок.

При нажатии на кнопку «Изменить с помощью замыкания» (рис. 10.16) контроллер А будет передавать данные для изменения, а также инициализировать значение замыкания.



Рис. 10.16. Новая кнопка на сцене А

- ▶ В классе **ViewController** реализуйте экшн-метод **editDataWithClosure** (листинг 10.18).

ЛИСТИНГ 10.18

```
// переход от А к Б
// передача данных с помощью свойства и инициализация замыкания
@IBAction func editDataWithClosure(_ sender: UIButton) {
    // получаем вью контроллер
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    let editScreen = storyboard.instantiateViewController(withIdentifier:
"SecondViewController") as! SecondViewController

    // передаем данные
    editScreen.updatingData = dataLabel.text ?? ""

    // передаем необходимое замыкание
    editScreen.completionHandler = { [unowned self] updatedValue in
        updatedData = updatedValue
        updateLabel(withText: updatedValue)
    }

    // открываем следующий экран
    self.navigationController?.pushViewController(editScreen, animated:
true)
}
```

Тело метода **editDataWithClosure** очень похоже на метод **editDataWithProperty** за исключением передачи замыкания.

- ▶ Свяжите метод **editDataWithClosure** с кнопкой «Изменить с помощью замыкания».

Теперь доработаем контроллер Б, реализовав вызов замыкания при изменении данных.

- ▶ На сцене Б разместите новую кнопку «Сохранить с помощью замыкания». Разместите ее ниже в ряд и оформите в соответствии со стилем остальных кнопок (рис. 10.17).
- ▶ В классе **SecondViewController** реализуйте экшн-метод **saveDataWithClosure** (листинг 10.19).

ЛИСТИНГ 10.19

```
// Переход от Б к А
// Передача данных с помощью замыкания
```

```
@IBAction func saveDataWithClosure(_ sender: UIButton) {  
    // получаем обновленные данные  
    let updatedData = dataTextField.text ?? ""  
    // вызываем замыкание  
    completionHandler?(updatedData)  
    // возвращаемся на предыдущий экран  
    navigationController?.popViewController(animated: true)  
}
```

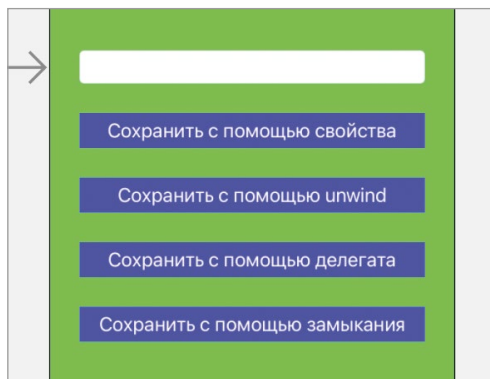


Рис. 10.17. Новая кнопка на сцене Б.

- ▶ Свяжите нажатие кнопки **«Сохранить с помощью замыкания»** с вызовом метода **saveDataWithClosure**.
- ▶ Запустите приложение и попробуйте в действии реализованную функциональность.

При использовании кнопки **«Сохранить с помощью замыкания»** данные успешно передаются в контроллер А только при условии, что переход к сцене Б был произведен с помощью кнопки **«Изменить с помощью замыкания»**. В ином случае, в свойстве **completionHandler** находится **nil**.

10.7 Другие способы передачи данных

В этой главе мы рассмотрели несколько способов передачи данных внутри приложения, которые вы сможете использовать в своих будущих проектах. Но это далеко не все варианты, что может предложить вам Swift. Чем больший практический и теоретический опыт вы будете получать, тем больше новых способов работы с данными узнаете. В этом разделе я хотел бы упомянуть еще несколько важных моментов, связанных с передачей данных.

Использование ссылочных типов при передаче данных

В рассматриваемом материале для работы с данными мы используем тип данных **String**, который является значимым (value type). Но что, если вместо этого использовать ссылочный тип, например, такой:

```
class AppData {  
    var data: String  
    init(data: String) {  
        self.data = data  
    }  
}
```

Данные из контроллера А в контроллер Б будут передаваться одним из рассмотренных способов, но обратной передачи не потребуется, так как оба контроллера будут ссылаться на одно и то же значение (рис. 10.18). Будет достаточно лишь обновить значение в свойстве контроллера Б, а при возвращении к сцене А обновить интерфейс с помощью метода **viewWillAppear**.



Рис. 10.18. Использование ссылочного типа данных

Работа с общим хранилищем

Одним из вариантов совместно работы с данными является использование различных хранилищ, например, User Defaults, когда каждый контроллер может самостоятельно подгружать из хранилища и обновлять требуемые ему данные. Нужно быть крайне осторожным, используя такой подход, так как в этом случае растет количество точек взаимодействия с хранилищем. Изменения данных в хранилище не всегда будут явными, а значит отслеживать их будет гораздо сложнее.

Использование паттерна «Одиночка»

Еще одним вариантом, который часто используют программисты, является применение шаблона проектирования «Одиночка» (Singleton). Напомню, что данный шаблон подразумевает единую точку входа в класс и один экземпляр этого класса на все приложение:

```
class User {  
    static shared = User()  
    var id: Int = 0  
    init() {  
        // ...  
    }  
}
```

Среди многих программистов данный шаблон считается антипаттерном. Его критикуют и не любят. Но лично я отношусь к нему более сдержанно и использую его в тех случаях, когда работа приложения очень тесно связана с экземпляром данного типа, и растрата памяти на его постоянное хранение оправдана.

Использование экземпляра класса AppDelegate

Иногда у вас может возникнуть желание хранить данные в классе **AppDelegate**, создав в нем несколько свойств и проинициализировав в них значения. Старайтесь избегать такого подхода. В задачи класса **AppDelegate** не входит обеспечение хранения — он предназначен для выполнения совершенно других задач.

Использование координаторов

Еще одним вариантом организации, о котором я хотел бы упомянуть, является использование координаторов. Я применял такой подход при разработке приложения «**Subs Tracker – Трекер подписок и регулярных платежей**». Это был учебный проект, созданный для изучения паттерна MVC, но с одним важным дополнением: в проекте используются так называемые координаторы.

Примечание Координаторы – это вовсе не мое изобретение. Они очень активно используются при разработке приложения командой Авито. На YouTube вы можете найти несколько прекрасных докладов по этой теме.

Координатор – это сущность, управляющая отображением сцен в приложении, а также обеспечивающая передачу и распространение данных внутри него. И таких координаторов в приложении может (и должно) быть несколько. Связь координаторов обычно представлена в виде дерева, где каждый координатор управляет собственным множеством сцен. Причем одни и те же сцены могут использоваться в различных координаторах (рис. 10.19).



Рис. 10.19. Пример структуры координаторов и сцен

Для чего нужны координаторы?

Они сводят к минимуму связанность контроллеров, позволяя сделать их по-настоящему независимыми и переиспользуемыми. Каждый координатор предназначен для решения собственной задачи: для авторизации и регистрации, для управления профилем, для работы с товарными позициями и т.д. Примеров может быть бесконечное множество. Каждый координатор работает с большим количеством сцен. Но при этом они ничего не знают друг о друге – все взаимодействие между ними происходит через координаторы.

Если приложению требуется отобразить новую сцену, например, перейти от экрана авторизации к основному экрану со списком товаров, то решение этого

вопроса ложится на плечи координаторов, но не вью контроллеров. Если вам требуется распространить измененные данные (например, о добавленном в корзину товаре) в приложении, для этого также используются координаторы.

Координатор с программной точки зрения представляет из себя класс, подписанный на один или несколько специальных протоколов. Протоколы наделяют класс некой стандартной функциональностью, позволяющей работать с контроллерами и обмениваться данными.

Примечание Конкретные реализации данных протоколов вы сможете найти в моем GitHub в исходном коде приложения Subs Tracker.

На этом мы завершаем изучение вопроса обмена данными между контроллерами внутри ваших приложений. В будущем вы неоднократно вернетесь к этой главе, так что советую оставить закладку.

Глава 11.

Контроллер табличного представления.

Класс UITableViewController.

В этой главе вы:

- познакомитесь с контроллером табличного представления (Table View Controller) и классом **UITableViewController**;
- узнаете, чем контроллер табличного представления отличается от уже знакомого вам табличного представления (Table View).

Табличное представление (Table View) – очень мощный инструмент визуального отображения данных, который используется разработчиками в большинстве приложений. В этой главе мы познакомимся с **контроллером табличного представления** (Table View Controller) и на протяжении нескольких глав будем заниматься разработкой приложения «**To-Do Manager**» («Менеджер задач») (рис. 11.1), основанного на данном типе контроллера.

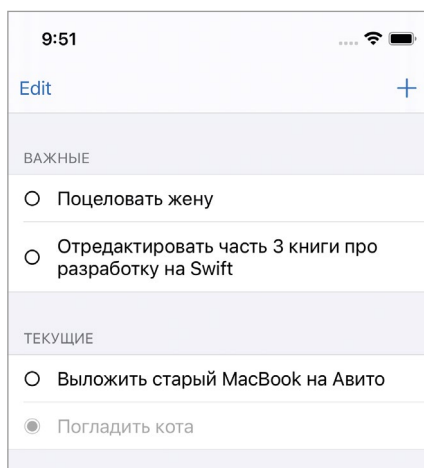


Рис. 11.1. Интерфейс приложения To-Do Manager

11.1 Создание проекта на основе Table View Controller

Приложение «**To-Do Manager**» будет типичным менеджером задач, в котором пользователь сможет вести учет текущих и выполненных задач (дел). Для вывода списка задач будет использоваться табличное представление, при этом создание и редактирование новых элементов будет производиться на отдельном экране (рис. 11.2). Для каждой задачи можно определить ее тип («текущая» или «важная»), от этого будет зависеть место ее отображения в общем списке, а также состояние («запланирована» или «выполнена»).

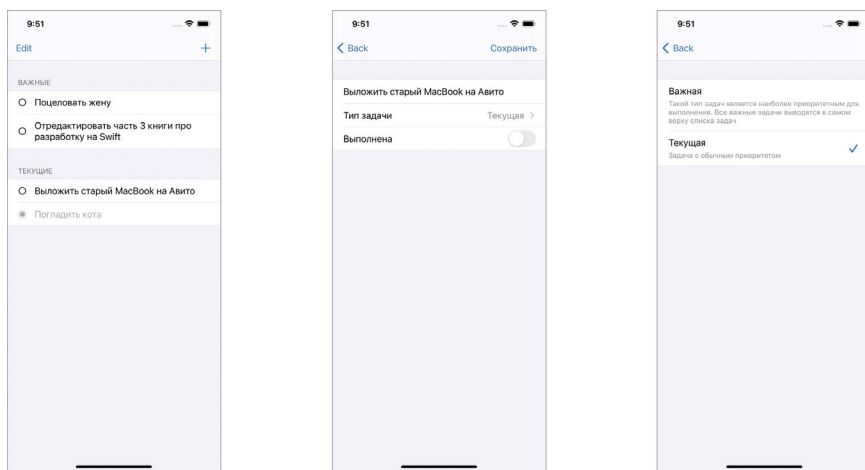


Рис. 11.2. Рабочие экраны приложения

Разработка любого приложения начинается с создания нового проекта в Xcode.

- Создайте проект с названием «**To-Do Manager**». В качестве шаблона выберите **App**. При определении параметров будущего проекта не забудьте снять галочки с **Use Core Data**, **Include Tests**.

Создаваемое приложение будет основано на контроллере табличного представления.

Контроллер табличного представления – это элемент, объединяющий в себе видо контроллер, табличное представление, делегат и источник данных табличного представления. То есть, с одной стороны – это обычный контроллер, но при работе с ним вам не требуется думать о размещении табличного представления, так как оно уже включено в его состав. При работе на сториборде табличное представление занимает все доступное на сцене пространство (рис. 11.3), и при этом класс, связанный с контроллером, уже является делегатом (delegate) и источником данных (data source) табличного представления.

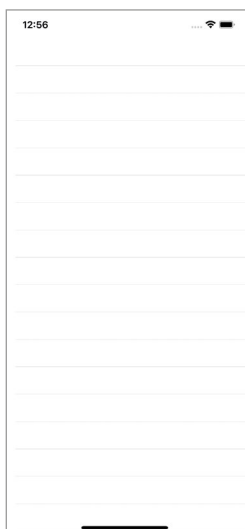


Рис. 11.3. Сцена на основе Контроллера табличного представления

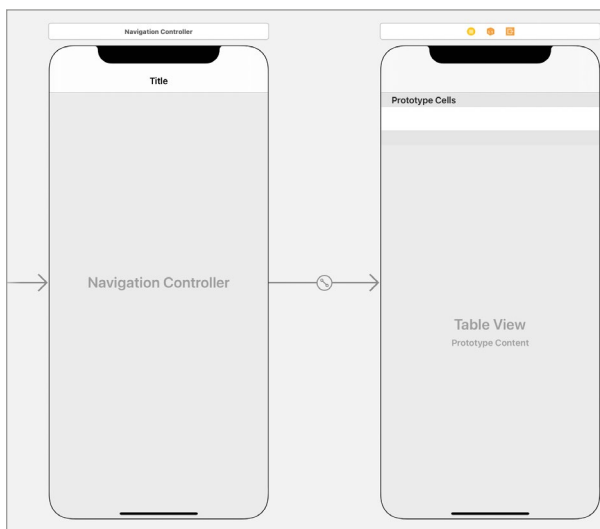


Рис. 11.4. Два контроллера на storyboard

Разместим на storyboard контроллер табличного представления.

- ▶ Откройте файл **Main.storyboard**.
- ▶ Удалите со сториборда созданный по умолчанию **View Controller**.
- ▶ Найдите в библиотеке объектов элемент **Table View Controller** и разместите его на сцене.
- ▶ Установите контроллер в качестве стартового (пункт **Is Initial View Controller**).
- ▶ Оберните контроллер в **Navigation Controller**.

На рисунке 11.4 показан пример того, что должно у вас получиться в результате выполненных действий.

Фреймворк **UIKit** содержит большое количество различных графических элементов, включая несколько типов вью контроллеров. Два из них (навигационный и табличный) уже используются в нашем проекте.

11.2 Класс UITableViewController

Каждому элементу в составе **UIKit** соответствует специальный класс, исключением не стал и контроллер табличного представления. Он представлен классом **UITableViewController**. Для того, чтобы управлять сценой на основе

данного контроллера, необходимо создать кастомный класс, наследуемый от **UITableViewController**.

- ▶ Удалите файл **ViewController.swift**.
- ▶ Создайте новый файл типа **Cocoa Touch Class**, содержащий класс **TaskListController**, потомок **UITableViewController** (рис. 11.5).

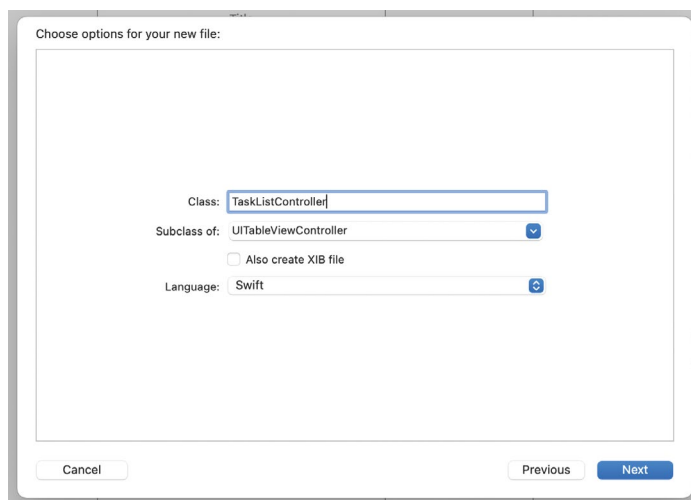


Рис. 11.5. Создание нового файла

Свяжите **Table View Controller**, размещенный на сториборде, с классом **TaskListController**.

- ▶ Откройте файл **TaskListController.swift**.

По умолчанию класс **TaskListController** содержит довольно много кода, среди которого вы можете увидеть уже знакомые вам методы, относящиеся к табличному представлению. Это не удивительно, так как (об этом уже было сказано ранее) Table View Controller включает в себя табличное представление и является его делегатом и источником данных одновременно.

Класс **UITableViewController** позволяет организовать полный контроль над табличным представлением, входящим в его состав. Для того, чтобы получить доступ к табличному представлению, в теле класса необходимо обратиться к свойству **tableView**.

Посмотрим на исходный код класса **UITableViewController**.

- ▶ Нажмите клавишу **Command** и щелкните по имени класса **UITableViewController**.
- ▶ В открывшемся окне выберите пункт **Jump to Definition** (рис. 11.6).

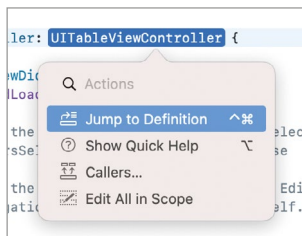


Рис. 11.6. Переход к объявлению класса

С помощью описанной функции мы осуществили переход к коду, объявляющему класс **UITableViewController** внутри фреймворка **UIKit**.

Как видно из кода класса изначально подписан на протоколы **UITableViewDelegate** и **UITableViewDataSource** (рис 11.7), а значит мы можем использовать в классе **TaskListController** методы этих протоколов.

```

15 @available(iOS 2.0, *)
16 open class UITableViewController : UIViewController, UITableViewDelegate, UITableViewDataSource {
17
18
19     public init(style: UITableView.Style)
20
21     public init(nibName nibNameOrNil: String?, bundle nibBundleOrNil: Bundle?)
22
23     public init?(coder: NSCoder)
24
25
26     open var tableView: UITableView!
27
28     @available(iOS 3.2, *)
29     open var clearsSelectionOnViewWillAppear: Bool // defaults to YES. If YES, any selection is cleared
30
31
32     @available(iOS 6.0, *)
33     open var refreshControl: UIRefreshControl?
34 }
35

```

Рис. 11.7. Соответствие класса протоколам

Table View Controller – это очень мощный инструмент, и, вполне возможно, вы будете работать с ним куда чаще, чем отдельно с View Controller и Table View. Используя данный элемент, все необходимое для наполнения таблицы находится у вас под рукой, вам не нужно думать о размещении таблицы, создании связей, определении делегата и источника данных. Это очень удобно.

11.3 Разработка прототипа Модели

Основной задачей этой главы является создание базы для будущего приложения. С целью дальнейшего изучения материала нам потребуется создать некий прототип Модели, основными задачами которого являются описание основных сущностей и обеспечение доступа к тестовому набору данных. Все остальные функции (загрузка и запись данных из/в долговременное хранилище) будут реализованы в последней главе данной части книги.

Основной сущностью, используемой в приложении, будет «Задача». Именно вокруг нее будет крутиться вся бизнес-логика. В составе данной сущности может быть выделен ряд атрибутов:

1. Название – текстовое описание задачи.
2. Тип/приоритет – указатель на важность задачи. Может иметь одно из следующих значений: «текущая» или «важная».
3. Статус – определяет текущее состояние задачи. Может иметь одно из следующих значений: «запланированная» или «выполненная».

Значения данных атрибутов будут использоваться для наполнения табличного представления данными.

Теперь поговорим о программной реализации сущности. В первую очередь создадим файл для программного кода.

- ▶ В составе проекта создайте папку с названием **Model**.
- ▶ В папке **Model** создайте файл **Task.swift**.

Если для определения названия можно использовать тип **String**, то для типа и статуса потребуются реализовать специальные типы, способные описать все возможные состояния этих атрибутов.

- ▶ Добавьте в файл **Task.swift** код из листинга 11.1.

ЛИСТИНГ 11.1

```
// тип задачи
enum TaskPriority {
    // текущая
    case normal
    // важная
    case important
}

// состояние задачи
enum TaskStatus {
    // запланированная
    case planned
    // завершенная
    case completed
}
```

Перечисления являются прекрасным решением в ситуациях, когда необходимо дать выбор из конечного множества значений, что и требуется в ситуации с типом и статусом задачи. Возможен и другой вариант реализации, когда вме-

сто типов **TaskPriority** и **TaskStatus** мы бы использовали логические параметры, например, **isImportant** и **isCompleted**. Но такой подход ограничивает нас в доступных возможностях в том случае, если вдруг будет принято решение расширить доступные варианты значений. Логический тип подразумевает выбор из двух значений: **true** и **false**.

Теперь перейдем к непосредственной реализации сущности «Задача». В первую очередь реализуем протокол, и далее конкретный тип на его основе.

► Добавьте в файл **Task.swift** код из листинга 11.2.

ЛИСТИНГ 11.2

```
// требования к типу, описывающему сущность "Задача"
protocol TaskProtocol {
    // название
    var title: String { get set }
    // тип
    var type: TaskPriority { get set }
    // статус
    var status: TaskStatus { get set }
}

// сущность "Задача"
struct Task: TaskProtocol {
    var title: String
    var type: TaskPriority
    var status: TaskStatus
}
```

«Задача» готова, теперь реализуем тип, обеспечивающий его хранение. На данном этапе мы не будем тратить время на создание всей функциональности хранилища, ограничимся лишь загрузкой тестовых данных. Тем не менее, в протоколе определим все методы, которые могут нам потребоваться в дальнейшем, а их конечной реализацией займемся позже.

► В папке **Model** создайте файл **TasksStorage.swift**.

► В **TasksStorage.swift** добавьте код из листинга 11.3.

ЛИСТИНГ 11.3

```
// Протокол, описывающий сущность "Хранилище задач"
protocol TasksStorageProtocol {
    func loadTasks() -> [TaskProtocol]
    func saveTasks(_ tasks: [TaskProtocol])
}
```

```
// Сущность "Хранилище задач"
class TasksStorage: TasksStorageProtocol {
    func loadTasks() -> [TaskProtocol] {
        // временная реализация, возвращающая тестовую коллекцию задач
        let testTasks: [TaskProtocol] = [
            Task(title: "Купить хлеб", type: .normal, status: .planned),
            Task(title: "Помыть кота", type: .important, status: .planned),
            Task(title: "Отдать долг Арнольду", type: .important, status:
.completed),
            Task(title: "Купить новый пылесос", type: .normal, status:
.completed),
            Task(title: "Подарить цветы супруге", type: .important, status:
.planned),
            Task(title: "Позвонить родителям", type: .important, status:
.planned)
        ]
        return testTasks
    }

    func saveTasks(_ tasks: [TaskProtocol]) {}
}
```

Примечание Убедитесь, что название каждой задачи содержит не более двух или трех слов. Это позволит им занять не более одной строки в текстовой метке в ячейке, а также не приведет к сбоям в отображении списка задач.

Далее мы рассмотрим, что делать в случае, когда задача содержит много текста.

Как отмечалось ранее, в текущем варианте класса **TasksStorage** отсутствует реализация работы с хранилищем: метод **loadTasks** возвращает тестовый набор данных, а в теле **saveTasks** вообще отсутствует код.

На этом мы завершаем работу над созданием базовой части программы, а в следующих главах приступим к рассмотрению новых способов создания ячеек табличных представлений.

Глава 12.

Табличные представления на основе прототипов ячеек

В этой главе вы:

- узнаете, что такое прототипы ячеек;
- научитесь использовать прототипы ячеек для наполнения табличных представлений данными;
- познакомитесь с UI-элементом Horizontal Stack;
- узнаете, как устранять проблемы позиционирования графических элементов.

Ранее для наполнения табличных представлений данными мы использовали доступные по умолчанию варианты ячеек со строго определенным стилем оформления. Такой подход довольно удобен и не требует объемного кода, но ограничивает разработчика в возможностях, так как он подходит далеко не во всех ситуациях. В каждом отдельном случае необходимо с умом подходить к разработке дизайна, и создавать такие варианты интерфейса, которые позволят получать положительный **пользовательский опыт**.

Примечание Пользовательский опыт (User Experience, сокращенно UX) – это, грубо говоря, впечатления, которые получает пользователь при работе с программой. На UX влияют многие элементы, включая удобство работы приложения, его внешний вид, компоновку элементов, скорость выполнения задач, отзывчивость и плавность интерфейса, доступные функции и т.д.

12.1 Прототипы ячеек

Xcode и Swift предоставляют вам все необходимое для создания удобного пользовательского интерфейса, совершенно не ограничивая ваши возможности. В случае с ячейками табличных представлений, когда стандартных стилей оформления вам недостаточно, вы можете создать собственные кастомные ва-

рианты оформления и компоновки элементов. Одним из способов сделать это является использование **прототипов ячеек**.

Прототип ячейки – это *переиспользуемый* шаблон, на основе которого создается ячейка. Прототип создается с помощью **Interface Builder** прямо на сцене. В приложении «**To-Do Manager**» прототипы будут использоваться на главном экране для отображения ячеек в списках задач.

12.2 Создание прототипов ячеек

Для создания прототипов используется **Interface Builder** и непосредственно само табличное представление, размещенное на сцене.

- ▶ Откройте проект «**To-Do Manager**», созданный в предыдущей главе.
- ▶ Перейдите к файлу **Main.storyboard**.

Табличное представление, размещенное в контроллере **TaskListController**, по умолчанию уже содержит один прототип. Визуально вы можете увидеть его ниже надписи **Prototype Cells**. Изначально он пуст, т.е. в его составе нет никаких графических элементов, кроме корневого представления (рис. 12.1). Также данный прототип можно найти в структуре сцены на панели **Document Outline** (рис. 12.2).

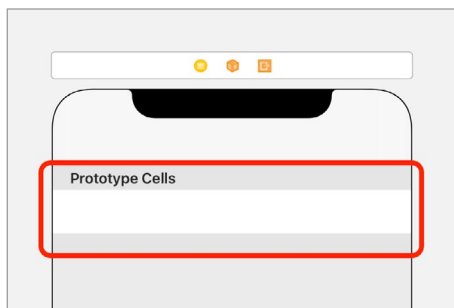


Рис. 12.1. Прототип ячейки в составе табличного представления на storyboard

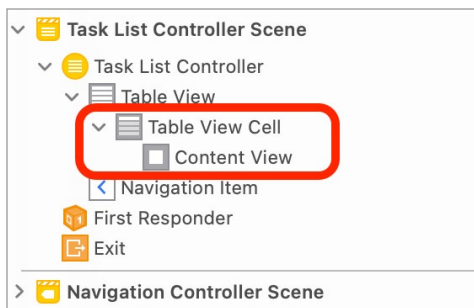


Рис. 12.2. Прототип ячейки в составе табличного представления на панели Document Outline

Примечание Наличие прототипа ячейки не обязывает вас его использовать и никак не влияет на работу табличного представления. Каждый созданный прототип может при необходимости использоваться в процессе жизненного цикла таблицы.

Прототип – это *локальный* для каждой конкретной таблицы элемент. То есть, прототип, созданный в одном табличном представлении, не может быть использован в другом.

Примечание О том, как создавать универсальные ячейки, доступные для использования в любом табличном представлении в рамках приложения, мы поговорим в следующих главах.

Табличные представления могут содержать произвольное количество прототипов, каждый из которых может быть оформлен и использован независимо от остальных. На рисунке 12.3 показан пример сцены на storyboard с пятью прототипами, созданными в рамках одного тейбл вью.

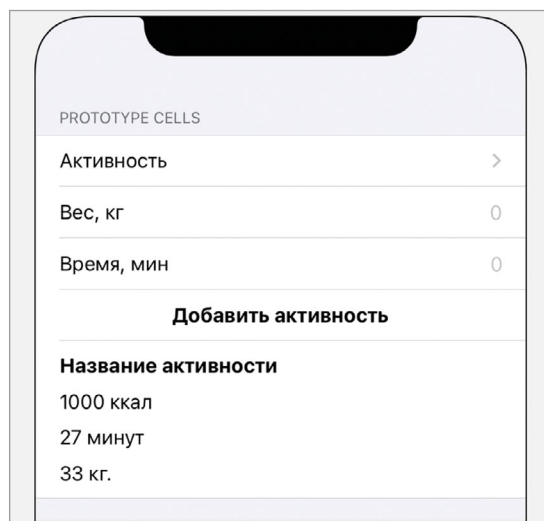


Рис. 12.3. Несколько прототипов ячеек в одной таблице

Примечание Обратите внимание, что на рисунках 12.1 и 12.3 стиль написания текста **Prototype Cells** отличается из-за различного стиля оформления **Table View**. Стиль оформления можно выбрать на панели **Attributes Inspector** в поле **Style**.

С помощью панели **Attributes Inspector** вы можете изменить количество прототипов в табличном представлении.

- ▶ Выделите Table View на сцене (для этого вы можете использовать **Document Outline**).
- ▶ Откройте панель **Attributes Inspector**.
- ▶ Измените значение поля **Prototype Cells** на 3.

Теперь таблица содержит 3 прототипа, каждый из которых может быть использован независимо от других. При этом не имеет никакого значения порядок прототипов. Каждый прототип – это отдельный и независимый шаблон ячейки, который может быть использован для создания ячеек для любых строк таблицы. Всю логику вывода ячеек вы задаете с помощью программного кода, а прототип лишь определяет внешний вид таблицы.

Прототипы в проекте To-Do Manager

Первый тип ячеек, который нам необходимо реализовать в проекте, будет предназначен для отображения задач в их общем списке на главном экране приложения. Воспользуемся прототипом ячеек.

Для того, чтобы в ячейке таблицы отображалась вся необходимая информация о задаче, разместим в прототипе две текстовые метки (рис. 12.4). Левая метка будет отображать специальный символ, указывающий на статус выполнения задачи, а правая – выводить название задачи. При этом, как текущие, так и выполненные задачи, будут отображаться на основе одного и того же прототипа, поскольку они имеют одинаковую структуру и расположение элементов.

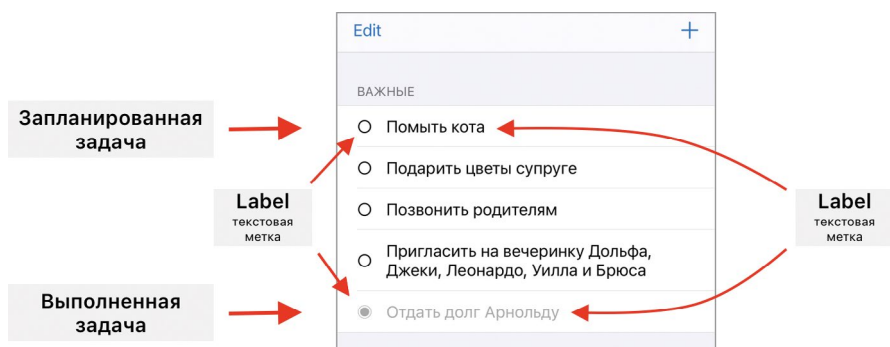


Рис. 12.4. Структура ячеек

При создании прототипа нам потребуется ответить на несколько вопросов.

1. Каким способом позиционировать элементы внутри ячейки?

В процессе работы над сценой мы рассмотрим два способа: с использованием констрейнтов и с помощью горизонтального стека (Horizontal Stack View). Оба варианта в результате приведут к одному и тому же результату, но такой подход позволит вам получить новый опыт в верстке интерфейса.

2. Каким способом осуществлять доступ к элементам ячейки (текстовым меткам) для изменения их содержимого?

Мы вновь рассмотрим два способа: с помощью тегов и с помощью кастомного класса с аутлетами.

Для рассмотрения материала нам потребуется создать два прототипа, в каждом из которых мы используем свой способ позиционирования и доступа.

► Измените значение поля **Prototype Cells** на 2.

Теперь в составе таблицы отображаются два пустых прототипа. Верхний будет использоваться при работе с констрейнтами и тегами, а нижний – при работе с горизонтальным стеком и кастомным классом.

12.3 Создание прототипа с использованием констрейнтов и тегов

В первую очередь разместим в прототипе текстовые метки.

- ▶ Откройте библиотеку объектов.
- ▶ Найдите элемент **Label** и разместите его в левой части прототипа.
- ▶ Дополнительно расположите второй **Label** в правой части прототипа.

На рисунке 12.5 показан примерный результат выполнения описанных действий.

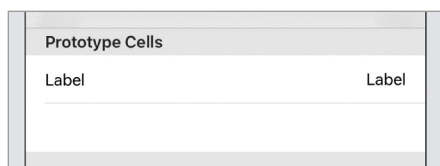


Рис. 12.5. Размещение текстовых меток в прототипе

Для того, чтобы элементы корректно отображались на любых устройствах, необходимо задать правила их размещения.

- ▶ Выделите обе текстовые метки (используйте при этом клавишу **Command**).

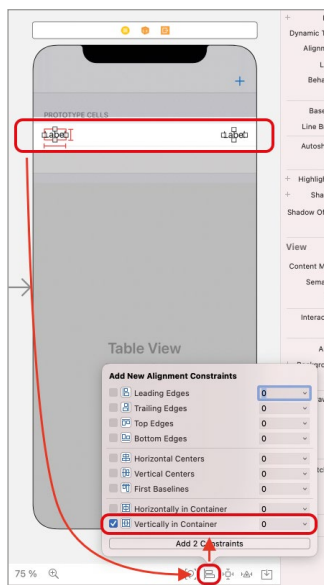


Рис. 12.6. Создание ограничений

- ▶ Создайте констрейнт, обеспечивающий центрирование текстовых меток по вертикали (рис. 12.6). Не забудьте нажать на кнопку **Add 2 Constraints**.

Теперь текстовые метки будут всегда находиться в центре вертикальной оси, независимо от размеров ячейки.

- ▶ Для метки, расположенной в левой части ячейки, создайте ограничения на отступ в 0 точек слева, сверху и снизу, и в 10 точек – справа (рис. 12.7). При этом убедитесь, что отмечен пункт **Constrain to margins**.

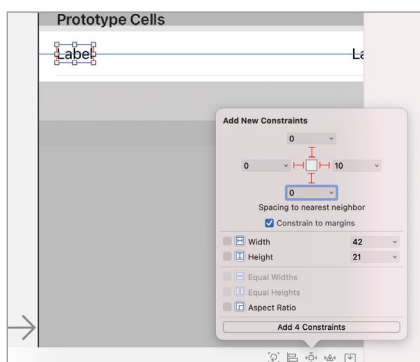


Рис. 12.7. Создание ограничений

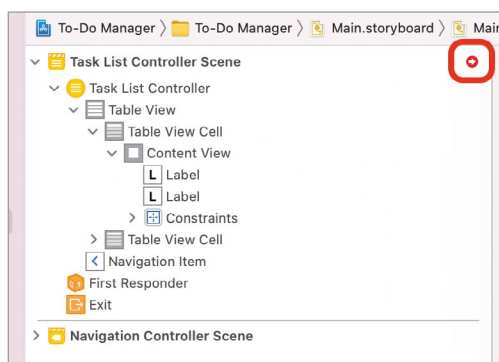


Рис. 12.8. Метка, сообщающая о наличии критических проблем позиционирования

Пункт **Constrain to margins** определяет, будет ли размещен элемент с учетом или без учета внутренних отступов. В нашем случае, так как **Constrain to margins** был активирован, текстовая метка разместится не вплотную к краям ячейки, а будет иметь небольшой отступ (margin).

При создании констрейнта для левой метки на отступ справа также автоматически был создан констрейнт и для правой метки, определяющий ее отступ слева. Поэтому для правой метки потребуется создать лишь три констрейнта: сверху, справа и снизу.

- ▶ Для текстовой метки, расположенной справа, создайте констрейнты на отступ в 0 точек сверху, справа и снизу. Убедитесь, что отмечен пункт **Constrain to margins**.

Несмотря на то, что все необходимые ограничения созданы, в **Document Outline** отображается значок, сообщающий о наличии проблем в позиционировании элементов (рис. 12.8).

- ▶ Нажмите на красный кружок со стрелкой.

Перед вами открылся список, содержащий подробное описание возникших проблем. В списке отображаются два типа проблем (рис. 12.9): **Missing**

Constraints сообщает об отсутствии констрейнтов на ширину и координату по оси X для текстовых меток, а **Content Priority Ambiguity** – на необходимость изменить некий **Horizontal hugging priority** до 252 для одной из меток.

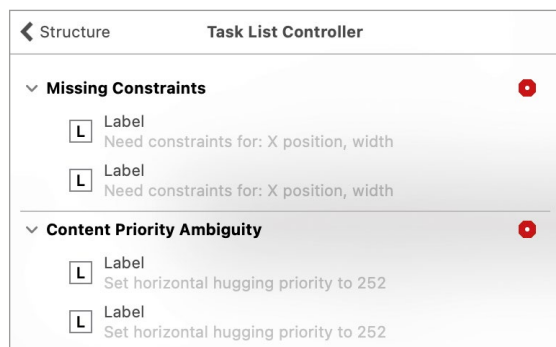


Рис. 12.9. Список критических ошибок позиционирования

Решение возникших проблем мы можем доверить Xcode, нажав на один из красных кружков, но лучше разберемся с этим вопросом и решим его самостоятельно.

По нашей задумке правая метка должна быть смещена влево и находиться в 10 точках от левой текстовой метки (рис. 12.10). При этом для обеих меток созданы одинаковые ограничения (отступы от краев ячейки и от соседней метки одинаковы). Какая метка в этом случае должна растянуться в сторону соседа, однозначно не скажешь, так как каждая из них имеет равные шансы.

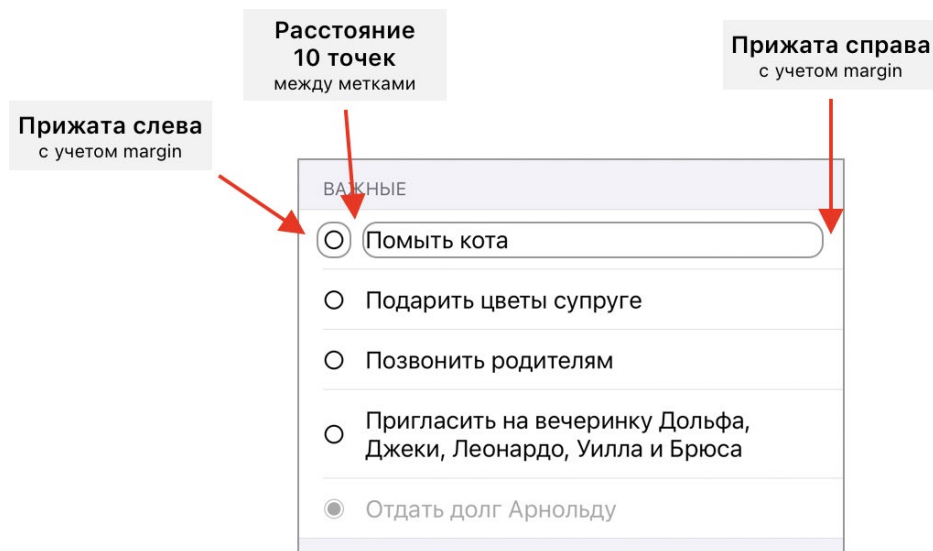


Рис. 12.10. Правила размещения элементов в ячейках

Первая группа ошибок (раздел **Missing Constraints**) сообщает как раз о том, что не выполняется ограничение, определяющее расстояние между метками. И для решения этой задачи мы можем сместить одну из меток, увеличив ее ширину, или задать координату по оси X. Но такой способ решения проблемы создаст больше проблем, чем решит их.

Дело в том, что каждый графический элемент имеет особый приоритет (**Content Hugging Priority**), определяющий, насколько данный элемент сопротивляется (или препятствует) увеличению своего размера. Иными словами, этот приоритет показывает, насколько элемент не хочет увеличиваться. **Content Hugging Priority** – это обычное число, и самое интересное в том, что по умолчанию данный приоритет одинаков у обеих текстовых меток.

- Выделите левую метку.
- Откройте панель **Size Inspector**.

В нижней части панели вы увидите значение приоритета **Content Hugging Priority** для вертикального и горизонтального расширения для левой текстовой метки (рис. 12.11).

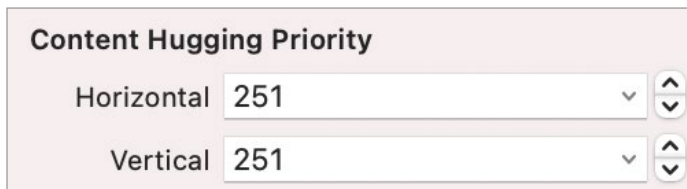


Рис. 12.11. Content Hugging Priority

- Посмотрите данные коэффициенты для правой текстовой метки.

Чем больше значение приоритета, тем больший вес он имеет при позиционировании элементов. То есть, чем выше числовое значение, тем сильнее (относительно других) данный элемент будет сопротивляться расширению.

Примечание Значение приоритетов могут изменяться в диапазоне от 1 до 1000.

Теперь вернемся к нашей задаче. Две текстовые метки находятся на горизонтальной оси. Система позиционирования пытается решить, какая из меток должна расширяться, а какая останется прежнего размера. Обе метки имеют один и тот же приоритет сопротивления по горизонтали, а значит нельзя однозначно сказать, какая из меток должна изменить свой размер и растянуться в сторону другой.

Именно об этом и сообщает вторая группа ошибок (раздел **Content Priority Ambiguity**). Xcode предлагает изменить приоритет одной из меток с 251 на 252, тем самым повысив его. В этом случае метка с меньшим приоритетом сможет увеличить свой размер, заняв все доступное пространство.

- Для правой метки измените значение поля **horizontal** в разделе **Content Hugging Priority** с 251 на 250.

После проделанного действия правая метка расширится в сторону левой, так как теперь она имеет меньший приоритет сопротивления, а значит меньше сопротивляется расширению (рис. 12.12). В результате мы получим то, что хотели: правая метка растянулась влево, а между метками осталось расстояние в 10 точек.

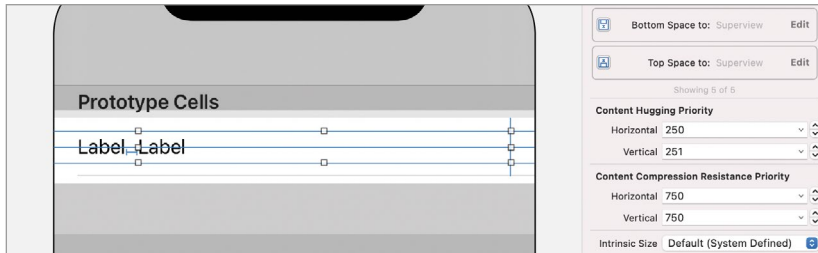


Рис. 12.12. Прототип после изменения приоритета сопротивления

Примечание Как вы могли заметить, на панели **Size Inspector** также присутствует **Content Compression Resistance Priority**. Он прямо противоположен **Content Hugging Priority**, так как определяет, насколько сильно элемент будет сопротивляться уменьшению своего размера, или другими словами, насколько он не хочет уменьшаться.

Описанные концепции могут оказаться слегка сложными для начинающего разработчика. Поэтому остановитесь на минутку и постарайтесь максимально осмыслить их.

Для того, чтобы созданный прототип мог быть многократно использованным при отображении строк таблицы, необходимо указать строковый идентификатор, по которому он может быть получен.

- Выделите созданный прототип ячейки. Будьте внимательны: по неосторожности вы можете выделить либо всю таблицу, либо один из вложенных в ячейку элементов.
- Откройте панель **Attributes Inspector**.
- В поле **Identifier** укажите **taskCellConstraints**.

После изменения идентификатора данное значение отобразится и в **Document Outline** в элементе, соответствующем ячейке. По данному идентификатору в дальнейшем будет обеспечено получение экземпляра ячейки, созданной на основе прототипа.

Связь прототипа с кодом

Визуальная часть прототипа готова: элементы находятся на требуемых позициях. Осталось лишь решить вопрос организации доступа к текстовым меткам для изменения текста, размещенного в них.

Одним из важных свойств любого вью является наличие тегов (tag) – специальных числовых идентификаторов, по которым представление может быть найдено среди всей иерархии представлений. Задав тег у вас появляется возможность вызвать метод **viewWithTag(tag:)**, передав в него заданное значение, и в результате получить ссылку именно на тот графический элемент, у которого указан данный тег.

При использовании данного метода происходит автоматический перебор представлений на всех уровнях вложенности. Таким образом, даже если элемент находится глубоко в иерархии, он будет найден, а ссылка на него возвращена.

Метод **viewWithTag** входит в состав класса **UIView**. В нашем случае мы будем вызывать его для корневого вью ячейки, чтобы обойти всю иерархию представлений. Важно следить за тем, чтобы случайно в иерархии не было определено два одинаковых тега.

- ▶ Выделите левую текстовую метку.
- ▶ Откройте панель **Attributes Inspector**.
- ▶ В разделе **View** в поле **Tag** укажите значение **1**.

Теперь для левой метки определен тег **1**.

- ▶ Аналогичным образом для правой текстовой метки укажите тег **2**.

Метод **viewWithTag** возвращает значение типа **UIView**, а значит нам потребуется использовать тайпкастинг, чтобы привести его к типу **UILabel**, например:

```
view.viewWithTag(tag:1) as? UILabel
```

На этом мы завершаем разработку прототипа с использованием констрейнтов и тегов. Далее в главе мы будем использовать его для наполнения таблицы данными.

12.4 Создание прототипа с использованием Horizontal Stack View и кастомного класса

В состав **UIKit** входит большое количество графических элементов. Ранее для их позиционирования на сцене мы создавали констрейнты, определяя требования к отступам элементов друг от друга и к их размерам. Но если перед вами стоит задача размещения элементов в ряд или в столбец, можно пойти другим путем и воспользоваться специальными группирующими элементами **Horizontal Stack View** и **Vertical Stack View**. Их отличительной особенностью является то, что они предназначены не для отображения каких-либо конкретных данных (например, текста или картинки), а для удобного *адаптивного*

группирования других графических элементов внутри себя. Под адаптивным группированием я подразумеваю, что содержимое стека будет подстраиваться под текущие размеры самого стека (а они могут изменяться в зависимости от размера экрана устройства).

В этом разделе мы поработаем с графическим элементом **Horizontal Stack View**, создав с его помощью еще один прототип. Оба прототипа (тот, что был создан ранее, и тот, что мы создадим сейчас) будут выглядеть совершенно одинаково и выполнять одну и ту же работу – отображать задачи в табличном представлении.

Примечание Все, что будет сказано про Horizontal Stack View будет актуально и для Vertical Stack View, за тем исключением, что второй группирует элементы в вертикальную последовательность, а не в горизонтальную.

Разместим горизонтальный стек на сцене.

- ▶ Откройте библиотеку объектов.
- ▶ Найдите элемент **Horizontal Stack View** и разместите его во втором прототипе.

Горизонтальный стек не имеет собственного графического интерфейса, поэтому на сцене появится пустой прозрачный элемент (рис. 12.13).

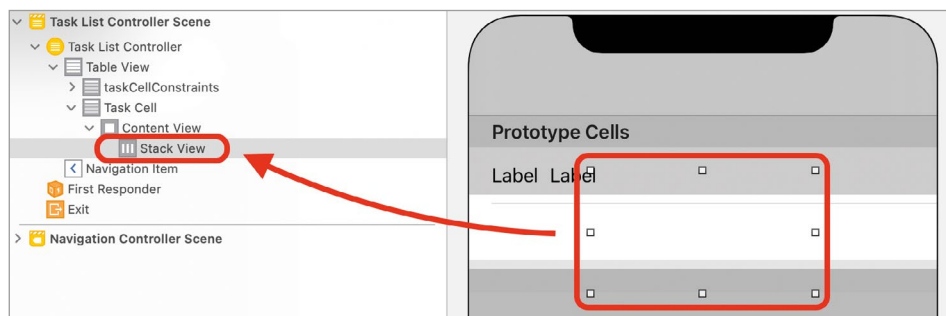


Рис. 12.13. Горизонтальный стек в ячейке

Изменим размеры стека так, чтобы он использовал все пространство ячейки. Для этого создадим необходимые констрейнты.

- ▶ Выделите **Horizontal Stack View**.
- ▶ Создайте ограничения в 0 точек на отступ со всех сторон от стека. При этом убедитесь, что пункт **Constrain to margin** активирован (рис. 12.14).

Теперь стек занимает все доступное, с учетом внутренних отступов ячейки, пространство (рис. 12.15).

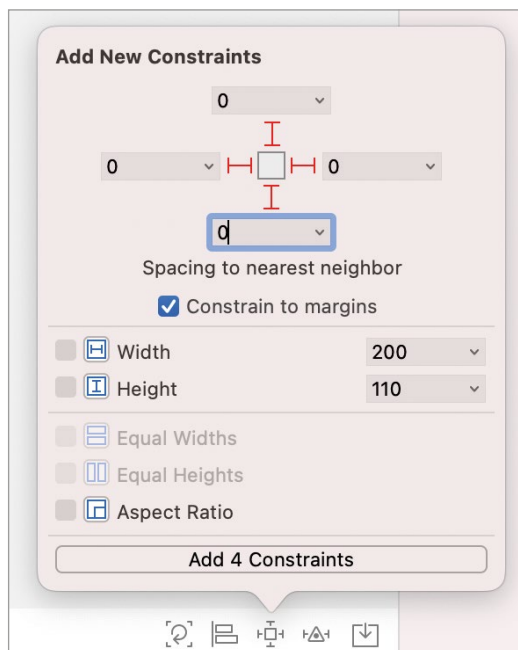


Рис. 12.14. Создание констрейнтов для стека

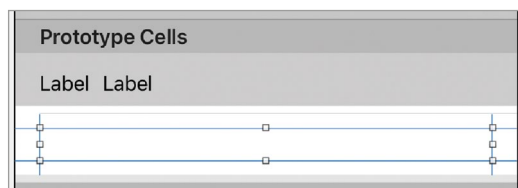


Рис. 12.15. Стек в ячейке

Элемент **Horizontal Stack View** предназначен для компоновки графических элементов в горизонтальную последовательность. В нашем случае требуется, чтобы в стеке находились две текстовые метки.

- Разместите в стеке на сцене две текстовые метки. Для этого достаточно поочередно перетянуть метки из библиотеки объектов прямо в стек на сцене или в **Document Outline**.

На рисунке 12.16 показан внешний вид стека после размещения меток.

Сейчас прототип имеет ту же самую проблему с растягиванием правой метки, которую мы решали ранее. Об этом говорит красный кружок в углу **Document Outline** (см. рис. 12.16). Для разрешения ситуации вновь изменим приоритет сопротивления расширению одной из меток.

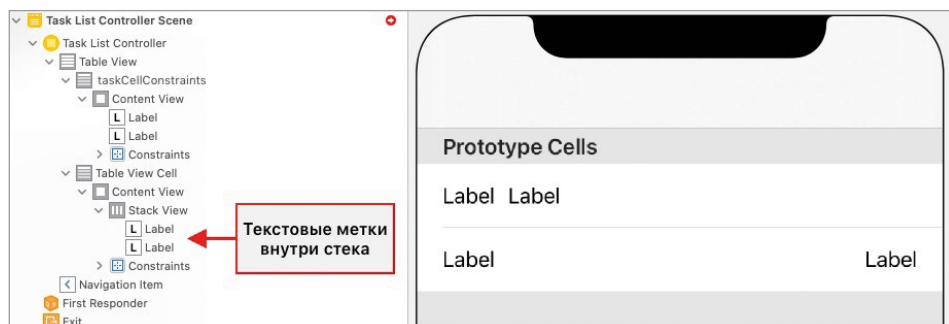


Рис. 12.16. Текстовые метки в составе горизонтального стека

- ▶ Выделите правую метку в составе стека.
- ▶ Откройте панель **Size Inspector**.
- ▶ Измените значение параметра **Horizontal** в разделе **Content Hugging Priority** с 251 на 250.

Теперь левая текстовая метка имеет больший приоритет при сопротивлении расширению, а значит правая метка свободно увеличивает свой размер (рис. 12.17).

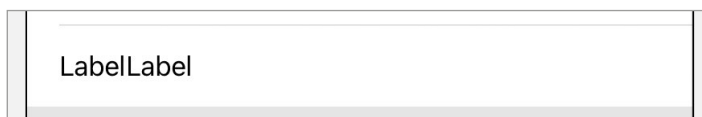


Рис. 12.17. Правая метка растянулась влево

Текстовые метки прижаты друг к другу, а нам требуется, чтобы между ними был отступ в 10 точек. Добиться этого можно настройкой горизонтального стека.

- ▶ Выделите горизонтальный стек на сцене.
- ▶ Откройте панель **Attributes Inspector**.

Изменяя значения свойств, размещенных в разделе **Stack View**, вы можете регулировать правила позиционирования элементов внутри стека (рис. 12.18):

- **Axis** определяет направление оси стека. С помощью этого свойства можно преобразовать **Horizontal Stack View** в **Vertical Stack View** и обратно.
- **Alignment** определяет способ выравнивания элементов внутри стека по оси, противоположной направлению стека. В нашем случае данный пункт указывает, как будут выровнены элементы по вертикали (прижаты к верху или низу, выровнены по центру или растянуты на всю высоту). Выбранное по умолчанию значение **Fill** указывает на то, что элементы должны быть растянуты.

- **Distribution** определяет способ размещения элементов по оси стека. Выбранное по умолчанию значение **Fill** определяет, что элементы должны быть растянуты на всю ширину стека, а, например, **Fill Equally** говорит о том, что элементы должны быть растянуты на всю ширину и иметь одинаковый размер.
 - **Spacing** определяет расстояние между элементами стека.
- ▶ Измените каждое из доступных значений и посмотрите, как будет изменяться внешний вид стека.
 - ▶ Верните значения всех параметров в исходное состояние (рис. 12.18).

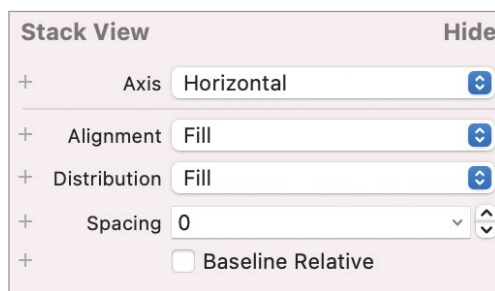


Рис. 12.18. Правила позиционирования элементов стека

По нашей задумке расстояние между текстовыми метками должно составлять 10 точек.

- ▶ Измените значение параметра **Spacing** на **10**.

С точки зрения графического интерфейса прототип ячейки полностью готов. Обратите внимание на то, что оба прототипа выглядят совершенно одинаково (рис. 12.19). Таким образом, мы добились одного и того же результата с помощью различных подходов. Осталось лишь указать идентификатор для переиспользования прототипа.

- ▶ Выделите второй прототип ячейки.
- ▶ Откройте панель **Attributes Inspector**.
- ▶ В поле **Identifier** укажите **taskCellStack**.

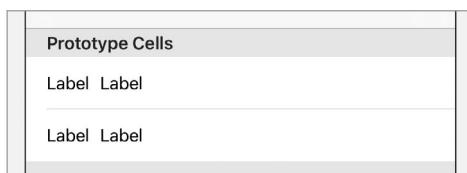


Рис. 12.19. Прототипы ячеек

Примечание Иногда перед разработчиком стоит задача обернуть уже размещенные на сцене элементы в стек. Чтобы максимально упростить эту задачу, исключить удаление и повторное размещение элементов, не заниматься их перетягиваниями и перемещениями, можно просто выделить обрабатываемые элементы и нажать кнопку **Embed In**, размещенную в нижней части **Interface Builder**, после чего во всплывающем окне выбрать подходящую обертку (рис. 12.20).

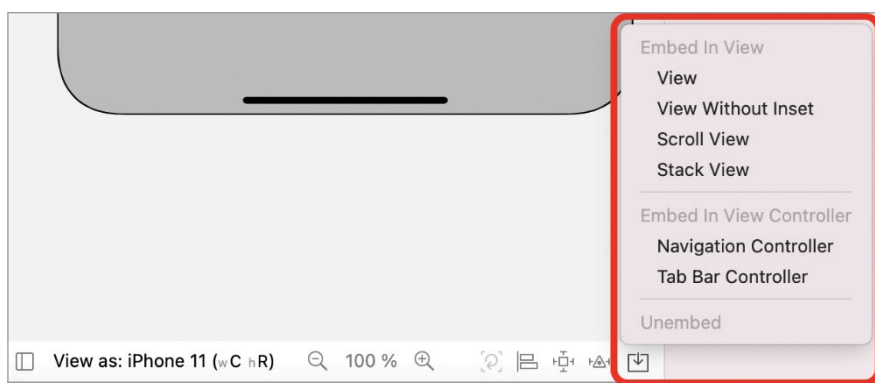


Рис. 12.20. Выбор обертки для элементов

Помимо двух видов стеков в данном разделе также доступны и другие элементы, включая уже известный вам навигационный контроллер.

Связь прототипа с кодом

Теперь займемся организацией связи между графическими элементами в составе прототипа и программным кодом. Ранее для этой цели мы использовали теги, но сейчас рассмотрим способ с созданием кастомного класса.

Говоря простым языком, мы можем создать кастомный класс, наследующий **UITableViewCell**, связать его с прототипом (подобно тому, как связывали view контроллер с классом) и создать аутлеты. Далее, обращаясь к аутлетам, мы сможем влиять на графические элементы, привязанные к ним, например, изменять текст в **Label**.

- ▶ В составе проекта создайте папку **View**.
- ▶ В папке **View** создайте папку **Cells**. В ней мы будем хранить все, что связано с ячейками.
- ▶ В папке **Cells** создайте новый файл, содержащий класс **TaskCell**. Он должен быть наследником **UITableViewCell**.

Примечание Напоминаю, что когда перед вами стоит задача создать класс, наследующий один из классов **UIKit**, то при выборе шаблона файла выбирайте **Cocoa Touch Class** (рис. 12.21).

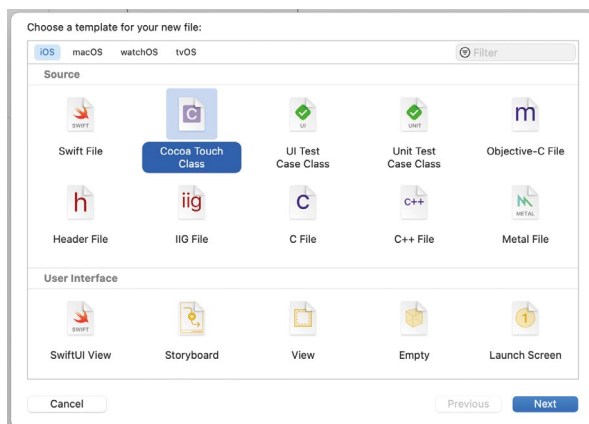


Рис. 12.21. Выбор шаблона при создании файла

В структуре проекта отобразился новый файл **TaskCell.swift**, в котором находится класс **TaskCell**. По умолчанию в данном классе уже реализованы методы **awakeFromNib** и **setSelected**:

```
class TaskCell: UITableViewCell {  
  
    override func awakeFromNib() {  
        super.awakeFromNib()  
        // Initialization code  
    }  
  
    override func setSelected(_ selected: Bool, animated: Bool) {  
        super.setSelected(selected, animated: animated)  
        // Configure the view for the selected state  
    }  
}
```

Ячейка, как и любой другой графический элемент, может быть создана исключительно с помощью программного кода, или с использованием визуального редактора **Interface Builder**. Метод **awakeFromNib** используется в тех случаях, когда ячейка версталась в **Interface Builder**, а значит ее структура хранится либо в storyboard-файле, либо в xib-файле (что это такое, мы рассмотрим несколько позже). Как только **UIKit** создает экземпляр класса, соответствующего ячейке, и загружает его структуру, сразу после этого происходит вызов **awakeFromNib**. Данный метод может быть использован, например, для того, чтобы произвести необходимые настройки графических элементов, размещенных в ячейке.

Метод **setSelected** вызывается после того, как ячейка была выбрана, то есть, после того, как пользователь нажмет на нее. Данный метод можно использовать для создания различных анимаций внутри ячейки.

Ни один из указанных методов не потребуется нам в ходе работы над прототипом. Свяжем класс **TaskCell** с прототипом.

- ▶ Выделите на сцене в составе табличного представления второй прототип. Для этого проще всего воспользоваться панелью **Document Outline**, выделив в составе **Table View** элемент **taskCellStack** (рис. 12.22).

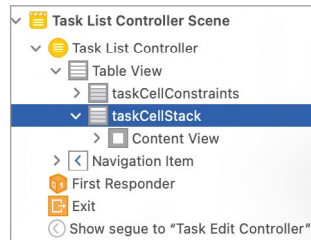


Рис. 12.22. Выбор прототипа

- ▶ Откройте панель **Identity Inspector**.
- ▶ В поле **Class** укажите **TaskCell**.

Созданная между классом и прототипом связь позволит с помощью программного кода изменять свойства элементов, размещенных в прототипе с помощью **Interface Builder**. Для этого необходимо реализовать аутлеты.

- ▶ В классе **TaskCell** создайте два аутлета типа **UILabel** (листинг 12.1).

ЛИСТИНГ 12.1

```
class TaskCell: UITableViewCell {

    @IBOutlet var symbol: UILabel!
    @IBOutlet var title: UILabel!
    // ...
}
```

- ▶ Свяжите аутлет-свойство **symbol** с левой меткой на сцене, а **title** – с правой. Для этого для каждого элемента выполните следующие действия (рис. 12.23):
 - выделите соответствующую метку на сцене;
 - откройте панель **Connections Inspector**;
 - перетяните серый кружок, расположенный напротив строки **New Referencing Outlet**, на элемент **taskCellStack** в составе сцены в **Document Outline**;
 - в выпадающем списке щелкните по имени аутлета, с которым связываете элемент.

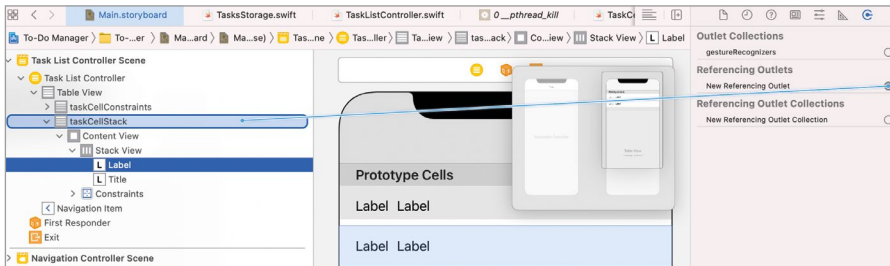


Рис. 12.23. Связь элемента в прототипе с аутлетом

Ранее мы связывали элементы, размещенные на сцене, с аутлетами, объявленными в классе вью контроллера, поэтому **New Referencing Outlet** перетягивали на значок контроллера (над сценой или в **Document Outline**). Сейчас же создается связь между элементом на сцене и аутлетом, реализованным внутри класса ячейки, а не класса контроллера. Именно по этой причине мы перетягивали серый кружок на элемент **Task Cell**, то есть на тот элемент, который связан с классом **TaskCell**.

12.5 Наполнение таблицы тестовыми данными

Прототипы ячеек созданы, а значит мы можем использовать их для наполнения табличного представления данными. Для этого будем использовать тестовый набор данных хранилища задач (класс **TasksStorage**).

В первую очередь необходимо подключить хранилище к контроллеру, в котором оно будет использоваться, а также создать несколько дополнительных свойств.

► В классе **TaskListController** объявите три свойства из листинга 12.2.

ЛИСТИНГ 12.2

```
// хранилище задач
var tasksStorage: TasksStorageProtocol = TasksStorage()
// коллекция задач
var tasks: [TaskPriority:[TaskProtocol]] = [:]

// порядок отображения секций по типам
// индекс в массиве соответствует индексу секции в таблице
var sectionsTypesPosition: [TaskPriority] = [.important, .normal]
```

Свойство **tasksStorage** будет использоваться для доступа к хранилищу задач. Актуальный список задач будет храниться в словаре **tasks**.

Список задач будет выводиться с помощью многосекционной таблицы – это обычные табличные представления, но содержащие несколько разделенных между собой секций (рис. 12.24). Каждая отдельная секция будет соответствовать задачам со своим приоритетом. Секции, как и строки таблицы, имеют упорядоченные целочисленные индексы (0, 1, 2 и т.д.). Верхняя секция имеет индекс 0. Для того, чтобы определить порядок отображения секций, будет использоваться свойство **sectionsTypesPosition**, где индекс элемента в массиве соответствует индексу секции. Если в ходе дальнейшей работы над проектом появится необходимость в новом типе задач, то для его отображения будет достаточно добавить новый элемент в перечислении **TaskPriority** и дополнить им свойство **sectionsTypesPosition**.

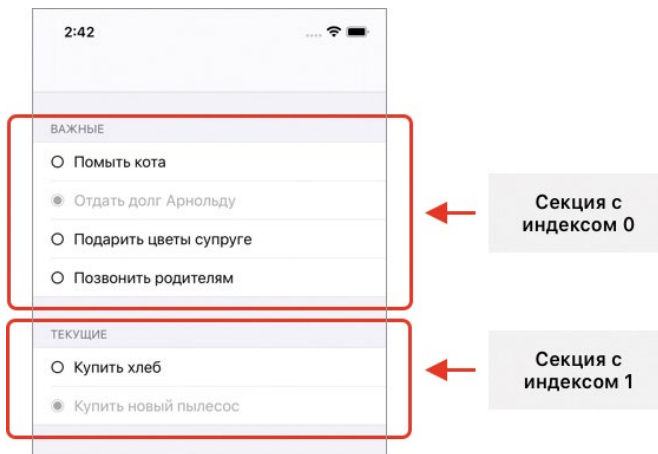


Рис. 12.24. Многосекционная таблица

В процессе загрузки сцены список задач из хранилища должен загружаться в контроллер для его последующего использования при наполнении табличного представления.

- Дополните метод **viewDidLoad** и реализуйте код загрузки и разбора задач (листинг 12.3).

ЛИСТИНГ 12.3

```
override func viewDidLoad() {
    super.viewDidLoad()
    // загрузка задач
    loadTasks()
}
```

```
private func loadTasks() {
```

```
// подготовка коллекции с задачами
// будем использовать только те задачи, для которых определена секция в
таблице
sectionsTypesPosition.forEach { taskType in
    tasks[taskType] = []
}
// загрузка и разбор задач из хранилища
tasksStorage.loadTasks().forEach { task in
    tasks[task.type]?.append(task)
}
}
```

Теперь в процессе подготовки сцены к отображению будет происходить загрузка из хранилища всех задач с их последующим разбором и размещением по элементам словаря **tasks**.

Отображение списка задач с использованием прототипа с ограничениями и тегами

При работе с многосекционными таблицами делегат табличного представления (им является вью контроллер) должен возвращать несколько значений:

- Количество секций в таблице.

В нашем случае оно будет равно количеству элементов в словаре **tasks**.

- Количество строк в определенной секции.

В нашем случае это количество значений в соответствующем элементе словаря **tasks**.

- Сконфигурированный экземпляр ячейки.

► Дополните класс **TaskListController** кодом из листинга 12.4. Обратите внимание, что некоторые из методов уже могут присутствовать в коде класса.

ЛИСТИНГ 12.4

```
// количество секций в таблице
override func numberOfSections(in tableView: UITableView) -> Int {
    return tasks.count
}

// количество строк в определенной секции
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
    // определяем приоритет задач, соответствующий текущей секции
    let taskType = sectionsTypesPosition[section]
```



```

    guard let currentTasksType = tasks[taskType] else {
        return 0
    }
    return currentTasksType.count
}

// ячейка для строки таблицы
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    return getConfiguredTaskCell_constraints(for: indexPath)
}

// ячейка на основе ограничений
private func getConfiguredTaskCell_constraints(for indexPath: IndexPath) ->
UITableViewCell {
    // загружаем прототип ячейки по идентификатору
    let cell = tableView.dequeueReusableCell(withIdentifier:
"taskCellConstraints", for: indexPath)
    // получаем данные о задаче, которую необходимо вывести в ячейке
    let taskType = sectionsTypesPosition[indexPath.section]
    guard let currentTask = tasks[taskType]?[indexPath.row] else {
        return cell
    }

    // текстовая метка символа
    let symbolLabel = cell.viewWithTag(1) as? UILabel
    // текстовая метка названия задачи
    let textLabel = cell.viewWithTag(2) as? UILabel

    // изменяем символ в ячейке
    symbolLabel?.text = getSymbolForTask(with: currentTask.status)
    // изменяем текст в ячейке
    textLabel?.text = currentTask.title

    // изменяем цвет текста и символа
    if currentTask.status == .planned {
        textLabel?.textColor = .black
        symbolLabel?.textColor = .black
    } else {
        textLabel?.textColor = .lightGray
        symbolLabel?.textColor = .lightGray
    }

    return cell
}

```

```
// возвращаем символ для соответствующего типа задачи
private func getSymbolForTask(with status: TaskStatus) -> String {
    var resultSymbol: String
    if status == .planned {
        resultSymbol = "\u{25CB}"
    } else if status == .completed {
        resultSymbol = "\u{25C9}"
    } else {
        resultSymbol = ""
    }
    return resultSymbol
}
```

Теперь в классе **TaskListController** реализованы три метода источника данных:

- метод **numberOfSections** возвращает количество секций в таблице;
- метод **numberOfRowsInSection** возвращает количество строк в определенной секции таблицы;
- метод **cellForRowAt** возвращает сконфигурированную ячейку для конкретной строки.

Также для обеспечения удобства работы код создания ячейки (на основе прототипа с ограничениями) вынесен в метод **getConfiguredTaskCell_constraints**. В дальнейшем мы воспользуемся и вторым прототипом, создав для него отдельный метод.

При создании ячеек и наполнении их данными, как и ранее, для того, чтобы определить, какой именно элемент необходимо вывести, используется значение типа **IndexPath**, содержащее информацию о «пути» к строке. Только если ранее было использовано свойство **row**, содержащее индекс строки, то теперь дополнительно происходит обращение и к свойству **section**, содержащему индекс секции.

Метод **getSymbolForTask** определяет символ, используемый для вывода в ячейке. Для запланированных (текущих) задач отображается круг, а для выполненных – закрашенный круг (рис. 12.25).

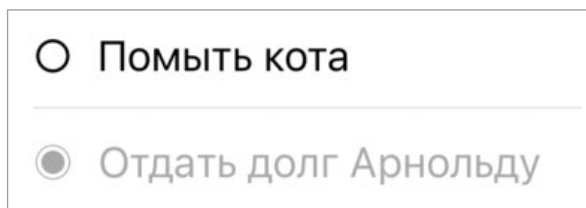


Рис. 12.25. Символы, используемые в таблице

► Запустите приложение.

На рисунке 12.26 показан текущий внешний вид табличного представления. Самое интересное то, что хотя наша таблица должна быть многосекционной, все задачи, занесенные в нее, отображаются единым списком.

«Где секции, Лебовски?»¹

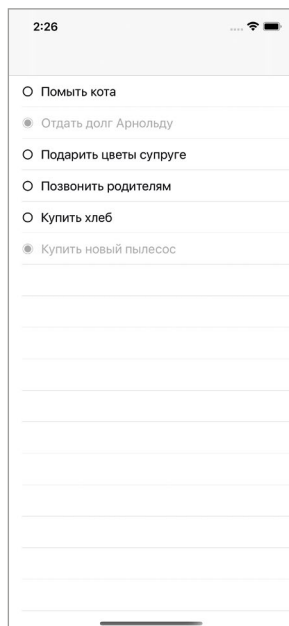


Рис. 12.26. Табличное представление с задачами

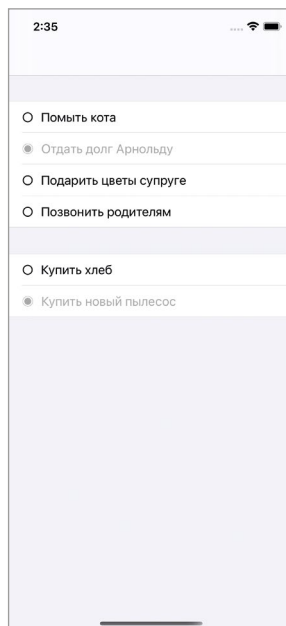


Рис. 12.27. Многосекционное табличное представление

По умолчанию табличное представление имеет плоский односекционный стиль отображения. Для того, чтобы таблица начала отображать более одной секции, требуется выполнить следующие действия:

- откройте файл **Main.storyboard**;
- выделите **Table View**;
- перейдите к панели **Attributes Inspector**;
- измените значение свойства **Style** на **Grouped**;
- запустите приложение.

Теперь таблица состоит из двух независимых секций (рис. 12.27).

¹ Фраза-мем, созданная на основе эпизода из фильма «Большой Лебовски».

Для каждой секции может быть определен текст, отображаемый в ее верхней части — так называемый заголовок секции.

► В классе **TaskListController** реализуйте метод из листинга 12.5.

ЛИСТИНГ 12.5

```
override func tableView(_ tableView: UITableView, titleForHeaderInSection
section: Int) -> String? {
    var title: String?
    let tasksType = sectionsTypesPosition[section]
    if tasksType == .important {
        title = "Важные"
    } else if tasksType == .normal {
        title = "Текущие"
    }
    return title
}
```

Метод **titleForHeaderInSection** позволяет на основании индекса секции указать текстовый заголовок, который будет выводиться над секцией.

► Запустите приложение.

Теперь каждая секция имеет заголовок, облегчающий навигацию в приложении (рис. 12.28). Ранее на экране присутствовал просто список задач, сгруппированных в секции по неизвестному для пользователя критерию. Теперь можно увидеть предназначение каждой секции.

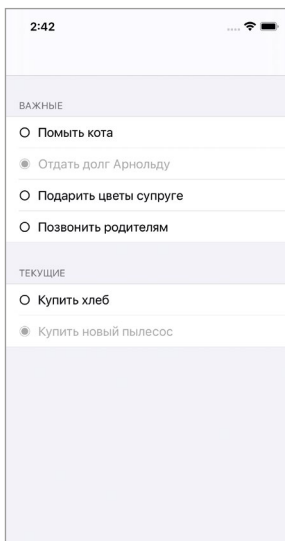


Рис. 12.28. Многосекционная таблица с заголовками секций

Сортировка списка задач

Сейчас запланированные и завершенные задачи выводятся общим списком, что не очень логично, так как большее внимание привлекают верхние элементы секций. Доработаем приложение таким образом, чтобы в обоих списках первыми шли запланированные задачи, а уже после них – выполненные.

В свойстве **tasks** контроллера находится неотсортированная коллекция задач, при этом Swift имеет встроенные средства сортировки – методы **sort** и **sorted**. Это прекрасный способ получить отсортированную коллекцию.

В этом случае возникает вопрос: как сопоставить элементы перечисления **TaskStatus**, на основе которых и должна производиться сортировка. Текущая реализация типа данных **TaskStatus** не позволяет сравнивать свои элементы. Для решения этой проблемы добавим к типу **TaskStatus** связанный тип и будем сравнивать значения перечисления именно по нему.

► Установите тип **Int** в качестве связанного для **TaskStatus** (листинг 12.6).

ЛИСТИНГ 12.6

```
enum TaskStatus: Int {  
    case planned  
    case completed  
}
```

Теперь первый элемент перечисления имеет связанное с ним целочисленное значение 0, второй – 1. Именно по этим значениям будет определяться порядок задач на сцене. Хотите изменить его? Просто поменяйте элементы в перечислении местами.

Воспользуемся новыми возможностями типа **TaskStatus** и осуществим сортировку задач при их загрузке из хранилища.

► Дополните код метода **loadTasks** кодом сортировки (листинг 12.7).

ЛИСТИНГ 12.7

```
private func loadTasks() {  
    // ...  
  
    // сортировка списка задач  
    for (tasksGroupPriority, tasksGroup) in tasks {  
        tasks[tasksGroupPriority] = tasksGroup.sorted { task1, task2 in  
            task1.status.rawValue < task2.status.rawValue  
        }  
    }  
}
```

► Запустите приложение.

Теперь списки важных и текущих задач отображаются с учетом сортировки элементов (рис. 12.29).

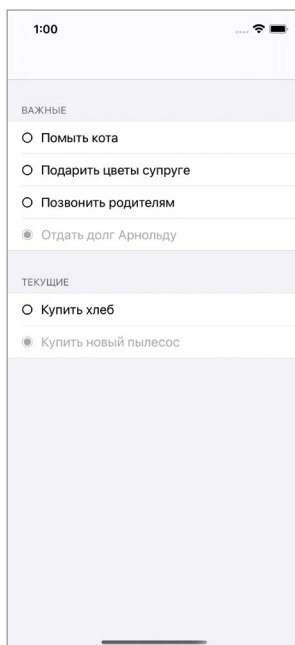


Рис. 12.29. Отсортированный список задач

Сортировка работает просто идеально, и при необходимости мы можем с легкостью изменить порядок элементов.

А теперь «вишенка на торте». Текущая реализация сортировки в корне неверна. Мы сделали ее лишь для того, чтобы указать на одну очень серьезную ошибку. Вспомните MVC – к какому компоненту относится визуальная составляющая приложения? К Представлению (View). Является ли порядок сортировки частью визуальной составляющей? Конечно! А к чему относится тип данных **TaskStatus**? К Модели (Model)! Он даже размещен в одноименной папке в составе проекта.

Использованный нами способ создает жесткую связь между Моделью и Представлением, а MVC создан для того, чтобы эту связь максимально ослабить.

Таким образом, логика, определяющая порядок элементов, должна быть вынесена из Модели.

► В классе **TaskListController** создайте свойство **tasksStatusPosition** (листинг 12.8).

ЛИСТИНГ 12.8

```
// порядок отображения задач по их статусу
var tasksStatusPosition: [TaskStatus] = [.planned, .completed]
```

- Измените код сортировки в методе **loadTasks** в соответствии с листингом 12.9.

ЛИСТИНГ 12.9

```
// сортировка списка задач
for (tasksGroupPriority, tasksGroup) in tasks {
    tasks[tasksGroupPriority] = tasksGroup.sorted { task1, task2 in
        let task1position = tasksStatusPosition.firstIndex(of: task1.status)
    ?? 0
        let task2position = tasksStatusPosition.firstIndex(of: task2.status)
    ?? 0
        return task1position < task2position
    }
}
```

Теперь данные в словаре **tasks** сортируются не на основе Модели, но порядок сортировки (свойство **tasksStatusPosition**) все еще не является частью Представления, а относится к Контроллеру. Однако такая реализация вполне допустима, так как в варианте MVC от Apple (а именно его мы рассматриваем в этой книге) Контроллер практически не отделим от Представления.

Примечание В варианте MVC от Apple есть один большой недостаток. Поскольку Контроллер слишком сильно связан с Представлением, возможно возникновение ситуации, называемой среди разработчиков Massive View Controller, при которой Контроллер разрастается до чрезвычайно больших размеров за счет включения в себя в том числе большей части Представления.

Оба рассмотренных варианта сортировки с точки зрения пользователя работают одинаково хорошо, но для разработчика должен быть очень важен момент выполнения принципа разделения полномочий. Каждый компонент программы должен выполнять задачи, которые соответствуют ему. Уменьшение связанности элементов проекта приведет к тому, что внесение правок в Модель не отобразится на внешнем виде.

Отображение списка задач с использованием прототипа со стеком и кастомным классом

Теперь посмотрим, как будет выглядеть таблица при использовании прототипа, основанного на применении Horizontal Stack View и кастомного класса **TaskCell**.

- Добавьте в класс **TaskListController** метод из листинга 12.10.

ЛИСТИНГ 12.10

```
// ячейка на основе стека
private func getConfiguredTaskCell_stack(for indexPath: IndexPath) ->
UITableViewCell {
    // загружаем прототип ячейки по идентификатору
    let cell = tableView.dequeueReusableCell(withIdentifier:
"taskCellStack", for: indexPath) as! TaskCell
    // получаем данные о задаче, которые необходимо вывести в ячейке
    let taskType = sectionsTypesPosition[indexPath.section]
    guard let currentTask = tasks[taskType]?[indexPath.row] else {
        return cell
    }

    // изменяем текст в ячейке
    cell.title.text = currentTask.title
    // изменяем символ в ячейке
    cell.symbol.text = getSymbolForTask(with: currentTask.status)

    // изменяем цвет текста
    if currentTask.status == .planned {
        cell.title.textColor = .black
        cell.symbol.textColor = .black
    } else {
        cell.title.textColor = .lightGray
        cell.symbol.textColor = .lightGray
    }

    return cell
}
```

Тело метода **getConfiguredTaskCell_stack** очень похоже на тело метода создания ячейки на основе ограничений **getConfiguredTaskCell_constraints**. Разница заключается лишь в способе доступа к текстовым меткам, размещенным в прототипе. Второй прототип связан с кастомным классом. Именно по этой причине используется приведение (**as! TaskCell**) при получении экземпляра ячейки. Далее обращение к меткам производится посредством аутлетов.

- Измените тело метода **cellForRowAt** в соответствии с листингом 12.11, добавив в него метод, возвращающий ячейку на основе стека. При этом закомментируйте вызов метода **getConfiguredTaskCell_constraints**.

ЛИСТИНГ 12.11

```
// ячейка на основе констрейнтов
// return getConfiguredTaskCell_constraints(for: indexPath)
// ячейка на основе стека
return getConfiguredTaskCell_stack(for: indexPath)
```

► Запустите приложение.

Попробуйте найти 10 отличий между внешними видами экранов, основанных на использовании созданных прототипов (рис. 12.30). Они выглядят полностью идентично.

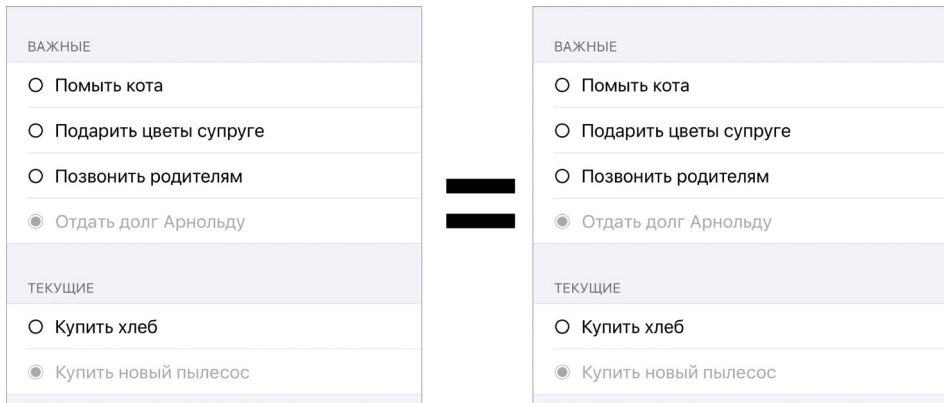


Рис. 12.30. Внешний вид списков задач

Какой вариант реализации использовать?

Нет однозначного ответа на вопрос, что использовать: стек или ограничения, теги или кастомный класс.

В общем случае, если ячейка (или любое другое представление) содержит мало элементов, использование тегов может ускорить разработку. Но такой подход не всегда очевиден для программиста, а вот кастомный класс все раскладывает по полочкам, хоть и требует больше времени для реализации.

Что касается стека, он, в отличие от констрейнтов, не дает вам полного контроля над расположением элементов. В то же время его хватает для выполнения 90% задач.

Моя основная цель заключается в том, чтобы дать вам первоначальные знания о работе с каждым из возможных вариантов, а полноценное понимание всех концепций придет к вам с опытом.

Доработка внешнего вида

Рассмотрим одну интересную ситуацию, с которой вы однозначно столкнетесь при использовании программы — это использование многострочных текстовых меток. Название задачи не обязательно должно состоять из двух или трех слов, написанных в одну строку — оно может включать в себя неограниченное количество текста. Как в этом случае поведет себя интерфейс приложения?

- В классе **TasksStorage** добавьте в тестовые данные еще одну задачу. Она должна содержать большое количество текста (листинг 12.12).

ЛИСТИНГ 12.12

```
let testTasks: [TaskProtocol] = [  
    // ...  
    Task(title: "Пригласить на вечеринку Дольфа, Джеки, Леонардо, Уилла и  
    Брюса", type: .important, status: .planned)  
]
```

- Запустите приложение.

Строка с задачей, содержащей длинный текст, отображается без иконки в самом начале, а текст и вовсе обрезан (рис. 12.31). Но в этом нет ничего удивительного. Дело в том, что правая метка пытается максимально расшириться, чтобы вместить в себя больше текста. А так как ее приоритет сопротивления расширению (**Content Hugging Priority**) ниже (250 против 251 у левой метки), ширина левой метки становится равной 0.

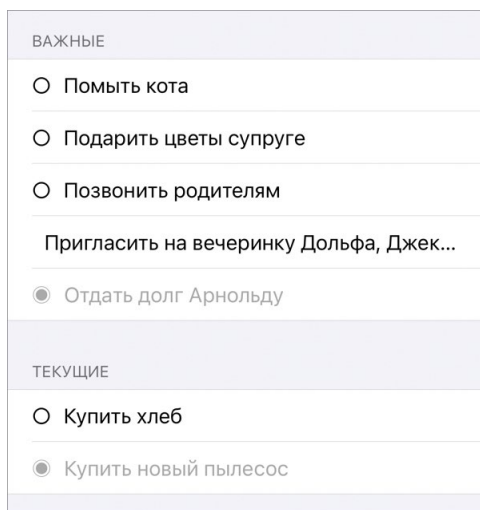


Рис. 12.31. Некорректное оформление ячейки при отображении длинного текста задачи

Текстовая метка позволяет выводить в себе многострочный текст, а текущие метки на сцене настроены для отображения однострочного текста.

- ▶ Во втором прототипе выделите правую текстовую метку.
- ▶ Откройте панель **Attributes Inspector**.
- ▶ Измените значение поля **Lines** на **0**.

Свойство **Lines** текстовой метки определяет, сколько строк текста может быть выведено в ней. При значении 0 метка может содержать произвольное количество строк.

- ▶ Запустите приложение.

«Да, что же это за напасть!», сцена опять выглядит не так, как требуется (рис. 12.32). Несмотря на то, что текст задачи виден полностью, справа от иконки слишком много свободного места, а сама ячейка разделена ровно пополам на иконку и текст задачи. Причем данное изменение отображается и в самом прототипе в **Interface Builder**.

ВАЖНЫЕ	
<input type="radio"/>	Помыть кота
<input type="radio"/>	Подарить цветы супруге
<input type="radio"/>	Позвонить родителям
<input type="radio"/>	Пригласить на вечеринку Дольфа, Джеки, Леонардо, Уилла и Брюса
<input checked="" type="radio"/>	Отдать долг Арнольду
ТЕКУЩИЕ	
<input type="radio"/>	Купить хлеб
<input checked="" type="radio"/>	Купить новый пылесос

Рис. 12.32. Некорректное оформление ячейки при отображении длинного текста задачи

Поговорим о причинах такого результата. Несмотря на то, что текстовые метки имеют различный приоритет сопротивления расширению, правая метка может включать любое количество строк текста. Получается, для нее нет необходимости растягиваться влево, так как она может расширяться в высоту. Поэтому различия в приоритете не приводят ни к каким изменениям и стек делится ровно пополам.

Одним из вариантов решения этой проблемы является жесткое указание ширины метки, содержащей иконку.

- Для левой метки во втором прототипе создайте ограничение на ширину в 20 точек (рис. 12.33).

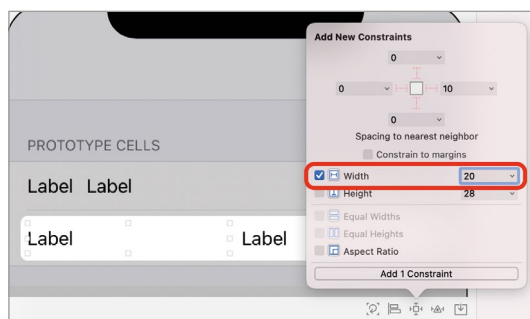


Рис. 12.33. Создание ограничения на ширину элемента

- Запустите приложение.

Теперь ячейки имеют корректный вид независимо от того, сколько текста включает в себя задача (рис. 12.34).

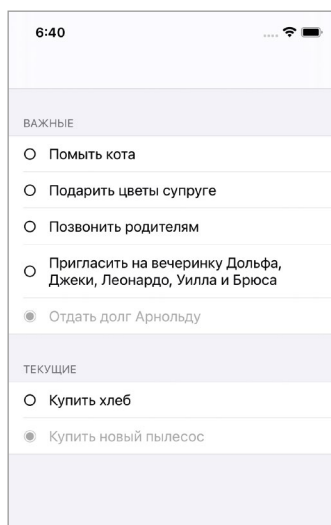


Рис. 12.34. Корректный внешний вид списка задач

На этом первая часть работ по оформлению экрана со списком задач завершена. В следующей главе мы займемся реализацией дополнительных функций, благодаря которым появится возможность перемещать задачи между секциями, отмечать их как выполненные и при необходимости удалять.

Глава 13.

Изменение элементов табличного представления

В этой главе вы:

- научитесь обрабатывать нажатия по строкам Table View;
- научитесь обновлять отдельные части табличных представлений;
- узнаете, что такое режим редактирования табличного представления.

Текущая версия приложения **«To-Do Manager»** пока умеет только выводить список задач, созданных пользователем (хотя на данный момент это лишь набор тестовых данных). Но каждая задача имеет свой жизненный цикл: после создания она может быть отмечена как выполненная или вовсе удалена. В этой главе мы реализуем функцию изменения статуса каждой отдельной задачи в списке, а также добавим режим редактирования табличного представления для удаления и сортировки его элементов.

13.1 Изменение статуса задач

Одной из функций, без которой не может существовать менеджер задач, является возможность изменения статуса задач с «запланирована» на «выполнена», а при необходимости и обратно.

Функцию выполнения задачи (то есть, изменения статуса с «запланирована» на «выполнена») реализуем через нажатие на строку, соответствующую этой задаче. Проще говоря, если пользователь нажимает на строку с запланированной задачей, она автоматически помечается как выполненная и перемещается в конец секции. Для обратного изменения статуса будем использовать свайп вправо по задаче, имеющей статус «выполнено».

Изменение статуса задачи на «выполнено»

Обработкой нажатий по строкам табличного представления занимается его делегат. При нажатии на любую строку происходит вызов метода **didSelectRowAt**

делегата, в который передается информация о строке, на которую было произведено нажатие.

СИНТАКСИС

Метод `UITableViewDelegate.tableView(_:didSelectRowAt:)`

Срабатывает при нажатии на строку таблицы.

Аргументы

- `_`: `UITableView` – экземпляр табличного представления, в котором было совершено нажатие.
- `didSelectRowAt: IndexPath` – объект, описывающий путь к строке, на которую было осуществлено нажатие.
 - `IndexPath.section` – индекс секции табличного представления, в которой было совершено нажатие.
 - `IndexPath.row` – индекс строки табличного представления, на которую было выполнено нажатие.

Делегатом табличного представления, отображающим список задач, является класс **`TaskListController`**, поэтому метод **`didSelectRowAt`** необходимо реализовать в нем.

- В классе **`TaskListController`** реализуйте метод **`didSelectRowAt`** (листинг 13.1).

ЛИСТИНГ 13.1

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    // 1. Проверяем существование задачи
    let taskType = sectionsTypesPosition[indexPath.section]
    guard let _ = tasks[taskType]?[indexPath.row] else {
        return
    }
    // 2. Убеждаемся, что задача не является выполненной
    guard tasks[taskType]![indexPath.row].status == .planned else {
        // снимаем выделение со строки
        tableView.deselectRow(at: indexPath, animated: true)
        return
    }
    // 3. Отмечаем задачу как выполненную
    tasks[taskType]![indexPath.row].status = .completed
    // 4. Перезагружаем секцию таблицы
    tableView.reloadSections(IndexSet(arrayLiteral: indexPath.section),
with: .automatic)
}
```

Разберем, какие операции выполняются в методе.

1. Для обеспечения безопасности работы (чтобы избежать внезапных падений приложения) происходит проверка того, что в свойстве **tasks** существует задача, соответствующая нажатой строке.

2. Далее происходит проверка условия, что выбранная задача не является выполненной. Нажатие на выполненную задачу не должно изменять статус на «запланированная» — для этого будет использоваться свайп.

Обратите внимание, что внутри **guard** с помощью метода **deselectRow** происходит снятие выделения нажатой строки. Если эту строчку опустить, нажатие по выполненной задаче приведет к ее выделению (перекрасит строку в серый цвет) и не снимет его.

3. Выбранная задача отмечается как выполненная.

4. Само по себе изменение значения свойства **tasks** не приведет к обновлению табличного представления. Для того, чтобы на экране отобразились актуальные данные, требуется принудительно вызвать обновление списка задач. Ранее с этой целью мы обращались к методу **reloadData**, но в данном случае используется **reloadSections**, который позволяет обновить только требуемые секции, а не всю таблицу целиком.

СИНТАКСИС

Метод `UITableView.reloadSections(_:with:)`

Обновляет указанные секции табличного представления.

Аргументы

- `_`: `IndexSet` – содержит индексы секций, которые необходимо обновить.
- `with`: `UITableView.RowAnimation` – тип анимации, с которой будет произведено обновление

► Запустите приложение и протестируйте реализованную функциональность.

Нажатие на строку успешно отмечает задачу как выполненную, но задачи остаются на своих местах и не опускаются в нижнюю часть секции.

Для устранения этой проблемы произведем доработку функции сортировки. В данный момент сортировка производится в теле метода **loadTasks**, но изменение статуса задачи никак на него не завязано. Если мы реализуем сортировку в **didSelectRowAt**, это приведет к дублированию кода (он будет реализован и в **loadTasks**, и в **didSelectRowAt**), что крайне нежелательно.

Одним из вариантов является вынос кода сортировки в отдельный метод, например, **sortTasks**, и его последующий вызов при загрузке данных из хранилища и изменении статуса задачи. Но мы поступим иначе. Подумайте, возможен

ли такой вариант, что один из элементов свойства **tasks** изменил свое значение (задача отмечена, как выполненная, удалена или добавлена в свойство), а коллекцию задач требуется оставить неотсортированной? Думаю, что нет. По этой причине мы можем использовать наблюдатель **didSet** для свойства **tasks**, чтобы обеспечить принудительную сортировку при каждом обновлении списка задач.

- Из метода **loadTasks** удалите код, производящий сортировку списка задач.
- Дополните свойство **tasks** наблюдателем, обеспечивающим сортировку элементов (листинг 13.2).

ЛИСТИНГ 13.2

```
var tasks: [TaskPriority:[TaskProtocol]] = [:] {
    didSet {
        for (tasksGroupPriority, tasksGroup) in tasks {
            tasks[tasksGroupPriority] = tasksGroup.sorted{ task1, task2 in
                let task1position = tasksStatusPosition.firstIndex(of:
task1.status) ?? 0
                let task2position = tasksStatusPosition.firstIndex(of:
task2.status) ?? 0
                return task1position < task2position
            }
        }
    }
}
```

- Запустите приложение и протестируйте реализованную функциональность.

Теперь при нажатии на любую из строк, соответствующих запланированной задаче, происходит ее перевод в статус «выполнена» с одновременным переносом в конец соответствующей секции. При этом сортировка данных сохранилась и непосредственно перед появлением сцены на экране. Любое изменение значений элементов свойства **tasks** приводит к немедленной сортировке списка задач.

Изменение статуса задачи на «запланирована»

Теперь реализуем функцию изменения статуса задачи с «выполнена» на «запланирована». Она может понадобиться в том случае, если задача была отмечена как выполненная случайно. Как уже говорилось ранее, для этого будем использовать свайп вправо.

- Дополните класс **TaskListController** кодом из листинга 13.3.

ЛИСТИНГ 13.3

```

override func tableView(_ tableView: UITableView, leadingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {
    // получаем данные о задаче, которую необходимо перевести в статус "запланирована"
    let taskType = sectionsTypesPosition[indexPath.section]
    guard let _ = tasks[taskType]?[indexPath.row] else {
        return nil
    }
    // проверяем, что задача имеет статус "выполнено"
    guard tasks[taskType]![indexPath.row].status == .completed else {
        return nil
    }

    // создаем действие для изменения статуса
    let actionSwipeInstance = UIContextualAction(style: .normal, title: "Не выполнена") { _, _, _ in
        self.tasks[taskType]![indexPath.row].status = .planned
        self.tableView.reloadSections(IndexSet(arrayLiteral: indexPath.section), with: .automatic)
    }
    // возвращаем настроенный объект
    return UISwipeActionsConfiguration(actions: [actionSwipeInstance])
}

```

Метод **leadingSwipeActionsConfigurationForRowAt** уже знаком вам по изученному материалу книги. Он возвращает значение типа **UISwipeActionsConfiguration**, содержащее данные о доступных действиях, отображаемых при свайпе по строке таблицы.

► Запустите приложение и протестируйте новую функциональность.

Теперь вы можете изменять статус задач: нажатие отмечает их как выполненные, а с помощью свайпа вправо их можно вернуть в список запланированных. При этом каждое изменение приводит к незамедлительной сортировке списка.

13.2 Режим редактирования

Некоторые из элементов, входящих в состав фреймворка **UIKit**, поддерживают режим редактирования. К таким элементам, к примеру, относятся текстовое поле (**UITextField**), табличное представление (**UITableView**) и вью контроллер (**UIViewController**).

Режим редактирования текстового поля

Если текстовое поле находится в режиме редактирования, это значит, что пользователь может изменять текст, находящийся в нем (в поле мигает индикатор ввода, а на экране отображается клавиатура) (рис. 13.1). Узнать, в каком состоянии находится Text Field можно обратившись к его свойству **isEditing**. Оно имеет логический тип данных и если возвращает true, то в данный момент поле редактируется. Свойство **isEditing** класса **UITextField**, представляющего собой текстовое поле, является «свойством только для чтения», то есть вы не можете изменить его. Но если у текстового поля определен делегат, поле уведомляет его при переходе к режиму редактирования и выходе из него.

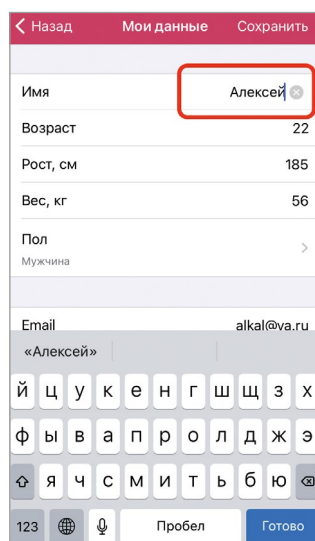


Рис. 13.1. Текстовое поле в режиме редактирования

Примечание Информация о текстовом поле приведена лишь для общего понимания картины. В этом разделе мы сфокусируемся на работе с табличным представлением.

Режим редактирования табличного представления

Табличное представление также может находиться в режиме редактирования (рис. 13.2). В этом случае в левой части каждой строки может отображаться иконка удаления (круг красного цвета с минусом внутри) или добавления (круг зеленого цвета с плюсом внутри). С помощью программного кода вы имеете возможность настроить наличие или отсутствие режима редактирования для каждой строки таблицы, а также реакцию нажатия на иконку.

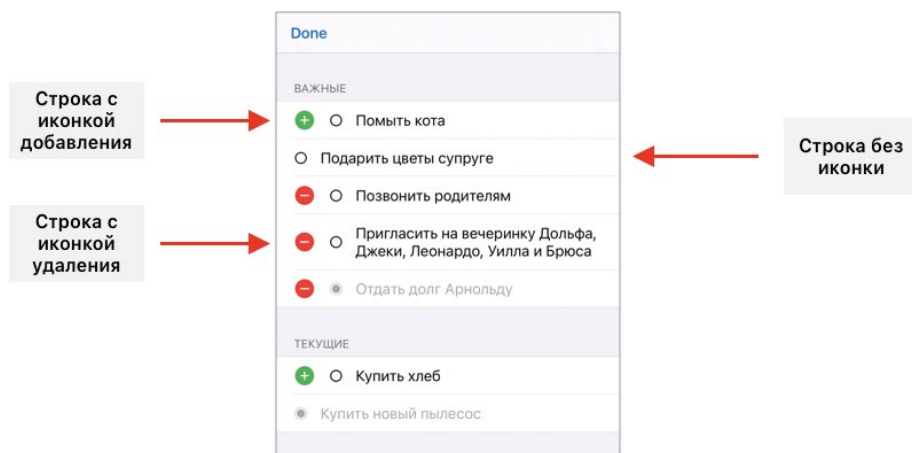


Рис. 13.2. Режим редактирования табличного представления

Как и в случае с текстовым полем, для работы с режимом редактирования используется свойство **isEditing**, определяющее, находится ли элемент в режиме редактирования. Но на этот раз свойство доступно не только для чтения, но и для записи. Иначе говоря, у вас есть возможность активировать и деактивировать режим редактирования Table View.

Всякий раз, когда значение свойства **isEditing** изменяется, происходит автоматический вызов метода **setEditing** данного табличного представления. При необходимости данный метод может быть переопределен (если конечно таблица основана на кастомном классе), чтобы реализовать, например, собственную анимацию изменения режима или выполнить другие операции.

Источник данных (Data Source) табличного представления позволяет произвести настройки режима редактирования табличного представления.

Возможность редактирования отдельной строки (отображение соответствующей иконки при переходе в режим редактирования) определяется с помощью метода **canEditRowAt**. Данный метод необходимо переопределить в источнике данных. По умолчанию, если метод не реализован, все строки считаются редактируемыми.

СИНАКСИС

Метод `UITableViewDataSource.tableView(_:canEditRowAt:) -> Bool`

Определяет возможность редактирования конкретной строки в табличном представлении.

Аргументы

- `_`: `UITableView` – табличное представление, для строк которого определяется возможность редактирования строк.
- `canEditRowAt`: `IndexPath` – путь к строке таблицы, для которой определяется возможность редактирования.

- `indexPath.section` – индекс секции.
- `indexPath.row` – индекс строки.

Возвращаемое значение

- `Bool` – определяет возможность редактирования строки при переходе в режим редактирования.

Пример

Запрет редактирования первой строки во всех секциях таблицы:

```
override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool {
    // если индекс строки равен 0
    if indexPath.row == 0 {
        return false
    }
    return true
}
```

Стиль режима редактирования каждой строки в таблице (какую иконку выводить и выводить ли вообще) определяется с помощью метода **`editingStyleForRowAt`**, который также необходимо реализовать в источнике данных. Если данный метод не определен, то по умолчанию для всех редактируемых строк используется стиль **`.delete`** (красный круг).

СИНТАКСИС

Метод `UITableViewDataSource.tableView(_:editingStyleForRowAt:)` -> `UITableViewCellEditingStyle`

Определяет стиль иконки, отображаемой в строке в режиме редактирования.

Аргументы

- `_:` `UITableView` – табличное представление, для которого настраивается режим редактирования.
- `editingStyleForRowAt: IndexPath` – путь к строке таблицы, для которой определяется стиль.
 - `indexPath.section` – индекс секции.
 - `indexPath.row` – индекс строки.

Возвращаемое значение

- `UITableViewCellEditingStyle` – перечисление, определяющее стиль режима редактирования.
 - `.delete` – отображается круг красного цвета с минусом внутри.
 - `.insert` – отображается круг зеленого цвета с плюсом внутри.
 - `.none` – символ не отображается.

Пример

Первая строка каждой секции таблицы имеет стиль `.insert`, а остальные `.delete`:

```
override func tableView(_ tableView: UITableView, editingStyleForRowAt
indexPath: IndexPath) -> UITableViewCell.EditingStyle {
    if indexPath.row == 0 {
        return .insert
    }
    return .delete
}
```

Обработка нажатия на иконку редактирования осуществляется с помощью метода `commit`. Его также необходимо реализовывать в источнике данных табличного представления.

СИНТАКСИС

Метод `UITableViewDataSource.tableView(_:commit:forRowAt:)`

Обрабатывает нажатие на иконку редактирования в строке.

Аргументы

- `_: UITableView` – табличное представление, для которого обрабатывается нажатие.
- `commit: UITableViewCell.EditingStyle` – используемый в строке стиль иконки, по которой произведено нажатие.
 - `.delete` – круг красного цвета с минусом внутри.
 - `.insert` – круг зеленого цвета с плюсом внутри.
- `forRowAt: IndexPath` – путь к строке таблицы, для которой определяется стиль.

Пример

Вставка или удаление строки в табличном представлении в зависимости от стиля.

```
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        tableView.deleteRows(at: [indexPath], with: .automatic)
    } else if editingStyle == .insert {
        tableView.insertRows(at: [indexPath], with: .automatic)
    }
}
```

Примечание Методы `deleteRows` и `insertRows` позволяют соответственно удалить или вставить строки в табличном представлении.

Режим редактирования вью контроллера

На первый взгляд может показаться очень странным само наличие режима редактирования у вью контроллера. Зачем он вообще нужен и какое визуальное представление имеет?

Режим редактирования view контроллера выглядит ... никак. Как и в случае с Table View, при изменении значения свойства **isEditing** экземпляра класса контроллера происходит вызов метода **setEditing**, что приводит к ... ничему. Не происходит ровным счетом ничего, но наличие этого режима предоставляет нам широкие возможности! При необходимости метод **setEditing** может быть переопределен в классе контроллера, а это позволяет реализовать в нем любую требуемую логику.

Рассмотрим пример.

Предположим, что приложение содержит сцену с размещенными на ней табличным представлением и кнопкой. Кнопка предназначена для активации режима редактирования на сцене. Для реализации этой функциональности при нажатии кнопки будет вызываться связанный с ней метод контроллера, принудительно изменяющий режим редактирования view контроллера:

```
@IBAction func setEditMode(_ sender: UIButton) {
    // если контроллер не в режиме редактирования
    if isEditing == false {
        // изменяем режим редактирования
        isEditing = true
        // изменяем текст кнопки перевода в режим редактирования
        button.titleLabel?.text = «Завершить редактирование»
    }
    // если контроллер уже в режиме редактирования
    if isEditing == true {
        // изменяем режим редактирования
        isEditing = false
        // изменяем текст кнопки перевода в режим редактирования
        button.titleLabel?.text = «Редактировать»
    }
}
```

При нажатии кнопки значение свойства **isEditing** будет изменяться, вследствие чего будет происходить вызов метода **setEditing**. Если в классе контроллера данный метод будет переопределен, в его теле можно будет активировать или деактивировать режим редактирования табличного представления, размещенного на сцене:

```
override func setEditing(_ editing: Bool, animated: Bool) {
    // применяем к TableView текущий режим контроллера
    tableView.isEditing = editing
}
```

Таким образом, нажав кнопку, которая не связана с таблицей, мы имеем возможность перевести таблицу в режим редактирования и изменить любые другие элементы на сцене (например, текст в этой кнопке, активирующей режим

редактирования). Наличие свойства **isEditing** и метода **setEditing** в классе `UITableViewController` позволяет полностью контролировать процесс перевода сцены и ее отдельных элементов в режим редактирования.

Режим редактирования контроллера табличного представления

Особенно полезным режим редактирования является в случае использования контроллера табличного представления. Как вы знаете, `Table View Controller` – это `View Controller`, работающий в жесткой сцепке с `Table View`. При переходе в режим редактирования `Table View Controller` также автоматически переводит в этот режим и `Table View`. И этой особенностью мы воспользуемся для реализации возможности удаления задач.

13.3 Удаление задач с помощью режима редактирования

В состав класса **`UIViewController`** входит особое свойство **`editButtonItem`** типа **`UIBarButtonItem`**, с помощью которого на сцену (посредством программного кода) можно добавить кнопку активации/деактивации режима редактирования. При нажатии на нее `UIViewController` автоматически переходит в режим редактирования, при этом текст кнопки соответствующим образом изменяется (в кнопке будет отображено **`Edit`** или **`Done`**, в зависимости от текущего значения свойства **`isEditing`** контроллера).

Класс **`UITableViewController`**, на котором основана сцена нашего приложения, является дочерним по отношению к **`UIViewController`**, а значит также имеет в своем составе свойство **`editButtonItem`**. Соответственно, при использовании кнопки из данного свойства в режим редактирования будет переводиться не только контроллер, но и входящее в его состав табличное представление.

Мы воспользуемся этой возможностью и добавим в панель навигации кнопку активации режима редактирования.

► Доработайте метод **`viewDidLoad`** в соответствии с листингом 13.4.

ЛИСТИНГ 13.4

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    // загрузка задач  
    loadTasks()  
    // кнопка активации режима редактирования  
    navigationItem.leftBarButtonItem = editButtonItem  
}
```

Свойство **navigationItem** возвращает ссылку на навигационную панель, относящуюся к навигационному контроллеру, в который обернута текущая сцена. Подобный подход, когда с помощью свойства осуществляется доступ к родительскому (с точки зрения иерархии) элементу, вы уже видели, когда рассматривали свойство **navigationController**. С помощью свойства **navigationItem** у вас есть возможность вносить изменения в состав элементов в навигационной панели. Так **leftBarButtonItem**, использованный в листинге, позволяет добавить кнопку (значение типа **UIBarButtonItem**) в левую часть панели. Аналогичным образом действует **rightBarButtonItem**, определяя кнопку для правой части панели.

К чему приведет данный код? В навигационную панель, в ее левую часть, будет добавлена кнопка активации/деактивации режима редактирования. Причем экземпляр, описывающий данную кнопку, хранится в свойстве **editButtonItem** вью контроллера.

► Запустите приложение.

В навигационной панели появилась кнопка **Edit**, при нажатии которой контроллер, а соответственно и таблица, переходят в режим редактирования. На рисунке 13.3 показан внешний вид сцены после нажатия кнопки. Как вы могли заметить, текст кнопки изменился на **Done**, а в левой части каждой строки отображается специальная иконка, символизирующая удаление.

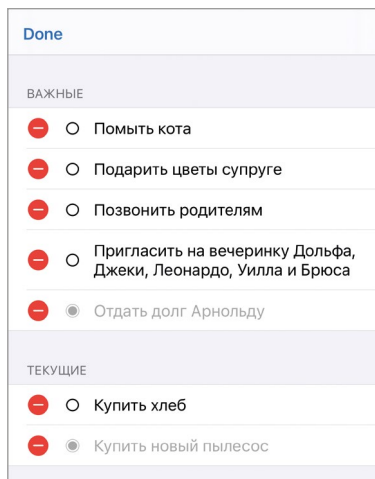


Рис. 13.3. Сцена с активированным режимом редактирования

Несмотря на то, что иконки для удаления строк отображаются, нажатие на них не приводит к какому-либо результату.

► В классе **TaskListController** реализуйте метод, осуществляющий удаление задачи (листинг 13.5).

ЛИСТИНГ 13.5

```
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellStyle, forRowAt indexPath: IndexPath) {
    // удаляем строку, соответствующую задаче
    tableView.deleteRows(at: [indexPath], with: .automatic)
}
```

- ▶ Запустите приложение и попробуйте удалить одну из задач в верхней секции.

При попытке удаления задачи приложение экстренно завершит свою работу, а в консоли отобразится сообщение об ошибке примерно следующего содержания:

```
Invalid update: invalid number of rows in section 0. The number of rows
contained in an existing section after the update (5) must be equal to the
number of rows contained in that section before the update (5), plus or
minus the number of rows inserted or deleted from that section (0 inserted,
1 deleted) and plus or minus the number of rows moved into or out of that
section (0 moved in, 0 moved out)
```

Неверное количество строк в секции с индексом 0. Количество строк в секции после обновления (5) должно быть эквивалентно количеству строк до обновления (5), плюс или минус количество добавленных или удаленных строк (0 добавлено, 1 удалена) и плюс или минус количество строк, перенесенных в другую или из другой секции (0 перенесено в, 0 перенесено из).

Разберем, что это значит, и почему возникла ошибка.

После того, как был вызван метод **deleteRows**, производящий удаление строки, табличное представление автоматически обновляет свое содержимое. И вот тут возникает нестыковка. Количество исходных строк – 5. Количество строк после удаления уменьшилось на 1 и составило 4. При этом метод **numberOfRowsInSection** для первой секции все еще возвращает 5, так как данные в свойстве **tasks** не были изменены. Получается, что в секции с одной стороны должно быть 4 строки, так как одна была удалена, а с другой – 5, так как в свойстве **tasks** содержатся старые данные.

Именно об этом и говорится в тексте ошибки: количество строк в секции (т.е. возвращаемых методом **numberOfRowsInSection**) после удаления должно быть равным исходному количеству строк минус количество удаленных строк (т.е. $5 - 1 = 4$).

Примечание Уделите особое внимание этой проблеме, поскольку вы неоднократно столкнетесь с ней в процессе работы iOS-разработчиком. Стоит помнить, что в данном случае мы используем метод **deleteRows** для удаления строк, но еще существует метод **insertRows**, добавляющий строки. При его использовании также может произойти подобная ситуация.

Как исправить данную ошибку? Все довольно просто: помимо строки в таблице необходимо удалить задачу и в свойстве **tasks**.

► Доработайте метод **commit** в соответствии с листингом 13.6.

ЛИСТИНГ 13.6

```
let taskType = sectionsTypesPosition[indexPath.section]
// удаляем задачу
tasks[taskType]?.remove(at: indexPath.row)
// удаляем строку, соответствующую задаче
tableView.deleteRows(at: [indexPath], with: .automatic)
```

► Запустите приложение и протестируйте функцию удаления задач.

Теперь функция удаления работает корректно. Кстати, сейчас вам стал доступен свайп влево для удаления строки (хотя в коде он явно нереализован).

13.4 Сортировка задач с помощью режима редактирования

Еще одной полезной функцией приложения, доступ к которой будет осуществляться в режиме редактирования, будет сортировка задач путем перемещения строк таблицы. Для изменения порядка задач потребуются перевести табличное представление в режим редактирования. После этого переместите необходимую строку, нажав на иконку с тремя параллельными линиями (рис. 13.4), на новую позицию. Задачи могут быть перенесены как в пределах одной секции, так и между ними. Для реализации этой функциональности вам потребуется написать всего несколько строк кода, так как Swift имеет уже все необходимое для этого.

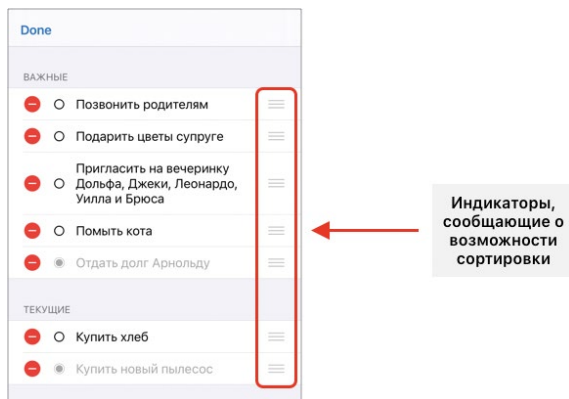


Рис. 13.4. Сортировка строк в режиме редактирования

Наличие функции сортировки определяет источник данных табличного представления. При этом у вас есть возможность определить доступность данной функции для каждой отдельной строки в таблице. Для этого используется метод **canMoveRowAt**.

СИНТАКСИС

Метод `UITableViewDataSource.tableView(_:canMoveRowAt:) -> Bool`

Определяет возможность изменения позиции строки в табличном представлении.

Аргументы

- `_`: `UITableView` – табличное представление, в котором производится сортировка.
- `canMoveRowAt: IndexPath` – путь к строке таблицы, для которой определяется возможность изменения позиции.
 - `indexPath.section` – индекс секции.
 - `indexPath.row` – индекс строки.

Возвращаемое значение

- `Bool` – определяет возможность изменения позиции строки.

Пример

Разрешить перемещать все строки, кроме первой в первой секции.

```
override func tableView(_ tableView: UITableView, canMoveRowAt indexPath:
IndexPath) -> Bool {
    if indexPath.section == 0 && indexPath.row == 0 {
        return false
    }
    return true
}
```

По умолчанию позиция любой строки в табличном представлении может быть изменена, поэтому, если ваш проект не предполагает другой логики, нет никакой необходимости в реализации метода **canMoveRowAt**, возвращающего **true** для всех строк таблицы:

```
override func tableView(_ tableView: UITableView, canMoveRowAt indexPath:
IndexPath) -> Bool {
    return true
}
```

Для того, чтобы в таблице появились индикаторы перемещения, в источнике данных необходимо реализовать метод **moveRowAt**, который вызывается после окончания перемещения строки, то есть сразу после того, как пользователь переместил строку на новую позицию и отпустил палец.

СИНТАКСИС

Метод `UITableViewDataSource.tableView(_:moveRowAt:to:)`

Вызывается при изменении позиции строки с `moveRowAt` на `to`.

Аргументы

- `_`: `UITableView` – табличное представление, в котором производится перемещение строки.
- `moveRowAt`: `IndexPath` – начальная позиция строки.
 - `indexPath.section` – индекс секции.
 - `indexPath.row` – индекс строки.
- `to`: `IndexPath` – конечная позиция строки.
 - `indexPath.section` – индекс секции.
 - `indexPath.row` – индекс строки.

► В классе **TaskListController** реализуйте метод **moveRowAt** (листинг 13.7).

ЛИСТИНГ 13.7

```
// ручная сортировка списка задач
override func tableView(_ tableView: UITableView, moveRowAt
sourceIndexPath: IndexPath, to destinationIndexPath: IndexPath) {
    // секция, из которой происходит перемещение
    let taskTypeFrom = sectionsTypesPosition[sourceIndexPath.section]
    // секция, в которую происходит перемещение
    let taskTypeTo = sectionsTypesPosition[destinationIndexPath.section]

    // безопасно извлекаем задачу, тем самым копируем ее
    guard let movedTask = tasks[taskTypeFrom]?[sourceIndexPath.row] else {
        return
    }

    // удаляем задачу с места, от куда она перенесена
    tasks[taskTypeFrom]!.remove(at: sourceIndexPath.row)
    // вставляем задачу на новую позицию
    tasks[taskTypeTo]!.insert(movedTask, at: destinationIndexPath.row)
    // если секция изменилась, изменяем тип задачи в соответствии с новой
    // позицией
    if taskTypeFrom != taskTypeTo {
        tasks[taskTypeTo]![destinationIndexPath.row].type = taskTypeTo
    }

    // обновляем данные
    tableView.reloadData()
}
```

Я специально добавил в листинг большое количество комментариев, чтобы вы могли самостоятельно изучить его. Особое внимание стоит обратить на порядок переноса задачи внутри свойства **tasks**. Сперва создается копия перемеща-

емой задачи (с помощью **guard**), далее задача удаляется из источника, после чего копия вставляется в место назначения.

На этом мы завершаем работу с изменением списка задач. Самое важное, что вы познакомились с режимом редактирования, и, я надеюсь, поняли, что это довольно удобный и функциональный элемент, для реализации которого в вашем проекте потребуется всего несколько строчек кода.

Глава 14.

Создание и изменение задач

В этой главе вы:

- научитесь создавать таблицы на основе статических ячеек;
- узнаете, что такое xib-файлы;
- разработаете ячейку на основе xib-файла.

Текущая версия приложения **«To-Do Manager»** умеет отображать список задач, а также предоставляет возможность их удаления и сортировки. В результате прохождения этой главы проект обзаведется еще двумя важнейшими функциями, такими как создание новых задач и редактирование уже созданных.

Для реализации описанных функций в приложение будут добавлены несколько новых сцен, а также организованы переходы и передача данных между ними. Всего будет создано две новых сцены: создания/редактирования задачи (рис. 14.1) (обе функции будут реализованы с помощью одного экрана) и выбора типа задачи (рис. 14.2). Реализуя новые возможности, вы изучите два новых для вас типа ячеек: статические ячейки и ячейки, основанные на xib-файлах. Каждый из способов имеет свои достоинства, о которых мы поговорим в ходе изучения материала.

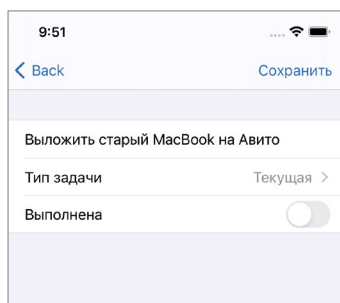


Рис. 14.1. Экран создания и изменения задачи

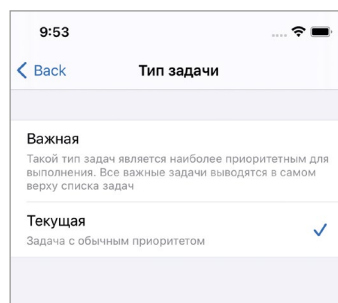


Рис. 14.2. Экран выбора типа задачи

14.1 Экран создания и изменения задачи

Для создания и редактирования задачи будет использоваться одна и та же сцена. Разместим ее на сториборде.

- ▶ Перейдите к файлу **Main.storyboard**.
- ▶ В библиотеке объектов найдите элемент **Bar Buttom Item** и разместите его в правой части навигационной панели главной сцены приложения, отображающей список задач.
- ▶ Добавьте на сториборд новый контроллер табличного представления.
- ▶ Создайте переход от **Bar Buttom Item**, размещенного на первой сцене, к новому контроллеру. Для этого:
 - нажмите клавишу **Control** и перетяните **Bar Buttom Item** на новую сцену (при этом должна отображаться линия синего цвета);
 - в выпадающем окне выберите **Show**.

Теперь, если запустить приложение и нажать на кнопку **Item**, произойдет автоматический переход к новой сцене. Причем переход будет осуществлен в рамках навигационного контроллера.

Текущий вид кнопки **Item** не позволяет пользователю однозначно понять, какую именно функцию она выполняет. Приведем ее к такому виду, чтобы она больше соответствовала выполняемой ею задаче. Сделать это можно несколькими способами: изменив текст кнопки на «**Создать**» или воспользовавшись предустановленным системным стилем. Мы реализуем второй вариант.

- ▶ Выделите **Bar Buttom Item**.
- ▶ Откройте панель **Attributes Inspector**.
- ▶ В поле **System Item** выберите **Add**.

После изменения стиля кнопки ее текст заменился символом «+». Теперь пользователь сразу сможет понять, что при нажатии кнопки будет осуществлено создание новой задачи.

На рисунке 14.3 показан текущий вид сториборда.

Новая сцена будет использоваться как для создания новых, так и для изменения уже существующих задач. При этом порядок действий для осуществления перехода к сцене будет отличаться в зависимости от решаемой задачи (создание или изменение):

- для создания будет использоваться размещенная в навигационной панели кнопка «+»;
- для редактирования – свайп по задаче (его мы еще не реализовали) с последующим выбором действия «Изменить».

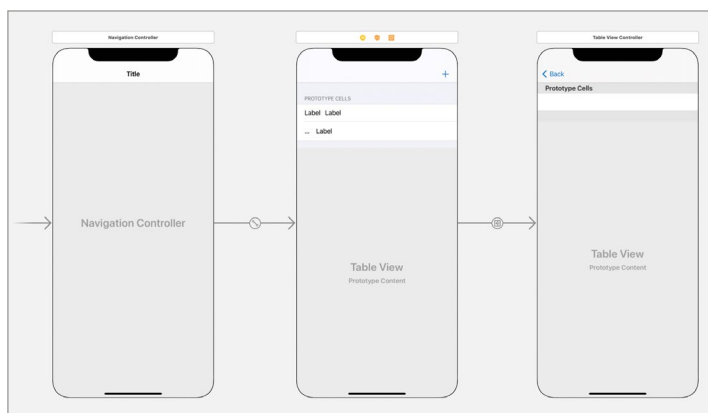


Рис. 14.3. Storyboard с тремя контроллерами

Структура экрана создания

Сущность «Задача» имеет три свойства: название, тип и статус. Все три свойства смогут быть настроены на экране создания. Для каждого из них будет использоваться своя строка табличного представления (рис. 14.4).

Первая строка позволит указать название задачи. В ней будет размещено текстовое поле.

Вторая строка позволит определить тип задачи. При нажатии на ячейку будет происходить переход к следующей сцене, на которой можно выбрать один из двух доступных типов (текущая и важная) (рис. 14.5).

В третьей строке с помощью переключателя будет настраиваться статус задачи («запланирована» или «выполнена»).

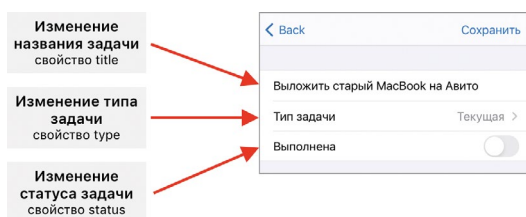


Рис. 14.4. Экран создания/изменения задачи

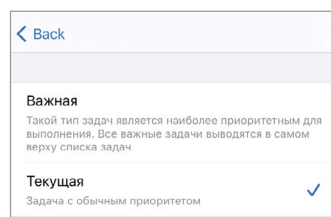


Рис. 14.5. Выбор типа задачи

14.2 Таблица на основе статических ячеек

Вы уже умеете использовать два типа ячеек: доступные по умолчанию и созданные на основе прототипов. В процессе работы над экраном создания задачи мы познакомимся с еще одним способом наполнения таблицы данными – статическими ячейками.

Табличные представления, с которыми мы работали ранее, были динамическими, то есть их строки динамически генерировались на основе каких-либо ячеек. Используя методы источника данных, вы могли управлять количеством секций и строк, их внешним видом и т.д. Таблицы на основе статических ячеек работают иначе: вы изначально верстаете все ячейки таблицы прямо на сцене с помощью **Interface Builder**, и на итоговой сцене каждой статический ячейке будет соответствовать ровно одна строка. Располагаться ячейки будут точно в таком же порядке, который был определен на сцене при их создании. Ни о какой переиспользуемости в данном случае говорить не приходится, да в этом, собственно, и нет необходимости.

Плюс использования статических ячеек заключается в том, что они становятся неотъемлемой частью сцены, существуют на ней в единичном экземпляре (одна сверстанная ячейка соответствует одной строке в таблице), а значит входящие в них элементы могут быть связаны с аутлетами в классе контроллера. Например, вы можете создать аутлет для текстового поля в первой ячейке или для переключателя в третьей.

Использование статических ячеек является подходящим решением, когда таблица будет иметь фиксированный внешний вид. В нашем проекте как раз такой случай: экран создания всегда будет выглядеть одинаково и будет состоять из табличного представления с тремя строками. Поэтому использование таблицы, основанной на статических ячейках, будет прекрасным решением.

- ▶ Откройте файл **Main.storyboard**.
- ▶ Выделите **Table View** в составе сцены создания задачи.
- ▶ Перейдите к панели **Attributes Inspector**.
- ▶ Измените значение поля **Content** с **Dynamic Prototypes** на **Static Cells**.

Теперь в составе табличного представления на сториборде отображаются три статических ячейки (рис. 14.6).

Примечание Если вдруг в вашем случае таблица содержит не три ячейки, вы можете добавить их копированием элемента **Table View Cell** в структуре сцены на панели **Document Outline** или изменить значение свойства **Static Cells** в **Attribute Inspector**.

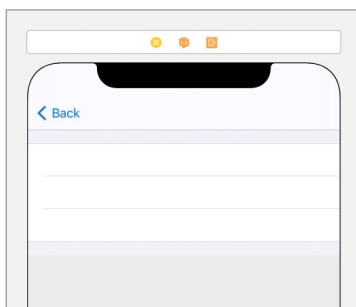


Рис. 14.6. Табличное представление со статическими ячейками

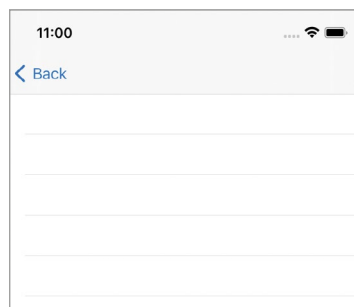


Рис. 14.7. Экран создания таблицы со стилем Plain

Повторюсь: разница между статическими и другими типами ячеек состоит в том, что статичная ячейка будет использоваться только один раз для создания одной строки, а остальные типы могут быть использованы необходимое количество раз (или не использованы вообще).

Примечание Стоит отметить, что и статические ячейки необязательно должны присутствовать в таблице. Дело в том, что даже если Table View основано на статичных ячейках, то за количество строк и секций в любом случае отвечают методы **numberOfSections** и **numberOfRowsInSection**. При этом вы не можете создать в таблице строк больше, чем реализовано статичных ячеек.

Если сейчас запустить приложение, экран создания будет выглядеть, как множество пустых строк (рис. 14.7). Такое мы уже встречали ранее – так работает стиль **Plain**, который определен для таблицы. Он заполняет все свободное пространство пустыми строками.

Изменим внешний вид таблицы, чтобы в ней не отображались лишние элементы.

- ▶ Выделите **Table View** в составе новой сцены.
- ▶ Перейдите к панели **Attributes Inspector**.
- ▶ Измените значение поля **Style** на **Grouped**.
- ▶ Измените значение поля **Background** на **System Grouped Background Color**.

Настройка класса контроллера

В процессе работы с экраном нам потребуется обращаться к элементам в ячейках таблицы для того, чтобы менять или получать их значения. Для организации доступа необходимо создать новый класс и связать с ним контроллер.

- ▶ В составе проекта создайте новый файл с классом **TaskEditController**. Он должен быть дочерним по отношению к **UITableViewController**.

- Свяжите сцену создания/редактирования задачи с классом **TaskEditController**.

В дальнейшем при работе над сценой необходимо помнить, что у нее может быть два сценария использования: создание новой задачи и редактирование уже существующей. Но как лучше реализовать эту функциональность? Давайте порассуждаем и найдем ответы на три важных вопроса.

Вопрос №1. Как сцена будет определять, какая именно операция происходит?

Для сцены, которую мы разработаем, не будет никакой разницы в том, какую операцию выполнять. Все, что она должна делать – это предоставлять механизм, позволяющий определить три значения (название, тип, статус), после чего вернуть их в контроллер вызова.

Вопрос №2. В случае операции редактирования – откуда сцена будет брать данные для наполнения?

Для этого в классе **TaskEditController** создадим три свойства, соответствующие каждому из трех элементов сущности «Задача».

- В классе **TaskEditController** создайте свойства из листинга 14.1.

ЛИСТИНГ 14.1

```
class TaskEditController: UITableViewController {  
    // параметры задачи  
    var taskText: String = ""  
    var taskType: TaskPriority = .normal  
    var taskStatus: TaskStatus = .planned  
    // ...  
}
```

Примечание Набор созданных свойств полностью соответствует элементам сущности «Задача». Поэтому еще одним вариантом решения проблемы было бы создание одного свойства типа **TaskProtocol**, значение которому инициализировалось бы при переходе с экрана со списком задач. И уже это значение можно было бы использовать для наполнения полей и редактирования задачи.

Вопрос №3. Как использовать данные из свойств **taskText**, **taskType** и **taskStatus** из контроллера **TaskEditController** для изменения списка задач в контроллере **TaskListController**?

Вопрос об изменении списка задач лежит вне плоскости контроллера **TaskEditController**. Все, что он должен сделать – вернуть новые значения. Задача обновления списка (добавления новой задачи или замены редактируемой) – это функционал **TaskListController**. В одной из предыдущих глав мы рассмотрели несколько способов передачи информации между контроллерами, и в этом случае применим один из них для того, чтобы **TaskEditController** просто возвращал в **TaskListController** значения, а не думал об обновлении итоговой коллекции задач.

Для решения этого вопроса реализуем передачу данных с использованием замыкания.

- В классе **TaskEditController** создайте свойство **doAfterEdit** (листинг 14.2).

ЛИСТИНГ 14.2

```
var doAfterEdit: ((String, TaskPriority, TaskStatus) -> Void)?
```

Всякий раз при переходе к экрану создания свойству **doAfterEdit** будет инициализироваться замыкание, определяющее дальнейшую судьбу измененных значений. Данное замыкание будет захватывать ссылку на контроллер **TaskListController** и вносить в свойство **tasks** необходимые исправления (добавлять или изменять элемент).

Настройка количества строк

Несмотря на то, что мы используем таблицу, основанную на статических ячейках, для нее также требуется реализовать методы, определяющие количество секций и строк.

- В классе **TaskEditController** реализуйте методы **numberOfSections** и **numberOfRowsInSection** (листинг 14.3).

Примечание Если вы создавали класс **TaskEditController** на основе шаблона **Cocoa Touch Class**, то в нем уже содержатся указанные методы. Вам остается лишь изменить возвращаемое значение.

ЛИСТИНГ 14.3

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}
override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {
    return 3
}
```

Теперь после запуска приложения и открытия сцены на ней будут отображаться три пустые строки (рис. 14.8).

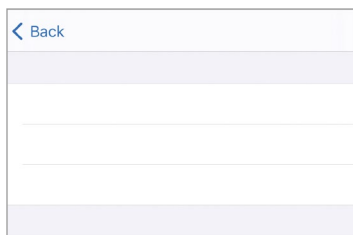


Рис. 14.8. Экран создания задачи

14.3 Ячейка для названия задачи

В первой ячейке будет задаваться название задачи, поэтому на ней требуется разместить текстовое поле.

- ▶ Перейдите к файлу **Main.storyboard**.
- ▶ Разместите в первой ячейке таблицы графический элемент **Text Field**.
- ▶ Создайте ограничения на отступ в 0 точек со всех сторон текстового поля. Убедитесь, что при создании ограничений вы отметили пункт **Constrain to margins**.

В результате проделанных действий в ячейке появилось текстовое поле, растянутое с помощью констрейнтов (рис. 14.9).

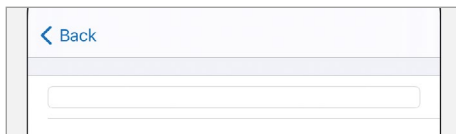


Рис. 14.9. Ячейка с текстовым полем

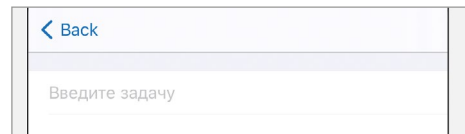


Рис. 14.10. Ячейка для ввода названия задачи

Доработаем внешний вид текстового поля.

- ▶ Выделите **Text Field**.
- ▶ На панели **Attributes Inspector** измените значение поля **Placeholder** на «Введите задачу».
- ▶ Измените значение поля **Border Style** на **None**.
- ▶ Измените размер шрифта (поле **Font**) на **17**.

Теперь ячейка стала выглядеть значительно лучше (рис. 14.10).

Для того, чтобы у нас появилась возможность доступа к полю из кода, необходимо создать соответствующий аутлет.

- ▶ В классе **TaskEditController** объявите аутлет **taskTitle** (листинг 14.4).

ЛИСТИНГ 14.4

```
@IBOutlet var taskTitle: UITextField!
```

- ▶ Свяжите Text Field из первой ячейки с аутлетом **taskTitle**. Помните, что в статической таблице все элементы ячеек – это часть сцены, поэтому для создания связи, выделив Text Field и открыв панель **Connections Inspector**, перетяните **New Referencing Outlet** на значок вью контроллера **TaskEditController**. Таким образом вы связываете элемент ячейки с аутлетом класса контроллера.

В случае, если переход к сцене будет произведен с целью редактирования задачи, ее название должно отобразиться в текстовом поле сразу после перехода к сцене.

- Дополните метод **viewDidLoad** в соответствии с листингом 14.5.

ЛИСТИНГ 14.5

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    // обновление текстового поля с названием задачи  
    taskTitle?.text = taskText  
}
```

14.4 Ячейка для типа задачи

Следующей в таблице идет ячейка, позволяющая указать тип задачи. Именно по этому значению будет определяться, в какой секции будет выводиться задача на первой сцене.

В данной ячейке будет отображаться текущий тип задачи, а по нажатию на нее будет осуществлен переход к сцене выбора типа. То есть, непосредственно сам выбор будет осуществляться на другой сцене, которую мы вскоре добавим в проект.

В первую очередь, мы разместим и настроим графические элементы ячейки.

- Разместите во второй ячейке две текстовые метки, первую слева, вторую справа (рис. 14.11).



Рис. 14.11. Размещение текстовых меток в ячейке.

- Для левой метки создайте ограничения в 0 точек со всех сторон. Убедитесь, что пункт **Constrain to margins** активирован.
- Для правой метки создайте точно такие же ограничения.

Сейчас метки прижаты одна к другой, а в **Document Outline** отображается ошибка, с которой мы встречались, когда верстали ячейки для экрана со списком задач. Напоминаю: проблема состоит в том, что обе метки имеют одинаковый приоритет сопротивления расширению, и Xcode не может однозначно определить итоговую ширину меток.

- Для левой метки измените значение **Horizontal Content Hugging Priority** на 252. Сделать это можно на панели **Size Inspector**.

Теперь левая метка имеет более высокий приоритет, то есть сопротивляется расширению сильнее, вследствие чего правая метка занимает все доступное в ячейке пространство. При этом ошибка, отображаемая ранее в **Document Outline**, исчезла.

Внесем финальные правки во внешний вид ячейки.

- ▶ Для правой метки измените цвет текста на **System Gray Color** и выровняйте текст по правому краю.
- ▶ Выделите ячейку. Для этого щелкните по второму по счету элементу **Table View Cell** в **Document Outline** в составе сцены (рис. 14.12).

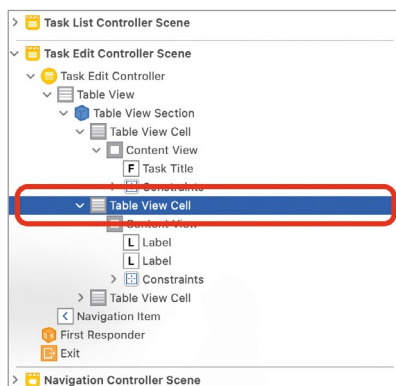


Рис. 14.12. Выбор ячейки в составе таблицы

- ▶ На панели **Attributes Inspector** измените значение свойства **Accessory** на **Disclosure Indicator**.

Свойство **Accessory** позволяет указать стиль вспомогательной иконки, отображаемой в правой части ячейки. На выбор вам доступны 4 значения, и по умолчанию иконка не выводится. **Disclosure Indicator** – это правая угловая скобка, которая обычно используется для обозначения возможности перехода к следующему экрану при нажатии на ячейку таблицы.

- ▶ Измените текст левой метки на «**Тип задачи**».

Итоговый внешний вид ячейки показан на рисунке 14.13.

Правая метка в ячейке предназначена для вывода текущего типа задачи. По этой причине для доступа к ней необходимо создать аутлет.



Рис. 14.13. Итоговый вид ячейки

- В классе **TaskEditController** создайте аутилет **taskTypeLabel** (листинг 14.6).

ЛИСТИНГ 14.6

```
@IBOutlet var taskTypeLabel: UILabel!
```

- Свяжите созданный аутилет с текстовой меткой, расположенной в правой части ячейки.

Прежде, чем сцена будет отображена на экране, необходимо обновить значение в правой текстовой метке. Для этого, в первую очередь, создадим словарь соответствия типа и строкового значения, которое должно быть отображено в правой текстовой метке.

- В классе **TaskEditController** реализуйте приватное свойство **taskTitles** (листинг 14.7). Оно будет определять соответствие типа задачи и текста, выводимого во второй ячейке.

ЛИСТИНГ 14.7

```
// Название типов задач
private var taskTitles: [TaskPriority:String] = [
    .important: "Важная",
    .normal: "Текущая"
]
```

- В методе **viewDidLoad** реализуйте обновление метки в соответствии с текущим типом (листинг 14.8).

ЛИСТИНГ 14.8

```
override func viewDidLoad() {
    // ...

    // обновление метки в соответствии текущим типом
    taskTypeLabel?.text = taskTitles[taskType]
}
```

14.5 Создание экрана выбора типа задачи

Для изменения типа задачи будет использоваться отдельная сцена (рис. 14.14), переход к которой будет осуществляться по нажатию на центральную ячейку на экране создания. На сцене будут отображаться все доступные в данный момент типы, при этом каждый из них будет содержать название и описание, а выбранный — дополнительно отмечаться с помощью галочки в правой части ячейки.

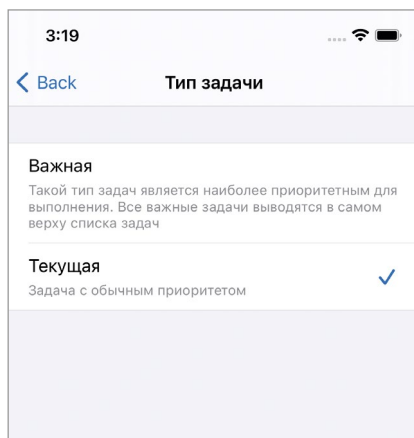


Рис. 14.14. Внешний вид экрана выбора типа

Создадим и настроим новую сцену.

- ▶ На сториборде разместите новый контроллер табличного представления.
- ▶ Из его табличного представления удалите все прототипы. Сделать это можно либо в **Document Outline**, либо с помощью свойства **Prototype Cells** на панели **Attributes Inspector**.
- ▶ Измените стиль табличного представления (свойство **Style** на панели **Attributes Inspector**) на **Grouped**.
- ▶ Измените фоновый цвет табличного представления на **System Grouped Background Color**.
- ▶ Создайте новый файл с классом **TaskTypeController** (наследник **UITableViewViewController**) и свяжите его с созданным контроллером.

Переход к новой сцене должен осуществляться по нажатию на вторую ячейку на экране создания задачи. Для его создания необходимо выполнить следующие действия.

- ▶ Выделите вторую ячейку в составе Table View на сцене создания задачи. Сделать это можно на панели **Document Outline**.
- ▶ С зажатой клавишей **Control** перетяните ячейку на сцену выбора типа. При этом должна отображаться синяя линия (рис. 14.15).
- ▶ Во всплывающем окне выберите пункт **Show**, находящийся в разделе **Selection Segue**.

Теперь ячейка связана с новой сценой и при нажатии на нее будет осуществляться переход. Как и ранее, смена контроллера будет происходить внутри навигационного стека.

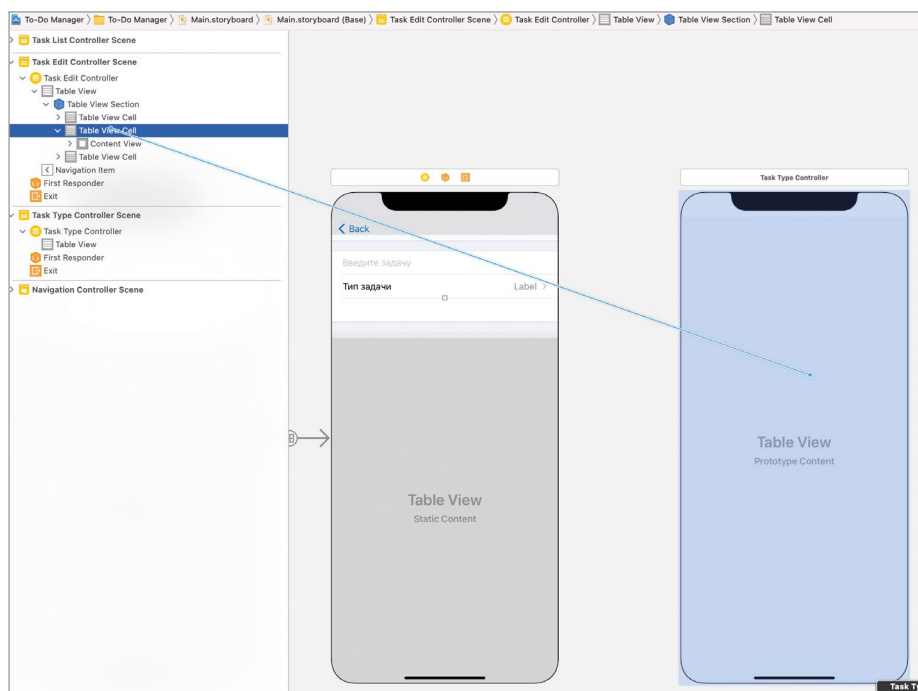


Рис. 14.15. Создание перехода по нажатию на строку

Xib-файлы

Одним из вариантов создания кастомных элементов в Xcode является использование xib-файлов – специальных файлов, предоставляющих разработчику возможность верстки графических элементов средствами **Interface Builder**. XIB расшифровывается как XML Interface Builder, так как на уровне файловой системы он представляет из себя обычный текстовый файл с данными стандарта XML.

Примечание В документации и статьях xib-файлы часто называются nib-файлами. Функционально оба типа файлов одинаковые, но начиная с Xcode 3 в составе проекта используется именно xib, который, в отличие от двоичного nib, является более удобным для разработчика. Тем не менее, исторически сложилось, что xib часто называют nib.

Окей, xib позволяет создавать графический интерфейс, но, стоп... Разве не для этого используются storyboard? Даже в структуре проекта файлы форматов xib и storyboard выглядят довольно похоже (рис. 14.16). Да, оба типа файлов служат для одного и того же, но у них принципиально разное предназначение:

- основная задача storyboard – это описание множества сцен приложения (вью контроллеров) и связей между ними (segue);

- основная задача `xib` – это описание одного графического элемента (корневого вью сцены, табличного представления, кнопки и т.д.), включая все вложенные в него представления.

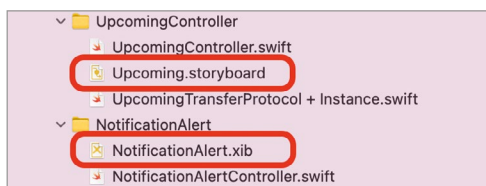


Рис. 14.16. Xib и storyboard файлы в составе приложения Subs Tracker

С помощью `xib` у вас появляется возможность создания собственных графических элементов, которые в дальнейшем могут быть многократно переиспользованы в проекте, например, кастомной кнопки или переключателя. Обычно вместе с `xib`-файлом создается и связанный с ним кастомный класс. На рисунке 14.17 схематично показан пример создания кастомной кнопки и использования ее в нескольких местах в проекте.



Рис. 14.17. Схематичное представление кастомного элемента в составе проекта

Для примера рассмотрим кастомный элемент в одном из реальных приложений, основной задачей которого является формирование плана питания для пользователя. Дизайн приложения предусматривает использование множества графических элементов, внешний вид которых отличается от доступных по умолчанию в **UIKit**. Например, все кнопки в приложении должны иметь закругленные углы, нестандартный увеличенный размер текста, «фирменный» фоновый цвет. Точно такая же ситуация и с ячейками табличных представлений.

Одна из ячеек, используемых в приложении, предусматривает вывод информации об одном блюде в составе плана питания. При этом данная ячейка используется многократно на различных экранах в различных табличных представлениях. Для удобства этот элемент был создан с использованием xib-файла и кастомного класса (рис. 14.18). Такой подход позволил создать элемент всего один раз, а не создавать прототипы ячейки в каждом табличном представлении.

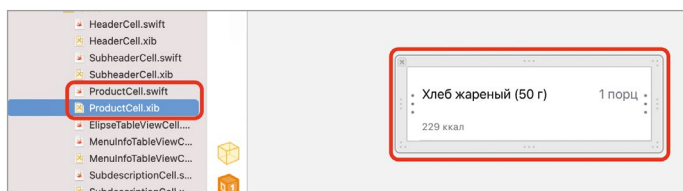


Рис. 14.18. Ячейка, созданная с помощью xib и кастомного класса

Ячейка на основе xib

Экран выбора типа задачи будет отображать строки в соответствии с доступными в приложении типами. Каждая строка будет содержать название и описание одного типа (рис. 14.19).

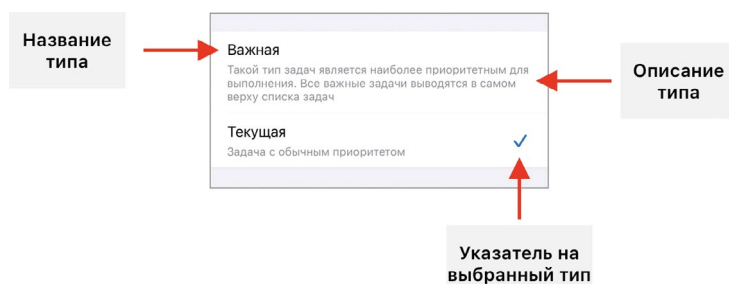


Рис. 14.19. Ячейка на экране выбора типа

Для разработки ячейки мы вполне могли бы воспользоваться функцией создания прототипа, но вместо этого рассмотрим способ использования xib-файла. Данный подход очень часто используется программистами, так как такой тип ячеек может использоваться в любых табличных представлениях в вашем проекте, в то время, как прототипы используются только в том Table View, в котором находятся.

Примечание Хочу еще раз отметить, что xib-файлы позволяют верстать любые графические элементы, будь то кнопка, текстовая метка, всплывающее окно и т.д.

- ▶ В папке **View\Cells** создайте на основе шаблона **Cocoa Touch Class** новый класс **TaskTypeCell** (наследник **UITableViewCell**). При этом обязательно установите галочку **Also create XIB file** (рис. 14.20).

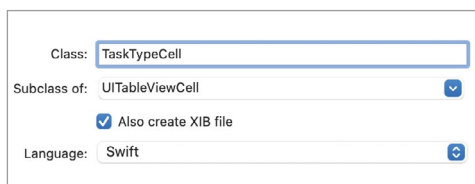


Рис. 14.20. Создание новой ячейки

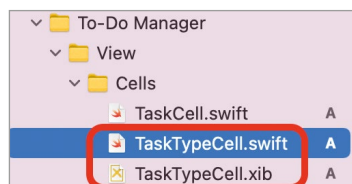


Рис. 14.21. Новые файлы в составе проекта

После этого в составе проекта появятся два новых файла: **TaskTypeCell.swift** и **TaskTypeCell.xib**. (рис. 14.21).

Посмотрим, что из себя представляет xib-файл.

► В составе проекта выберите файл **TaskTypeCell.xib**.

Перед вами откроется **Interface Builder** (рис. 14.22).

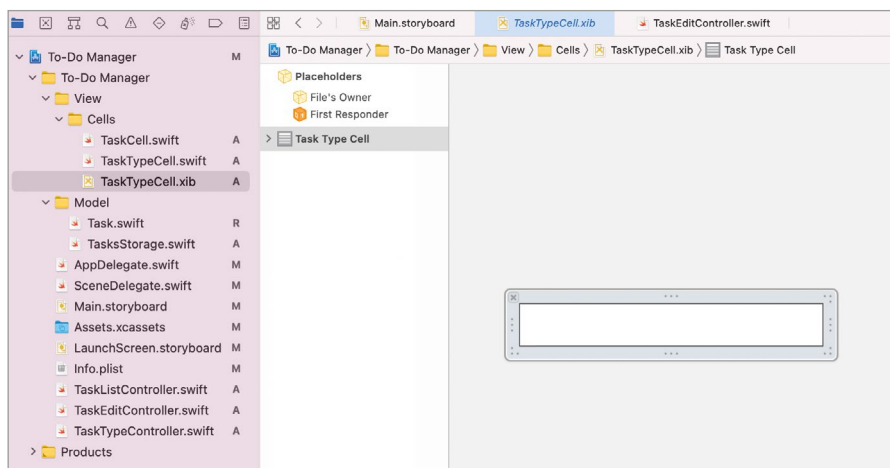


Рис. 14.22. Интерфейс xib-файла

В центральной части **Interface Builder** отображается пустая ячейка, которая является основой для будущей ячейки. В левой части **Interface Builder** отображается **Document Outline**, отображающий пока еще пустую структуру ячейки, а также несколько вспомогательных элементов.

Примечание Возможно, внешний вид панели **Document Outline** в вашем случае отличается от того, что показан на рисунке. В этом случае вам достаточно просто растянуть панель вправо, чтобы увидеть ее полный вариант.

Ячейка будет состоять из двух текстовых меток: одна для вывода названия типа, а вторая – для описания.

► Разместите в ячейке две текстовые метки одна под другой (рис. 14.23).

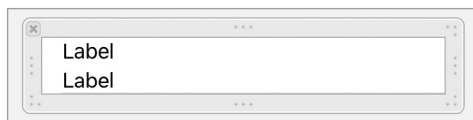


Рис. 14.23. Текстовые метки в ячейке

- ▶ Измените текст верхней метки на «**Название типа**», а нижней — на «**Описание типа**».
- ▶ Измените размер текста нижней метки на 13, а цвет текста — на **System Gray Color**.
- ▶ Для верхней метки создайте следующие констрейнты (рис. 14.24):
 - 0 точек слева, сверху и справа;
 - 5 точек снизу.
- ▶ Убедитесь, что пункт **Constrain to margins** отмечен.

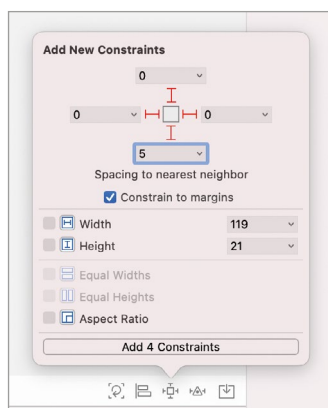


Рис. 14.24. Создание ограничений для верхней метки

Вследствие созданных констрейнтов верхняя метка сместилась немного вниз, а нижняя практически скрылась за границей ячейки (рис. 14.25).

- ▶ Увеличьте размер ячейки до 70 точек в высоту. Для этого ее нижнюю грань можно потянуть вниз, нажав на область с тремя точками.
- ▶ Для нижней метки создайте констрейнты в 0 точек слева, справа и снизу (рис. 14.26). Не забывайте про пункт **Constrain to margins**.

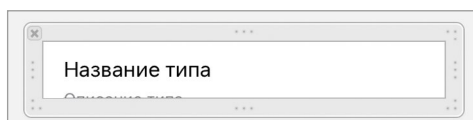


Рис. 14.25. Внешний вид ячейки

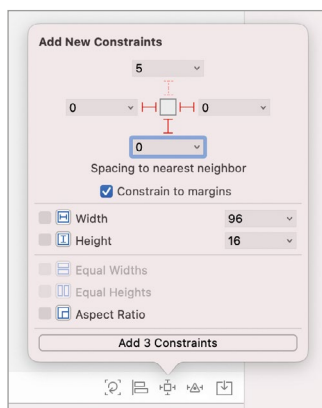


Рис. 14.26. Создание ограничений для нижней метки

И вновь в **Document Outline** вы можете увидеть сообщение о наличии критической ошибки. Как и в прошлый раз, она связана с позиционированием элементов, а точнее с невозможностью однозначно определить размеры текстовых меток, размещенных в ячейке

- ▶ Для верхней метки измените значение свойства **Vertical Content Hugging Priority** на **252** (найти его можно на панели **Size Inspector**).

Метка с описанием типа может содержать произвольное (и даже довольно большое) количество текста, для размещения которого может потребоваться разбить его на несколько строк. При этом размеры ячейки также должны изменяться соответствующим образом, увеличиваясь в высоту и позволяя всему контенту отображаться на экране.

- ▶ Для нижней текстовой метки измените значение свойства **Lines**, доступного на панели **Attributes Inspector**, на 0.
- ▶ На панели **Document Outline** выделите ячейку (элемент **Task Type Cell**).
- ▶ Откройте панель **Size Inspector** и активируйте пункт **Automatic** возле поля **Row Height** (рис. 14.27).

Теперь текст в нижней метке при необходимости автоматически разделится на строки, а ячейка при этом изменит свой размер таким образом, чтобы весь контент мог отображаться на экране.

При создании кастомного класса ячейки вы указали пункт **Also create XIB file**, который привел к созданию xib-файла вместе со swift-файлом, содержащим класс (рис. 14.21). По этой причине ячейка в xib-файле уже связана с классом **TaskTypeCell**, а значит нам не нужно беспокоиться о создании связи между ними.

Добавим в класс **TaskTypeCell** аутлеты для обеих текстовых меток, чтобы при заполнении таблицы данными мы могли изменять их значения.

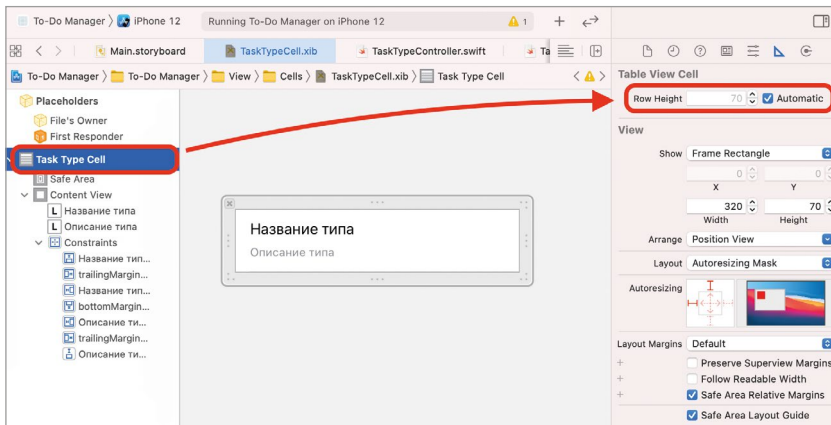


Рис. 14.27. Установка автоматического расчета размера высоты ячейки

- В классе **TaskTypeCell** создайте два аутлета для меток (листинг 14.9).

ЛИСТИНГ 14.9

```
@IBOutlet var typeTitle: UILabel!
@IBOutlet var typeDescription: UILabel!
```

- Свяжите верхнюю метку с аутлетом **typeTitle**, для этого:
 - выделите метку;
 - откройте панель **Connections Inspector**;
 - перетяните круг справа от **New Reference Outlet** на элемент **Task Type Cell** в **Document Outline** (рис. 14.28);
 - во всплывающем окне выберите **typeTitle**.

Примечание Возможно, что у вас все еще могут возникать сложности в вопросе создания аутлетов: что и на какой элемент перетягивать, между чем создавать связь, почему в данном случае мы перетягивали **New Reference Outlet** на **Type Task Cell** – все это вполне естественные вопросы для начинающего разработчика.

Создавая связь, всегда думайте о том, где и в каком классе находится аутлет-свойство, с которым связывается графический элемент. Если оно находится в классе контроллера, перетягивайте на значок контроллера; если оно (как в данном случае) находится в классе ячейки, перетягивайте на значок ячейки.

- Таким же образом свяжите нижнюю метку со свойством **typeDescription**.

Использование созданной ячейки

Ячейка готова, следующим шагом будет наполнение табличного представления данными.

- В классе **TaskTypeController** реализуйте свойства из листинга 14.10.

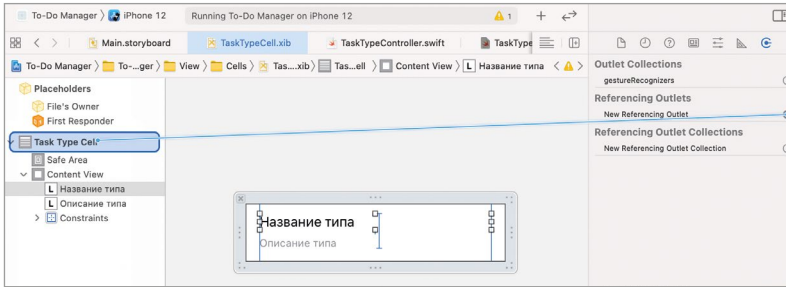


Рис. 14.28. Создание связи между меткой и аутлетом

ЛИСТИНГ 14.10

```
class TaskTypeController: UITableViewController {

    // 1. кортеж, описывающий тип задачи
    typealias TypeCellDescription = (type: TaskPriority, title: String,
    description: String)

    // 2. коллекция доступных типов задач с их описанием
    private var taskTypesInformation: [TypeCellDescription] = [
        (type: .important, title: "Важная", description: "Такой тип задач
является наиболее приоритетным для выполнения. Все важные задачи выводятся
в самом верху списка задач"),
        (type: .normal, title: "Текущая", description: "Задача с обычным
приоритетом")
    ]

    // 3. выбранный приоритет
    var selectedType: TaskPriority = .normal

    // ...
}
```

Разберем листинг.

1. Кортеж (**type: TaskPriority, title: String, description: String**) позволит описать типы задач. С его помощью в дальнейшем таблица будет наполняться данными. Он содержит информацию о конкретном типе задачи (элемент **type**), его названии (элемент **title**), а также текстовом описании (элемент **description**). Для удобства работы с данным кортежем создан алиас **TypeCellDescription**.
2. Свойство **taskTypesInformation** содержит массив доступных для выбора типов. Каждый элемент массива будет соответствовать одной строке в таблице.

3. Свойство **selectedType** используется для определения выбранного значения. Соответствующая ему строка таблицы (на основе данных свойства **taskTypesInformation**) будет отмечаться галочкой в табличном представлении.

Далее добавим информацию о количестве строк и секций.

- Добавьте (или измените, если они уже есть в классе) методы **numberOfSections** и **numberOfRowsInSection** в соответствии с листингом 14.11.

ЛИСТИНГ 14.11

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}

override func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
    return taskTypesInformation.count
}
```

В отличие от статических ячеек и прототипов, ячейки, основанные на `xib`, для их использования в табличном представлении требуют регистрации с помощью специального метода **register**. Это связано с тем, что такой тип ячеек существует отдельно и независимо от Table View, и для того, чтобы он узнал о ней, ему необходимо сообщить, что кастомная ячейка существует.

- В методе **viewDidLoad** класса **TaskTypeController** произведите регистрацию созданной ячейки с помощью метода **register** (листинг 14.12).

ЛИСТИНГ 14.12

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

    // 1. получение значение типа UINib, соответствующее xib-файлу кастомной ячейки
    let cellTypeNib = UINib(nibName: "TaskTypeCell", bundle: nil)
    // 2. регистрация кастомной ячейки в табличном представлении
    tableView.register(cellTypeNib, forCellReuseIdentifier: "TaskTypeCell")
}
```

Метод **viewDidLoad** срабатывает до того, как табличное представление начинает использовать ячейки для наполнения себя данными. По этой причине данный метод прекрасно подходит для того, чтобы произвести регистрацию кастомной ячейки.

Разберем код метода.

1. Тип данных **UINib** используется для программного описания сущности «XIB-файл». Он очень похож на использованный ранее **UIStoryboard**. Передавая в инициализатор название xib-файла в составе проекта (ячейка описана в файле **TaskTypeCell.xib**), мы получаем значение типа **UINib**, описывающее этот файл, и, соответственно, все размещенные в нем view.

2. После того, как xib-файл загружен, ячейка может быть зарегистрирована с помощью метода **register**. Данный метод вызывается для экземпляра табличного представления, размещенного на сцене. В качестве аргумента в метод передаются экземпляр типа **UINib**, в который загружена ячейка, и строковый идентификатор (reusable identifier).

После регистрации ячейка может быть использована в методе **cellForRowAt** для наполнения строк таблицы.

► В классе **TaskTypeController** реализуйте метод **cellForRowAt** (листинг 14.13).

ЛИСТИНГ 14.13

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    // 1. получение переиспользуемой кастомной ячейки по ее идентификатору
    let cell = tableView.dequeueReusableCell(withIdentifier: "TaskTypeCell",
for: indexPath) as! TaskTypeCell

    // 2. получаем текущий элемент, информация о котором должна быть выведе-
на в строке
    let typeDescription = taskTypesInformation[indexPath.row]

    // 3. заполняем ячейку данными
    cell.typeTitle.text = typeDescription.title
    cell.typeDescription.text = typeDescription.description

    // 4. если тип является выбранным, то отмечаем галочкой
    if selectedType == typeDescription.type {
        cell.accessoryType = .checkmark
    } // в ином случае снимаем отметку
    else {
        cell.accessoryType = .none
    }
    return cell
}
```

Разберем код метода.

1. При вызове метода **register** для ячейки был указан идентификатор **TaskTypeCell**, с помощью которого метод **dequeueReusableCell** возвращает переиспользуемый экземпляр кастомной ячейки. Приведение к типу **TaskTypeCell** необходимо для того, чтобы обеспечить доступ к созданным аутлетам.

2. На основе индекса строки (**`indexPath.row`**) возвращается элемент, информация о котором должна быть выведена в ячейке.
3. Текстовым меткам в составе ячейки присваиваются значения. Причем для доступа к самим меткам использованы аутлет-свойства, объявленные в классе **`TaskTypeCell`**.
4. Свойство **`accessoryType`** ячейки позволяет определить стиль вспомогательного элемента, выводимого в правой части ячейки. Ранее мы настраивали его с помощью панели **`Attributes Inspector`**, когда добавляли угловую скобку во вторую ячейку на экране создания задачи. Значение **`.checkmark`** добавляет в ячейку галочку, тем самым отмечая ее как активную.

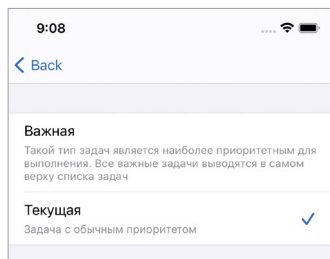


Рис. 14.29. Внешний вид экрана выбора типа

- Выполните запуск приложения и перейдите к экрану выбора типа.

Теперь на экране выбора типа отображается весь перечень доступных значений (рис. 14.29). Причем высота ячеек автоматически изменяется таким образом, чтобы все их содержимое было отображено на экране.

Вот таким, совершенно не хитрым способом, вы можете создавать собственные варианты оформления ячеек с помощью xib-файлов. Создавайте произвольные комбинации графических элементов, используйте различные методы класса **`UITableViewCell`**, и вы сможете реализовать совершенно любую идею.

14.6 Передача данных между сценами

Следующая задача, которую необходимо решить – организовать передачу данных между экраном создания и экраном выбора типа. При этом связь должна быть двухсторонней.

Передача данных от **`TaskTypeController`** к **`TaskEditController`**

В первую очередь, организуем обратную передачу данных, от экрана выбора типа к экрану создания. Данные должны передаваться после того, как будет

осуществлен выбор нового типа, то есть нажата одна из строк таблицы. Новые данные будут использованы на экране создания для отображения в средней ячейке и изменения свойства **taskType** класса **TaskEditController**.

Одним из наиболее удобных вариантов будет использование замыкания. Это приведет к тому, что ответственность за работу с обновленными данными ляжет на вызывающий контроллер.

- В класс **TaskTypeController** добавьте свойства **doAfterTypeSelected** (листинг 14.14).

ЛИСТИНГ 14.14

```
// обработчик выбора типа
var doAfterTypeSelected: ((TaskPriority) -> Void)?
```

Замыкание **doAfterTypeSelected** принимает значение типа **TaskPriority**, указывающее на выбранный тип. Что именно необходимо сделать с этим значением, будет решать контроллер, вызывающий экран выбора (в нашем случае, контроллер создания задачи). В задачи контроллера **TaskTypeController** входит только вывод доступных типов, передача выбранного типа обратно путем вызова замыкания и переход к предыдущему экрану в навигационном стеке.

Для обработки выбора значения воспользуемся методом **didSelectRowAt**. Проще говоря, после нажатия на строку таблицы будет определяться, какое значение выбрано, после чего оно будет передано в замыкание.

- В классе **TaskTypeController** реализуйте метод **didSelectRowAt** (листинг 14.15).

ЛИСТИНГ 14.15

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    // получаем выбранный тип
    let selectedType = taskTypesInformation[indexPath.row].type
    // вызов обработчика
    doAfterTypeSelected?(selectedType)
    // переход к предыдущему экрану
    navigationController?.popViewController(animated: true)
}
```

Передача данных от TaskEditController к TaskTypeController

Теперь реализуем передачу данных от экрана создания. Для корректного функционирования экрана выбора типа при переходе к нему требуется передать

данные о текущем типе (свойство **selectedType**), а также инициализировать значение обработчику выбора (свойство **doAfterTypeSelected**). Так как переход от **TaskEditController** к **TaskTypeController** организован на основе **segue**, наиболее оптимальным вариантом передачи требуемых данных будет использование метода **prepare** в классе **TaskEditController**.

- ▶ Откройте файл **Main.storyboard**.
- ▶ Укажите идентификатор **toTaskTypeScreen** для **segue**, идущему от **TaskEditController** к **TaskTypeController**.
- ▶ В классе **TaskEditController** реализуйте метод **prepare** в соответствии с листингом 14.16.

ЛИСТИНГ 14.16

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "toTaskTypeScreen" {
        // ссылка на контроллер назначения
        let destination = segue.destination as! TaskTypeController
        // передача выбранного типа
        destination.selectedType = taskType
        // передача обработчика выбора типа
        destination.doAfterTypeSelected = { [unowned self] selectedType in
            taskType = selectedType
            // обновляем метку с текущим типом
            taskTypeLabel?.text = taskTitles[taskType]
        }
    }
}
```

- ▶ Запустите приложение и попробуйте осуществить выбор типа на соответствующем экране.

Теперь оба экрана связаны между собой: экран создания всегда отображает актуальное значение выбранного типа, а экран выбора всегда корректно отмечает галочкой выбранный тип.

14.7 Ячейка изменения статуса задачи

На экране создания задачи осталось сверстать одну ячейку, предназначенную для определения статуса (выполнена или запланирована) (рис. 14.30). Смена статуса будет заключаться в установке переключателя (Switch) в одно из двух положений:

- «Переключатель отключен» – задача находится в статусе «запланирована»;

- «Переключатель включен» – задача находится в статусе «выполнена».

В своем составе ячейка будет содержать два графических элемента: текстовую метку и переключатель.



Рис. 14.30. Ячейка для управления статусом задачи

- ▶ В левой части ячейки разместите текстовую метку, а в правой – переключатель (Switch) (рис. 14.31).
- ▶ Измените текст в метке на «**Выполнена**».
- ▶ Для текстовой метки определите отступы в 0 точек со всех сторон.
- ▶ Для переключателя создайте следующие ограничения:
 - центрирование по вертикали;
 - отступ в 0 точек справа.

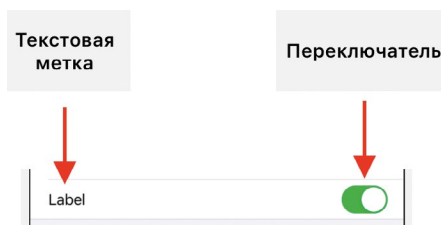


Рис. 14.31. Графические элементы в ячейке

Примечание Если вдруг Label или Switch прижались к самым краям ячейки, вероятно, вы забыли активировать пункт **Constrain to margins**. В этом случае вам потребуется пересоздать констрейнты.

Обратите внимание, что в этот раз после создания констрейнтов Xcode не отобразил сообщений об ошибках позиционирования, вследствие чего не пришлось изменять приоритеты. Дело в том, что переключатель изначально имеет фиксированный размер и не предназначен для расширения или сжатия, а значения его свойств **Content Hugging Priority** по умолчанию равны 750, в то время, как у текстовой метки они равняются 250. По этой причине метка имеет меньшее сопротивление растяжению, а значит в данном случае займет все доступное пространство.

Переключатель будет использоваться для определения текущего статуса задачи, поэтому его необходимо связать с классом контроллера.

- ▶ В классе **TaskEditController** создайте аутлет **taskStatusSwitch** (листинг 14.17) и свяжите его с переключателем в ячейке.

ЛИСТИНГ 14.17

```
// переключатель статуса
@IBOutlet var taskStatusSwitch: UISwitch!
```

Класс **UISwitch** представляет собой программную реализацию переключателя. Он как и все остальные классы с приставкой **UI** входит в состав фреймворка **UIKit**. Переключатель может иметь всего два состояния: **on** (сдвинут вправо, окрашен в зеленый цвет) и **off** (сдвинут влево, окрашен в белый цвет). По умолчанию он находится в состоянии **on**, в нашем случае это означает, что задача будет отмечена как выполненная. Изменим его значение по умолчанию на **off**.

- ▶ Выделите переключатель на сцене.
- ▶ Откройте панель **Attributes Inspector**.
- ▶ Измените значение свойства **State** на **Off**.

После проделанных изменений переключатель соответствующим образом изменил свой внешний вид (рис. 14.32).

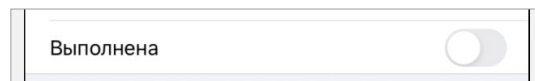


Рис. 14.32. Внешний вид ячейки с отключенным переключателем

Свойство **taskStatus** будет хранить текущий статус задачи. Его значение будет определяться при вызове сцены и использоваться для установки переключателя в необходимое положение.

- ▶ В классе **TaskEditController** дополните метод **viewDidLoad** в соответствии с листингом 14.18.

ЛИСТИНГ 14.18

```
override func viewDidLoad() {
    // ...

    // обновляем статус задачи
    if taskStatus == .completed {
        taskStatusSwitch.isOn = true
    }
}
```

Свойство **isOn** предназначено для получения и изменения текущего положения переключателя. Его тип данных — **Bool**, и, соответственно, при работе с ним вам доступны два значения: **true** для состояния **on** и **false** — для **off**.

- ▶ Запустите приложение и проверьте работу переключателя.

14.8 Сохранение задачи

Экран создания будет выполнять две функции: создание новой задачи и редактирование уже существующей. В обоих случаях сам контроллер и его элементы будут работать совершенно одинаково, но вот доступ к этим функциям будет осуществляться по-разному.

Сохранение новой задачи

Для создания новой задачи, а также сохранения редактируемой, добавим в навигационную панель кнопку «**Сохранить**». При ее нажатии будет вызываться обработчик (свойство **doAfterEdit**), и осуществляться переход к предыдущему экрану.

- ▶ Откройте файл **Main.storyboard**.
- ▶ Добавьте в навигационную панель (в правую ее часть) графический элемент **Bar Button Item**.
- ▶ Измените текст кнопки на «**Сохранить**».
- ▶ В классе **TaskEditController** создайте экшн-метод **saveTask** в соответствии с листингом 14.19.

ЛИСТИНГ 14.19

```
@IBAction func saveTask(_ sender: UIBarButtonItem) {  
    // получаем актуальные значения  
    let title = taskTitle?.text ?? ""  
    let type = taskType  
    let status: TaskStatus = taskStatusSwitch.isOn ? .completed : .planned  
    // вызываем обработчик  
    doAfterEdit?(title, type, status)  
    // возвращаемся к предыдущему экрану  
    navigationController?.popViewController(animated: true)  
}
```

- ▶ Свяжите вызов метода **saveTask** с нажатием кнопки «**Сохранить**».

Экран создания задачи (за исключением функции редактирования существующей задачи) полностью готов, но нажатие на кнопку «**Сохранить**» не приводит к какому-либо эффекту, так как свойство **doAfterEdit**, которое и отвечает за дальнейшую судьбу задачи, не содержит значения. Для решения этого вопроса необходимо обеспечить передачу соответствующего замыкания в данное свойство при переходе к сцене.

Для этого вновь воспользуемся возможностями **segue** и методом **prepare**. На сториборде уже создан переход, активируемый при нажатии кнопки «+» на экране со списком задач.

- Для данного **segue** укажите идентификатор **toCreateScreen**.
- В классе **TaskListController** реализуйте метод **prepare** из листинга 14.20.

ЛИСТИНГ 14.20

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "toCreateScreen" {
        let destination = segue.destination as! TaskEditController
        destination.doAfterEdit = { [unowned self] title, type, status in
            let newTask = Task(title: title, type: type, status: status)
            tasks[type]?.append(newTask)
            tableView.reloadData()
        }
    }
}
```

- Запустите приложение и протестируйте функцию создания задачи.

Реализованная функциональность выполняется без ошибок: задача создается, при этом корректно определяется ее тип и статус.

Сохранение измененной задачи

Для реализации функции изменения задачи нам потребуется внести всего несколько дополнений в уже существующий код. Доступ к редактированию будет осуществляться путем свайпа по строке с задачей. При этом для задач в статусе «выполнено» при свайпе вправо будут отображаться два доступных действия (изменение статуса на «запланировано» и редактирование задачи), а для запланированных – одна (только редактирование) (рис. 14.33).

На storyboard создан всего один segue, позволяющий осуществить переход к сцене контроллера **TaskEditController**. Он вызывается при нажатии кнопки «+», расположенной в навигационной панели. Для редактирования задачи также потребуется осуществить переход к данной сцене. В этом случае будем производить загрузку контроллера назначения и осуществлять переход с помощью программного кода.

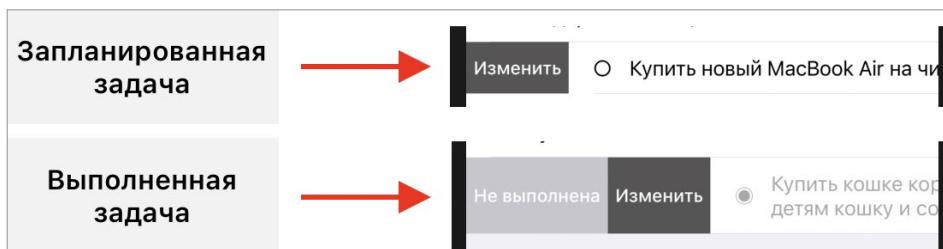


Рис. 14.33. Действия, доступные через свайп по строке с задачей

- Для сцены контроллера **TaskEditController** укажите **Storyboard ID** «**TaskEditController**».

Теперь для сцены указан идентификатор Storyboard ID, а значит она может быть загружена с помощью метода **UIStoryboard.instantiateViewController**.

Добавим в табличное представление сцены **TaskListController** свайп и реализуем переход к сцене редактирования.

- В классе **TaskListController** измените метод **leadingSwipeActionsConfigurationForRowAt** в соответствии с листингом 14.21.

ЛИСТИНГ 14.21

```
override func tableView(_ tableView: UITableView, leadingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration? {
    // получаем данные о задаче, по которой осуществлен свайп
    let taskType = sectionsTypesPosition[indexPath.section]
    guard let _ = tasks[taskType]?[indexPath.row] else {
        return nil
    }

    // действие для изменения статуса на "запланирована"
    let actionSwipeInstance = UIContextualAction(style: .normal, title: "Не выполнена") { _, _, _ in
        self.tasks[taskType]![indexPath.row].status = .planned
        self.tableView.reloadSections(IndexSet(arrayLiteral: indexPath.section), with: .automatic)
    }

    // действие для перехода к экрану редактирования
    let actionEditInstance = UIContextualAction(style: .normal, title: "Изменить") { _, _, _ in
        // загрузка сцены со storyboard
        let editScreen = UIStoryboard(name: "Main", bundle: nil).instantiateViewController(identifier: "TaskEditController") as! TaskEditController
        // передача значений редактируемой задачи
        editScreen.taskText = self.tasks[taskType]![indexPath.row].title
        editScreen.taskType = self.tasks[taskType]![indexPath.row].type
        editScreen.taskStatus = self.tasks[taskType]![indexPath.row].status
        // передача обработчика для сохранения задачи
        editScreen.doAfterEdit = { [selfself] title, type, status in
            let editedTask = Task(title: title, type: type, status: status)
            tasks[taskType]![indexPath.row] = editedTask
            tableView.reloadData()
        }
    }
}
```

```
        // переход к экрану редактирования
        self.navigationController?.pushViewController(editScreen, animated:
true)
    }
    // изменяем цвет фона кнопки с действием
    actionEditInstance.backgroundColor = .darkGray

    // создаем объект, описывающий доступные действия
    // в зависимости от статуса задачи будет отображено 1 или 2 действия
    let actionsConfiguration: UISwipeActionsConfiguration
    if tasks[taskType]![indexPath.row].status == .completed {
        actionsConfiguration = UISwipeActionsConfiguration(actions:
[actionSwipeInstance, actionEditInstance])
    } else {
        actionsConfiguration = UISwipeActionsConfiguration(actions:
[actionEditInstance])
    }

    return actionsConfiguration
}
```

- Запустите приложение и проверьте корректность выполнения функций создания и редактирования задач.

Мы реализовали все основные функциональные возможности приложения: задачи создаются, отображаются и редактируются, данные между сценами успешно передаются. Конечно, у приложения еще остались проблемные моменты, но о них мы поговорим в следующей главе.

Глава 15.

Завершение разработки приложения

В этой главе вы:

- проведете финальную доработку приложения «**To-Do Manager**».

Менеджер задач, работу над которым мы ведем на протяжении всей главы, практически готов к выпуску первой версии. Вся функциональность по работе с задачами уже реализована, и нам осталось произвести последнюю доработку – подключить работу с долговременным хранилищем данных.

15.1 Доработка хранилища задач

Основная проблема приложения состоит в том, что оно все еще работает с тестовым набором данных и не обеспечивает долговременное хранение созданных задач. Для решения этого вопроса обратимся к уже рассмотренному ранее User Defaults.

Работа с хранилищем заключается в выполнении двух операций: «сохранить» и «загрузить». Для каждой из них в протоколе **TasksStorageProtocol**, регламентирующем взаимодействие с хранилищем, определены методы **saveTasks** и **loadTasks**. Первому будет передаваться массив сущностей для их сохранения, а второй будет этот массив возвращать.

Класс **TasksStorage**, подписанный на протокол **TasksStorageProtocol**, в своей текущей реализации является «заглушкой», поэтому проведем его переработку. Код класса будет аналогичен тому, что вы уже видели в главе про локальные хранилища во второй части книги, когда сохраняли и загружали данные о контактах.

- Измените тело класса **TasksStorage** в соответствии с листингом 15.1

ЛИСТИНГ 15.1

```
class TasksStorage: TasksStorageProtocol {  
    // Ссылка на хранилище
```

```

private var storage = UserDefaults.standard
// Ключ, по которому будет происходить сохранение и загрузка хранилища
из User Defaults
var storageKey: String = "tasks"

// Перечисление с ключами для записи в User Defaults
private enum TaskKey: String {
    case title
    case type
    case status
}

func loadTasks() -> [TaskProtocol] {
    var resultTasks: [TaskProtocol] = []
    let tasksFromStorage = storage.array(forKey: storageKey) as?
[[String:String]] ?? []
    for task in tasksFromStorage {
        guard let title = task[TaskKey.title.rawValue],
              let typeRaw = task[TaskKey.type.rawValue],
              let statusRaw = task[TaskKey.status.rawValue] else {
            continue
        }
        let type: TaskPriority = typeRaw == "important" ? .important :
.normal
        let status: TaskStatus = statusRaw == "planned" ? .planned :
.completed

        resultTasks.append(Task(title: title, type: type, status:
status))
    }
    return resultTasks
}

func saveTasks(_ tasks: [TaskProtocol]) {
    var arrayForStorage: [[String:String]] = []
    tasks.forEach { task in
        var newElementForStorage: Dictionary<String, String> = [:]
        newElementForStorage[TaskKey.title.rawValue] = task.title
        newElementForStorage[TaskKey.type.rawValue] = (task.type ==
.important) ? "important" : "normal"
        newElementForStorage[TaskKey.status.rawValue] = (task.status ==
.planned) ? "planned" : "completed"
        arrayForStorage.append(newElementForStorage)
    }
    storage.set(arrayForStorage, forKey: storageKey)
}
}

```

Сейчас в коде контроллера **TaskListController** используется только метод **loadTasks**, осуществляющий загрузку данных из хранилища. Поэтому перед нами возникает задача — добавить в проект сохранение списка созданных задач. Одним из вариантов решения является использование наблюдателя в свойстве **tasks**. Пусть помимо сортировки, также происходит и передача всей коллекции задач в хранилище.

- Добавьте в наблюдатель **didSet** свойства **tasks** функциональность сохранения задач (листинг 15.2).

ЛИСТИНГ 15.2

```
var tasks: [TaskPriority:[TaskProtocol]] = [:] {  
    didSet {  
        // ...  
  
        // сохранение задач  
        var savingArray: [TaskProtocol] = []  
        tasks.forEach { _, value in  
            savingArray += value  
        }  
        tasksStorage.saveTasks(savingArray)  
    }  
}
```

- Запустите приложение, создайте и сохраните несколько задач.
- Закройте (не скройте, а именно завершите выполнение) приложение и откройте его вновь.

Удивительно, но список пуст! В написанном нами коде кроется очень интересная ошибка. Сможете ли вы самостоятельно ее идентифицировать?

Запись в хранилище и загрузка из него работает полностью без ошибок — проблема кроется в стартовом значении свойства **tasks**, которое инициализируется ему при создании экземпляра **TaskListController**. Так как передача данных в User Defaults происходит при каждом создании главной сцены проекта, значение **[:]**, иницируемое свойству **tasks**, передается в хранилище при каждом запуске приложения, перезаписывая уже имеющиеся в нем данные. Получается следующая ситуация:

1. приложение запущено, список задач пуст;
2. задачи создаются, значение свойства **tasks** изменяется, данные записываются в User Defaults;
3. приложение закрывается, данные остаются в User Defaults;
4. приложение запускается, создается экземпляр класса **TaskListController**,

свойству **tasks** инициализируется пустой словарь, который изменяет данные в User Defaults, удаляя все созданные задачи;

5. в результате список задач на экране пуст.

Есть множество вариантов решения данной проблемы. Мы же воспользуемся жизненным циклом приложения и будем загружать и передавать данные из хранилища в контроллер еще в процессе загрузки.

Для этого проведем доработку класса **TaskListController** и реализуем необходимую логику в **SceneDelegate**.

► Реализуйте в классе **TaskListController** метод (листинг 15.3).

ЛИСТИНГ 15.3

```
// получение списка задач, их разбор и установка в свойство tasks
func setTasks(_ tasksCollection: [TaskProtocol]) {
    // подготовка коллекции с задачами
    // будем использовать только те задачи, для которых определена секция
    sectionsTypesPosition.forEach { taskType in
        tasks[taskType] = []
    }
    // загрузка и разбор задач из хранилища
    tasksCollection.forEach { task in
        tasks[task.type]?.append(task)
    }
}
```

► Откройте файл **Main.storyboard** и измените **Storyboard ID** для сцены со списком задач на «**TaskListController**».

► Измените тело метода **willConnectTo** класса **SceneDelegate** в соответствии с листингом 15.4.

ЛИСТИНГ 15.4

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options
connectionOptions: UIScene.ConnectionOptions) {

    guard let windowScene = (scene as? UIWindowScene) else { return }
    window = UIWindow(windowScene: windowScene)
    guard let window = window else {
        return
    }

    // загрузка списка задач
```



```
// ее необходимо выполнить до создания экземпляра класса
TaskListController
// иначе данные будут перезаписаны
let tasks = TasksStorage().loadTasks()

// загрузка сцены со списком задач
let taskListController = UIStoryboard(name: "Main", bundle: nil).instantiateViewController(identifier: "TaskListController") as! TaskListController

//передача списка задач в контроллер
taskListController.setTasks(tasks)

// создание навигационного контроллера
let navigationController = UINavigationController(rootViewController: taskListController)

// отображение сцен
self.window?.windowScene = windowScene
window.rootViewController = navigationController
window.makeKeyAndVisible()
}
```

Приведенный код содержит множество комментариев и не должен вызывать у вас проблем в понимании.

Ошибка исправлена, и вы можете протестировать работу приложения.

► Запустите приложение.

Стартовый экран не содержит ни одной задачи, обе секции полностью пусты и отображают лишь собственные названия.

► Протестируйте всю реализованную функциональность, создав и отредактировав несколько задач.

Так как в приложении реализована функция долговременного хранения задач, все созданные элементы будут отображены после перезагрузки приложения.

► Закройте (не скройте, а именно завершите выполнение) приложение и откройте его вновь.

Все созданные ранее элементы на своих местах! Сохранение и загрузка данных работает.

На этом мы завершаем разработку приложения «**To-Do Manager**». Оно вполне функционально, но все-таки имеет ряд недостатков, о которых мы поговорим в следующем разделе.

15.2 Недостатки приложения To-Do Manager

Повторяющиеся данные

В составе контроллеров приложения есть повторяющиеся данные, например, соответствие типов задач и названий этих типов. Так, в контроллере **TaskEditController** эти данные содержатся в свойстве **taskTitles**, а в контроллере **TaskTypeController** – в свойстве **taskTypesInformation**.

Такой подход может привести к проблемам в случае, если вам потребуется внести правки, затрагивающие эти названия. Таким образом, придется исправлять текст в нескольких элементах проекта. И чем больше элементов будет включать проект, тем больше работы потребуется.

Одним из вариантов решения данной проблемы является использование функций локализации приложения. Для этого в проекте создаются специальные файлы (по одному для каждого языка), в котором определяются ключи и их строковые значения. Далее в проекте вместо конкретных данных вы используете эти самые ключи, при этом один и тот же ключ может использоваться в произвольных местах в коде проекта. То есть, при необходимости изменить название какого-либо элемента потребуется внести правки только в языковой файл.

Излишняя связанность

Большой проблемой с точки зрения архитектуры является слишком большая связанность между компонентами проекта. Об этом, в частности, говорит то, что при осуществлении переходов между сценами используются конкретные вью контроллеры. Например, переходя со сцены **TaskListController** на сцену **TaskEditController** в методе **prepare** используется тайпкастинг к конкретному типу **TaskEditController**, а в методе-обработчике свайпа создается экземпляр этого типа.

Что, если вы решили использовать другой экран создания/редактирования? Как при таком подходе заменить контроллер назначения? Вам потребуется вносить исправления как в схему связи на сториборде, так и во многих других местах в коде проекта. При этом потребуется быть очень внимательным и не забыть объявить все необходимые свойства.

Для решения этой проблемы стоит использовать уже известный вам прием с протоколами. Каждый из контроллеров необходимо скрыть за соответствующим протоколом, определяющим требования к свойствам и методам. При этом для создания экземпляров этих контроллеров можно использовать шаблон про-

ектирования «Фабрика» (или «Абстрактная фабрика»), при котором созданием объектов будет заниматься внешний класс, не относящийся к контроллерам.

В результате, когда требуется осуществить переход к новой сцене, производится следующая последовательность действий:

1. контроллер, из которого происходит переход, обращается к «Фабрике» с просьбой вернуть определенную сцену (а точнее, соответствующий ей контроллер);
2. «Фабрика» возвращает экземпляр, но не конкретного типа, а в качестве типа указывается протокол, которому соответствует контроллер назначения;
3. контроллер-источник инициализирует все необходимые свойства контроллера-назначения и осуществляет переход к нему.

При этом источник даже не предполагает, к какому конкретно контроллеру происходит переход: ему эти знания совершенно не нужны. Все, что требуется от него – это передать данные в соответствии с протоколом.

Пока вы не попробуете самостоятельно реализовать «Фабрику» в одном из своих проектов, вероятнее всего, вы с трудом сможете понять всю мощь данной концепции. Но главное, чтобы вы старались снижать связанность между отдельными элементами, в том числе с использованием протоколов.

Отсутствие необходимых проверок

Экран создания/редактирования задачи прекрасно выполняет возложенные на него функции. Но что произойдет, если вы попробуете создать задачу с пустым названием? Она будет создана, что не очень логично.

Задание В контроллере `TaskEditController` при сохранении задачи добавьте проверку названия. При отсутствии значения отображайте всплывающее окно с соответствующим сообщением (`UIAlertController`).

Обратите внимание, что название может состоять из пробелов, и, с точки зрения наличия значения, оно не будет пустым. То есть, стоит обрезать пробелы слева и справа от текста.

Проблемы интерфейса

Приложение имеет несколько проблем, связанных с графическим интерфейсом.

На сцене создания задачи при нажатии на строку статуса задачи она выделяется серым, и выделение не снимается.

Задание В контроллере `TaskEditController` реализуйте снятие выделения со строки, определяющей статус задачи. Можете выполнить задание, вообще запретив выделение данной строки, или не запрещая, а просто убирая его после выделения.

Второй проблемой, связанной с удобством работы, является полное отсутствие каких-либо уведомлений в случае, если в списке задач на главном экране этих самых задач нет.

Задание На сцене со списком задач добавьте функциональность, при которой в случае отсутствия задач в каком-либо разделе (секции таблицы) отображалась строка, содержащая сообщение «Задачи отсутствуют».

Также использование приложения будет затруднено при активации темной темы в операционной системе (рис. 15.1). Запланированные задачи при этом вообще не читаются, а выполненные слишком хорошо видны.

Для решения проблемы в приложении потребуется внедрить полноценную поддержку темной темы, при которой в зависимости от того, какая тема используется, применяются различные цвета в интерфейсе. То есть для списка задач можно указать какие цвета будут использованы днем, а какие ночью. Другим вариантом решения является полный запрет на ночную тему.

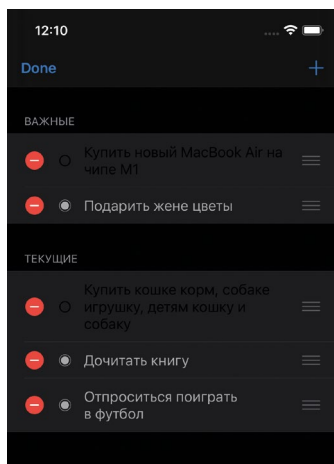


Рис. 15.1. Темная тема в приложении

ИТОГИ ТРЕТЬЕЙ ЧАСТИ

Вы проделали очень большую работу на пути к завершению приложения «**To-Do Manager**»! Табличные представления, в которых теперь вы довольно неплохо ориентируетесь, — это один из наиболее используемых элементов в iOS-разработке. Вы большой молодец, так как теперь за вашими плечами довольно хорошие знания предметной области. Но впереди еще много новых и интересных возможностей. Не откладывайте изучение последней четвертой части, в которой мы займемся разработкой игры «**Cards**», основанной на использовании второго по популярности элемента – Collection View.

Часть IV

ГРАФИЧЕСКИЙ ИНТЕРФЕЙС

ПРОЕКТ «CARDS»

Вы реализовали уже несколько несложных приложений, в процессе работы над которыми изучили вопросы построения архитектуры и функционирования приложения и его компонентов. В этой части мы более подробно поговорим о создании графического интерфейса. В частности, мы рассмотрим, с помощью каких механизмов создаются UI-элементы, как они позиционируются на экране, как обрабатывают касания, а также как анимируются свойства. В самом конце мы получим полностью функциональную игру «Cards» (один из вариантов интерфейса представлен на рисунке 1), часть функций которой вы реализуете самостоятельно.

Наверняка вы играли в подобные игры ранее: ее основной целью является поиск пар одинаковых карточек. В соответствии с правилами, все карточки на игровом поле расположены рубашкой вверх, и пользователь не видит рисунки, изображенные на них. Раз за разом, переворачивая по две карточки, он пытается выбрать одинаковые. Если выбор был сделан верно, карточки остаются лежать рисунком вверх, если неверно – автоматически переворачиваются. Задача состоит в том, чтобы за минимальное количество попыток найти все пары одинаковых карточек.

Знания по отдельным темам, которые вы получите в ходе изучения материала этой части, будут несколько глубже тех, которыми обладают некоторые из Junior Swift Developers, устраиваясь на свою первую работу. Но изучив мате-

риал, не спешите размещать в сети свое резюме, так как если взглянуть на ситуацию в общем, вам предстоит еще многому научиться, прежде чем вы будете готовы покорять сердце потенциального работодателя. Поэтому не откладывайте обучение и не тратьте время впустую — впереди вас ждет заключительная часть книги, содержащая очень много интересного и полезного материала.

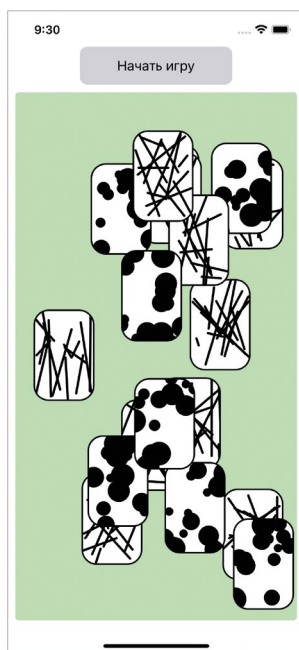


Рис. 1. Интерфейс игры «Cards»



Программный код, написанный в данной части книги, доступен по ссылке:

<https://swiftme.ru/listings21>

Глава 16. Классы UIView и CALayer

Глава 17. Кривые Безье

Глава 18. Создание игровой карточки. Разработка кастомных представлений и слоев

Глава 19. События и анимации в iOS

Глава 20. Разработка игрового поля и Модели игры

Глава 16.

Классы UIView и CALayer

В этой главе вы:

- узнаете, какие фреймворки обеспечивают отображение графического интерфейса;
- разберетесь с тем, что такое точки, и чем они отличаются от пикселей;
- рассмотрите, из чего состоят представления;
- узнаете, как позиционируются графические элементы внутри друг друга;
- научитесь создавать кастомные представления и изменять их внешний вид;
- узнаете, чем отличаются frame и bounds;
- познакомитесь со слоями.

Примечание В данной главе речь пойдет о классе **UIView**, который входит в состав фреймворка **UIKit**. По этой причине все сказанное относится только к тем приложениям и графическим элементам, которые работают на основе данного фреймворка, но не на основе **SwiftUI**.

Класс **UIView** – это основа любого графического элемента. Все кнопки, слайдеры, текстовые метки, табличные представления, с которыми вы работали ранее, основаны на данном классе. Но помимо стандартных сущностей, входящих в состав фреймворка **UIKit**, класс **UIView** предоставляет вам все необходимое для создания и отображения в интерфейсе собственных UI-элементов. В этой главы мы подробно разберем основные возможности, которые предоставляет класс **UIView** для создания собственных графических элементов.

Примечание Напоминаю, что для обозначения экземпляра класса **UIView** или одного из его потомков, мы будем использовать слова «View», «вью» и «представление». Это синонимы.

16.1 Фреймворки UIKit, Core Animation и Core Graphics

UIKit – это прекрасный фреймворк, созданный «разработчиками для разработчиков». Каждый класс, входящий в него, каждый метод или свойство прозрачны, понятны и удобны при использовании. К примеру, достаточно взглянуть на то, как легко с помощью **UIKit** определить красный цвет:

```
UIColor.red
```

Примечание Вы уже знакомы с классом **UIColor** в ходе самостоятельной работы в первой части.

Класс **UIColor** входит в состав **UIKit**. Он позволяет определить произвольный цвет. Для указания красного цвета используется свойство **red**. Если написать данное выражение в playground-проекте, на панели результатов вы увидите строку

UIExtendedSRGBColorSpace указывает на то, что для определения цвета используется RGB схема. 4 цифры обозначают конкретный цвет: первые 3 определяют интенсивность красного зеленого и синего каналов, а 4-я задает прозрачность. Все довольно просто и понятно.

Но у этого удобства есть и обратная сторона: то, насколько удобен **UIKit** для разработчика, настолько же неудобен для аппаратной части устройства. Прямое взаимодействие между **UIKit** и аппаратной составляющей было бы слишком затратным и могло стать причиной снижения производительности приложений.

Чтобы лучше понять это, рассмотрим один пример. Все свои программы вы пишете на языках высокого уровня, хотя аппаратуре удобно работать с кодом ассемблера. И в этой аналогии **UIKit** – это язык высокого уровня, удобный разработчику, а ассемблер – язык, удобный оборудованию.

Для того, чтобы все возможности **UIKit** были удобны и понятны «железу», значения, которыми он оперирует, специальным образом преобразовываются и конвертируются. В результате чего ваш iPhone способен выдавать честные 60 или 120 кадров/с.

Для решения проблемы преобразования функций **UIKit** в функции аппаратной составляющей в системе iOS существует специальная прослойка, представленная в виде фреймворков **Core Animation** и **Core Graphics** (рис. 16.1).

И **Core Animation (CA)**, и **Core Graphics (CG)** находятся ближе к «железу», чем **UIKit**. Поэтому для реализации одной и той же функциональности на **UIKit** и на **CA** и **CG**, вторые потребуют написать больше программного кода. Но при этом, если вам понадобятся какие-то нестандартные возможности, данные фреймворки позволят вам их реализовать куда шире, нежели **UIKit**.

К примеру, посмотрите, что представляет из себя красный цвет, определенный значением типа **CGColor** (входит в состав фреймворка **Core Graphics**):

```
CGColor.init(red: 1, green: 0, blue: 0, alpha: 1) // <CGColor 0x600001be9920>
[<CGColorSpace 0x600001be25e0> (kCGColorSpaceICCBased; kCGColorSpaceModelRGB;
sRGB IEC61966-2.1; extended range)] ( 1 0 0 1 )
```

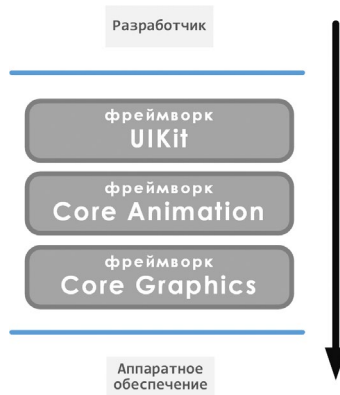


Рис. 16.1. Фреймворки iOS

В комментариях в выражении выше написана строка, выводимая в области результатов в Playground. Но что это вообще такое? Как понять эту строку? Дело в том, что такой формат данных более понятен оборудованию.

Фреймворк **UIKit** – это очень удобное для разработчика средство, но вся функциональность, реализованная при помощи его средств (разговор идет о работе с графическим интерфейсом), автоматически преобразуется с помощью **CA** и **CG** в данные, удобные для обработки аппаратной частью.

Огромный плюс состоит в том, что в своих проектах вы можете комбинировать возможности всех трех фреймворков, и когда вам будет не хватать **UIKit** (он наиболее ограничен и одновременно наиболее дружелюбен для разработчика), вы будете переходить на уровень ниже и использовать **Core Animation** и **Core Graphics**. При этом Swift позволяет вам с легкостью преобразовывать данные одних типов в данные других типов. Например, красный цвет, объявленный с помощью **UIColor**, преобразуется в значение типа **CGColor** с помощью простого свойства **cgroundColor**:

```
UIColor.red // данные типа UIColor
UIColor.red.cgColor // данные преобразованы к типу CGColor
```

В процессе изучения материала вы познакомитесь со многими типами данных, входящими как в **UIKit**, так и в **Core Animation** и **Core Graphics**. При этом об-

ратите внимание, если название типа начинается с **UI** – он относится к **UIKit**, если с **CA** – к **Core Animation**, если с **CG** – то к **Core Graphics**. Вот несколько примеров:

UIKit: **UIColor**, **UILabel**.

Core Animation: **CALayer**, **CATransition**.

Core Graphics: **CGRect**, **CGAffineTransform**.

16.2 Точки и пиксели

Мы уже неоднократно сталкивались с вами с вопросами позиционирования элементов в интерфейсе приложения. В первой части я отмечал, что для определения координат используются точки, а не пиксели. В этом разделе мы подробно разберем, с чем это связано, и что же такое на самом деле точки.

Мобильные устройства постоянно эволюционируют, в том числе улучшаются и характеристики их экранов: увеличивается плотность пикселей, повышаются их разрешение и размер диагонали. И это может создать значительные трудности для разработчиков. Только представьте: вы создали приложение, которое должно одинаково выглядеть на любом iPhone. Но дисплей каждого из устройств имеет свои характеристики. Это приводит к тому, что каждый графический элемент, например, кнопка, должен быть отдельно спозиционирован для каждого из устройств:

- на iPhone 3GS (имеет экран с низкой плотностью пикселей) кнопка должна располагаться в 20 пикселях от верхнего края и быть 50 пикселей в ширину;
- на iPhone SE2 (имеет экран Retina, с высокой плотностью пикселей) кнопка должна располагаться в 40 пикселях от верхнего края и быть 100 пикселей в ширину (так как плотность пикселей увеличилась);
- на iPhone 8 Plus кнопка должна располагаться в 60 пикселях от края и быть 150 пикселей в ширину.

Такой подход создал бы значительные трудности для разработчиков. Вместо того, чтобы тратить свое драгоценное время на создание качественного программного кода, нам бы пришлось прилагать значительные усилия на оптимизацию внешнего вида. Но компания Apple не была бы Apple, если бы не предложила механизмы, позволяющие создавать унифицированный графический интерфейс, с одним из которых (ограничения) вы уже начали свое знакомство. В основе этих механизмов лежит понятие точки. Все координаты и расстояния между графическими элементами в iOS (и, соответственно, iPadOS) измеряются в точках.

Точка (или **поинт**, от англ. **point**) – это абстрактная, унифицированная для всех устройств единица измерения расстояния, используемая при размещении графических элементов. Вся хитрость точек состоит в том, что они могут быть переведены в пиксели в соответствии с определенным коэффициентом, значение которого зависит от характеристик экрана текущего устройства.

С помощью точек мы можем указать единую для всех устройств позицию графического элемента. И если мы скажем, что кнопка должна иметь отступ в 20 точек слева и справа от края экрана, то на каком бы устройстве эта кнопка ни отображалась, она всегда будет выглядеть практически одинаково: на одних устройствах отступ будет составлять 20 пикселей, на других – 40, а на третьих – 60, в зависимости от характеристик дисплея.

Примечание В настоящее время соотношение в 1 пиксель на 1 точку уже не используется, так как экраны с низкой плотностью пикселей больше не доступны на рынке. Одной точке всегда соответствует 4 (квадрат 2x2) или 9 (квадрат 3x3) пикселей.

Таблица 16.1 содержит характеристики нескольких устройств с указанием плотности пикселей, разрешения в пикселях, разрешения в точках и коэффициента перевода пикселей в точки. Коэффициент определяет соотношение разрешения в точках и пикселях.

Таблица 16.1. Характеристики экранов

Модель	Разрешение в точках	Коэффициент	Разрешение в пикселях
iPhone 12	390 x 844	3x	1170x2532
iPhone XR	414 x 896	2x	828 x 1792
iPhone XS Max	414 x 896	3x	1242 x 2688
iPhone 8	375 x 667	2x	750 x 1334

Примечание Полную таблицу соответствия можно посмотреть по ссылке: <https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/adaptivity-and-layout/>

Позиционируя элементы на сцене (с помощью **Interface Builder** или средств программного кода), вы будете использовать точки, но не пиксели. Преобразование точек в конкретное количество пикселей на конкретном устройстве – это задача операционной системы. Таким образом, вы сможете создавать действительно универсальные графические интерфейсы.

Примечание Стоит отметить, что в некоторых случаях вы все же сможете работать с пикселями, к примеру, при использовании технологий вроде OpenGL, или при обработке изображений. Но при позиционировании изображения на сцене вы вновь будете возвращаться к точкам.

16.3 Позиционирование представлений и системы координат

Теперь поговорим о том, как работает класс **UIView**, и обсудим вопрос позиционирования графических элементов.

Класс **UIView** – это основа любого графического элемента. Он обладает следующими особенностями:

- определяет прямоугольную область с собственной системой координат;
- может иметь вложенные в себя графические элементы (так создается уже известная нам иерархия представлений);
- может реагировать на события касания;
- позволяет рисовать внутри себя контент средствами **Core Animation** и **Core Graphics**;
- к нему могут быть применены возможности Auto Layout (например, ограничения) для адаптации размеров и позиции элемента.

С помощью экземпляра класса **UIView** на экране устройства определяется прямоугольная область (ее размеры задаются в точках). Она может сама обеспечивать отрисовку графических элементов, или включать в себя другие прямоугольные области (другие экземпляры **UIView**). Как вы знаете, вложенные друг в друга представления создают иерархию. При этом встает вопрос, как в таком случае позиционируются элементы, чтобы они корректно отображались на экране на требуемых местах. Мы уже касались этого вопроса в первой части, когда говорили о том, что такое frame. Но разберем данный вопрос более подробно.

При позиционировании графических элементов используется система координат родительского представления (супервью). При этом каждое представление создает и свою собственную систему координат, в которой позиционируются дочерние по отношению к нему представления.

Таким образом, для каждого вью характерны две системы координат:

1. система координат родительского представления (супервью), в которой позиционируется данный элемент;
2. собственная система координат представления, в которой будут позиционироваться вложенные элементы.

Система координат в iOS немного отличается от той, к которой мы привыкли из курса алгебры (рис. 16.2):

1. ось X – горизонтальная ось, значения по которой растут слева-направо;
2. ось Y – вертикальная ось, значения по которой растут сверху вниз.

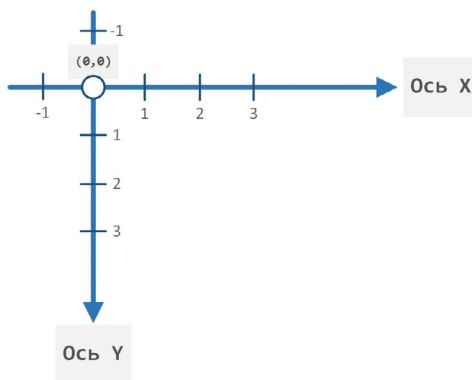


Рис. 16.2. Система координат в iOS

Начало координат обычно находится в левом-верхнем углу представления. Таким образом получается, что если рассматривать корневое представление (root View) всего приложения в целом (это экземпляр **UIWindow**), то оно занимает всю площадь экрана и имеет начало координат, совпадающее с левым-верхним углом экрана устройства (рис. 16.3).

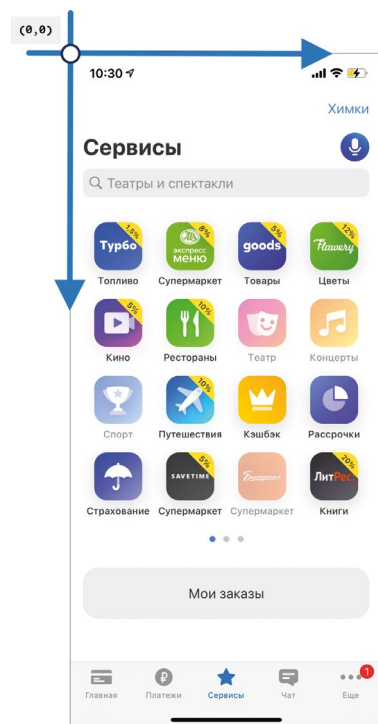


Рис. 16.3. Система координат приложения

Рассмотрим пример на основе следующей иерархии представлений (рис. 16.4):

UIWindow – корневой элемент

- **rootView** – вложен в **UIWindow**

- - **firstView** – вложен в **rootView**

- - - **secondView** – вложен в **firstView**

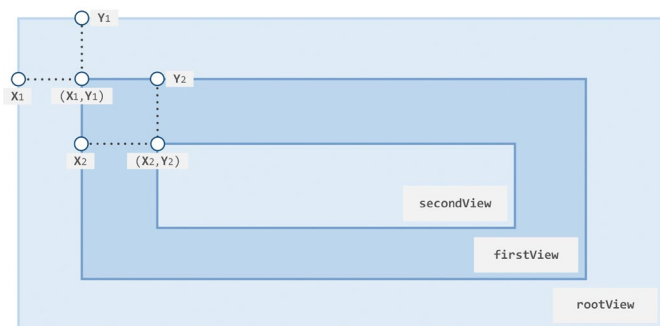


Рис. 16.4. Иерархия графических элементов

Представление **rootView** имеет координаты $(0,0)$ в системе координат **UIWindow** и занимает всю доступную ему площадь, то есть его размеры соответствуют размерам **UIWindow**.

Представление **firstView** позиционируется уже в системе координат **rootView** и имеет координаты $(X1, Y1)$.

Представление **secondView** позиционируется в системе координат **firstView** и имеет координаты $(X2, Y2)$.

Примечание Хочется отметить, что подобный интерфейс можно создать и с помощью другой иерархии, в частности, когда **firstView** и **secondView** являются дочерними к **rootView**. В этом случае представление **secondView** имело бы координаты $(X1 + X2; Y1 + Y2)$.

Каждый экземпляр класса **UIView позиционируется в системе координат родительского представления (супервью) и определяет собственную систему координат для позиционирования дочерних элементов.**

Каждый раз, когда вы делаете какое-либо действие, связанное с размещением графического элемента, вы должны четко понимать, в какой именно системе координат производится работа.

Размеры графического элемента в родительской системе координат называются **frame**, а в собственной системе координат – **bounds**. Сейчас это довольно сложно понять, но это два важнейших механизма. К концу главы вы полностью осознаете их суть и научитесь применять их на практике.

16.4 Создание кастомных представлений

Создание проекта playground в составе Xcode-проекта

Приступим к практической части изучения возможностей класса **UIView**.

- ▶ В Xcode создайте новый проект с именем «**Cards**» (рис. 16.5).

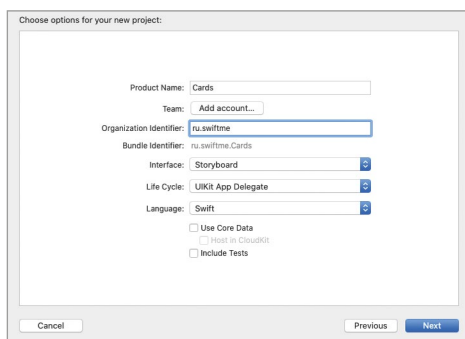


Рис. 16.5. Создание проекта «Cards»

В первой книге мы активно использовали среду Xcode Playground, но для работы в ней необязательно создавать отдельный playground-проект. Вы можете добавить песочницу непосредственно в Xcode-проект в качестве составного элемента.

- ▶ В составе проекта создайте новый файл с именем **ViewExperiments.playground**, при этом в качестве типа файла выберите **Single View Playground** (он расположен внизу списка, рис. 16.6).

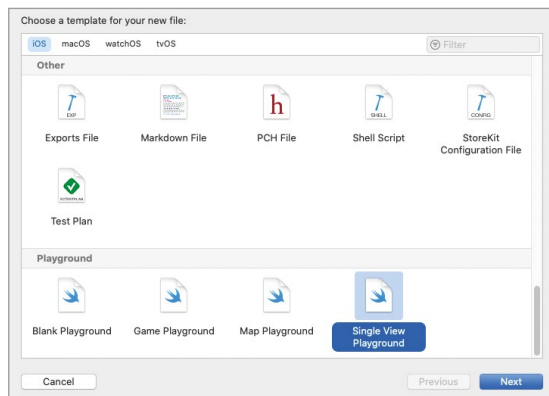


Рис. 16.6. Создание playground-файла

Теперь в составе проекта появился новый playground-файл (рис. 16.7), с его помощью мы познакомимся с классом **UIView** и попробуем на практике его возможности.

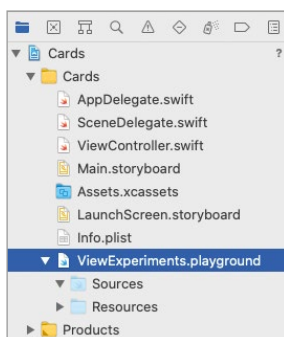


Рис. 16.7. Playground-файл в составе проекта

► Откройте файл **ViewExperiments.playground**.

Созданный файл уже содержит класс **MyViewController**, сцена которого будет отображена на панели **Live View** после запуска исполнения кода.

Примечание Помните, что сейчас код необходимо запускать в песочнице, а не собирать весь проект целиком. Используйте кнопку запуска, расположенную в нижней части редактора.

На сцене с помощью программного кода уже создана текстовая метка (рис. 16.8).

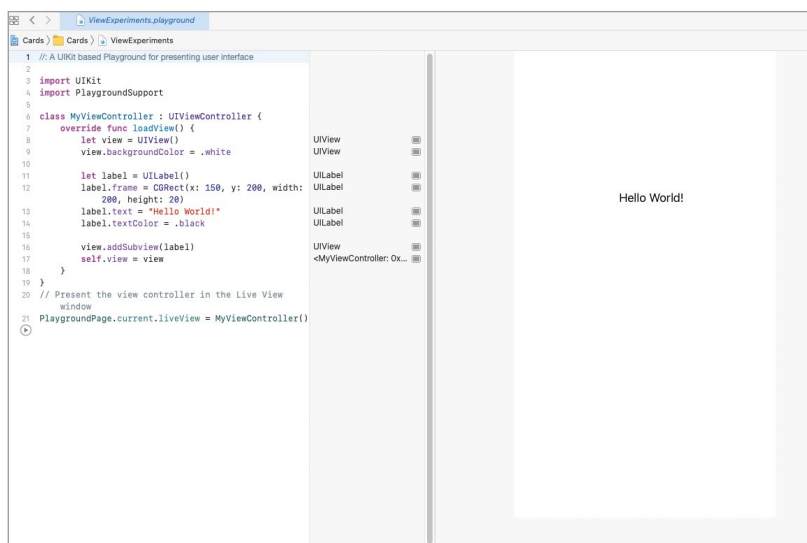


Рис. 16.8. Live View playground-файла

Любой View Controller имеет корневое представление, доступ к которому можно получить с помощью свойства **view**. Ранее данное представление мы называли корневым представлением сцены – на нем размещаются все графические элементы в составе сцены.

В момент вызова метода **loadView()** корневое представление еще не инициализировано (**self.view** соответствует **nil**), и для того, чтобы создать его, мы можем использовать один из следующих способов.

1. Вызвать родительский метод **super.loadView()**.

В результате этого произойдет вызов метода **loadView** родительского класса **UIViewController**, в котором свойству **view** будет проинициализировано значение.

Примечание Если вы забыли предназначение метода **loadView**, вернитесь к рассмотрению жизненного цикла вью контроллера.

2. Создать экземпляр типа **UIView** самостоятельно и инициализировать его свойству **self.view**.

В текущей реализации метода **loadView** класса **MyViewController** как раз используется данный способ (листинг 16.1). Он наиболее интересен нам, так как позволит лучше изучить возможности класса **UIView**.

ЛИСТИНГ 16.1.

```
override func loadView() {
    // 1
    let view = UIView()

    // 2
    view.backgroundColor = .white

    // 3
    let label = UILabel()

    // 4
    label.frame = CGRect(x: 150, y: 200, width: 200, height: 20)

    // 5
    label.text = "Hello World!"

    // 6
    label.textColor = .black

    // 7
    view.addSubview(label)

    // 8
    self.view = view
}
```

Рассмотрим тело метода построчно.

1. Создается значение типа **UIView** (графическое представление).

```
let view = UIView()
```

Класс **UIView** имеет несколько инициализаторов, но в данном случае используется инициализатор без параметров. Позже созданное представление будет выступать в качестве корневого для сцены, т.е. значение параметра **view** будет проинициализировано в свойство **view** класса **MyViewController**.

2. Изменяется цвет фона представления.

```
view.backgroundColor = .white
```

Свойство **backgroundColor** имеет тип **UIColor** и позволяет получить или изменить фоновый цвет представления. В нашем случае в качестве фонового устанавливается белый цвет.

3. Создается значение типа **UILabel**, описывающее сущность «Текстовая метка».

```
let label = UILabel()
```

Класс **UILabel** является дочерним к **UIView**, и в нашем случае для его создания используется тот же самый инициализатор без входных параметров, что и при создании экземпляра **UIView** в пункте 1. Позже, для того, чтобы отобразить метку на сцене, нам потребуется добавить ее в иерархию представлений.

4. Для метки определяются координаты и размеры.

```
label.frame = CGRect(x: 150, y: 200, width: 200, height: 20)
```

Свойство **frame** определяет позицию и размеры представления в системе координат родителя (супервью). В данном случае метка еще не добавлена в качестве сабвью, но мы уже можем указать ее будущие размеры и координаты. Для этого используется значение типа **CGRect**, которое описывает сущность «Прямоугольник» (от англ. rectangle) и позволяет определить координаты левого верхнего угла, а также ширину и высоту.

Примечание Тип **CGRect** входит в состав фреймворка **Core Graphics**. Данный класс определяет не контейнер, как **UIView**, а просто сущность «Прямоугольник» с заданными координатами и размерами. Значение типа **CGRect** используется классом **UIView** для определения собственных размеров при создании представления.

Вместо двух предыдущих строчек:

```
let label = UILabel()
label.frame = CGRect(x: 150, y: 200, width: 200, height: 20)
```

можно было обойтись одной, используя соответствующий инициализатор класса **UILabel** (унаследованный от **UIView**), который принимает значение типа **CGRect**:

```
let label = UILabel(frame: CGRect(x: 150, y: 200, width: 200, height: 20))
```

Оба варианта приведут к одному и тому же результату.

5. Определяется текст, который будет отображен в метке.

```
label.text = "Hello World!"
```

Свойство **text** класса **UILabel** позволяет указать текст, который должен быть выведен в метке на сцене.

6. Определяется цвет текста метки.

```
label.textColor = .black
```

Свойство **textColor** имеет тип **UIColor**, с его помощью устанавливается черный цвет.

7. Текстовая метка добавляется в иерархию представлений.

```
view.addSubview(label)
```

Теперь текстовая метка – это сабвью для представления хранящегося в параметре **view**. Но несмотря на это, на данном этапе метка еще не будет отображена на экране, так как само значение **view** еще не добавлено в иерархию представлений.

8. Установка значения корневого представления сцены

```
self.view = view
```

Подготовленное представление, уже включающее в себя текстовую метку, устанавливается в качестве корневого для сцены вью контроллера **MyViewController**.

После проделанных действий сцена готова к отображению, что вы и можете видеть на панели **Live View**. Обратите внимание на расположение текстовой метки, имеющей координаты (150, 200). Она находится не в центре сцены, а в 150 точках по оси X и 200 точках по оси Y от левого-верхнего угла корневого представления.

Создание представлений

Выше вы видели пример того, как с помощью программного кода создаются и отображаются графические элементы. Сейчас мы удалим их и потренируемся в создании собственных представлений.

- ▶ Удалите метод **loadView**.
- ▶ В классе **MyViewController** реализуйте код из листинга 16.2.

ЛИСТИНГ 16.2

```
override func loadView() {
```

```
        setupViews()  
    }  
  
    // настройка представлений сцены  
    private func setupViews() {  
        // создание корневого view  
        let view = UIView()  
        view.backgroundColor = .gray  
        self.view = view  
    }  
}
```

Теперь сцена окрашена в серый цвет (рис. 16.9).

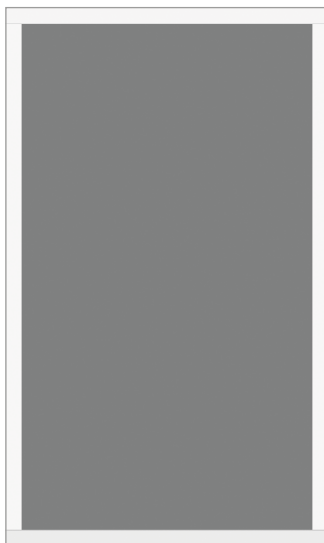


Рис. 16.9. Сцена серого цвета

Разберем, что же мы сделали, чтобы сцена стала серой.

В методе **setupViews()** создается экземпляр типа **UIView** серого цвета, который устанавливается в качестве корневого для сцены. Все довольно просто.

Обратите внимание, что данная функциональность вынесена нами в отдельный метод (**setupViews**), а не реализована прямо в **loadView**. Такой подход позволит нам разгрузить тело метода **loadView** и в дальнейшем поддерживать высокий уровень удобства работы, так как в ином случае, при написании последующего кода, метод **loadView** может увеличиться до совершенно неприличных размеров.

Метод **setupViews** будет производить всю работу по настройке представлений сцены, поэтому произведем еще одну доработку.

- Вынесите код создания корневого представления в метод **getRootView** и внесите изменения в метод **setupViews** в соответствии с листингом 16.3.

ЛИСТИНГ 16.3

```
// настройка представлений сцены
private func setupViews() {
    self.view = getRootView()
}

// создание корневого представления
private func getRootView() -> UIView {
    let view = UIView()
    view.backgroundColor = .gray
    return view
}
```

Теперь, при необходимости внесения исправлений в корневое вью, мы будем работать только с методом **getRootView**, но не с **loadView** или **setupViews**. Такой подход повышает удобство работы с кодом, так как реализует конкретные методы, которые отвечают за решение конкретной задачи:

- **loadView** выполняет операции при подготовке сцены к отображению.
- **setupViews** подготавливает представления, входящие в состав сцены.
- **getRootView** возвращает корневое представление сцены.

Также уменьшается вероятность случайного внесения ошибки.

Теперь на сцене создадим квадрат красного цвета со сторонами 200 точек и координатами (50, 50).

- Реализуйте метод **getRedView** и внесите изменения в метод **setupViews** в соответствии с листингом 16.4.

ЛИСТИНГ 16.4

```
// настройка представлений сцены
private func setupViews() {
    self.view = getRootView()
    self.view.addSubview( getRedView() )
}

// создание красного представления
private func getRedView() -> UIView {
    let viewFrame = CGRect(x: 50, y: 50, width: 200, height: 200)
    let view = UIView(frame: viewFrame)
    view.backgroundColor = .red
    return view
}
```

После запуска кода в **Live View** будет отображен красный квадрат с длиной сторон в 200 точек (рис. 16.10).

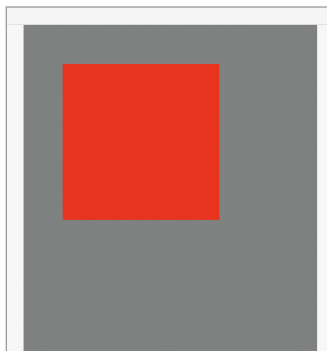


Рис. 16.10. Красный квадрат в составе сцены

Вот таким нехитрым способом создаются, группируются и отображаются графические элементы. Да, сейчас это просто серый прямоугольник и красный квадрат, но уже в скором времени вы узнаете более продвинутые способы работы с графикой.

Свойство `clipsToBounds`

Задание Добавьте на сцену квадрат зеленого цвета с длинами сторон 180 точек и координатами (100,100). Зеленый квадрат должен быть дочерним по отношению к красному квадрату. Программный код, реализующий создание, должен быть вынесен в метод **`getGreenView`**.

Мой вариант решения задания показан в листинге 16.5.

ЛИСТИНГ 16.5

```
// настройка представлений сцены
private func setupViews() {
    self.view = getRootView()
    let redView = getRedView()
    self.view.addSubview( redView )
    redView.addSubview( getGreenView())
}

// создание зеленого представления
private func getGreenView() -> UIView {
    let viewFrame = CGRect(x: 100, y: 100, width: 180, height: 180)
    let view = UIView(frame: viewFrame)
    view.backgroundColor = .green
    return view
}
```

Теперь сцена содержит три представления, включая серое корневое (рис. 16.11). Но вот, что интересно: для красного представления определены границы, но квадрат зеленого цвета все равно выходит за них! Такова стандартная логика работы представлений, но у вас есть возможность изменить ее с помощью свойства **clipsToBounds**. Если установить его в значение **true**, то все дочерние представления будут обрезаться в соответствии с границами родителя.

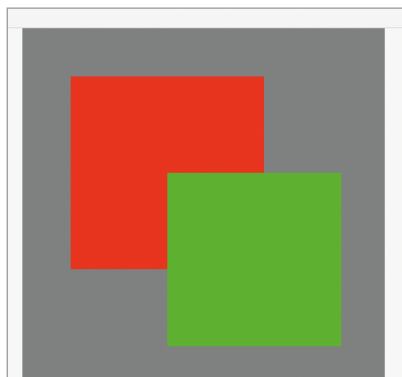


Рис. 16.11. Внешний вид сцены



Рис. 16.12. Внешний вид сцены

- Проинициализируйте значение **true** свойству **clipsToBounds** для представления, создаваемого в методе **getRedView** (листинг 16.6).

ЛИСТИНГ 16.6

```
private func getRedView() -> UIView {  
    let viewFrame = CGRect(x: 50, y: 50, width: 180, height: 180)  
    let view = UIView(frame: viewFrame)  
    view.backgroundColor = .red  
    view.clipsToBounds = true  
    return view  
}
```

Мы добились того, чего хотели – зеленое представление обрезается точно в соответствии с размерами своего родителя (рис. 16.12).

Свойство **clipsToBounds** – это очень удобный механизм, который вы будете использовать при создании кастомных представлений. Используйте его всякий раз, когда отображение дочерних элементов должно быть ограничено родителем.

Свойство origin

Задание Измените расположение зеленого квадрата таким образом, чтобы его центр был точно в центре красного квадрата (рис. 16.13).

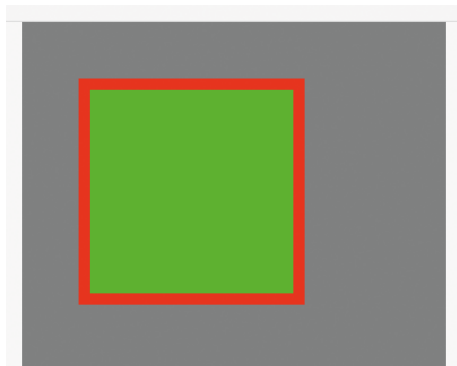


Рис. 16.13. Внешний вид сцены

Самый простой способ решения этой задачи – это рассчитать координаты левого верхнего угла зеленого представления по следующим формулам:

$$X = (\text{ширина красного вью} - \text{ширина зеленого вью}) / 2 = 10$$

$$Y = (\text{высота красного вью} - \text{высота зеленого вью}) / 2 = 10$$

Таким образом, в инициализатор **greenView** необходимо передать координаты (10, 10) (листинг 16.7).

ЛИСТИНГ 16.7.

```
private func getGreenView() -> UIView {
    let viewFrame = CGRect(x: 10, y: 10, width: 180, height: 180)
    let view = UIView(frame: viewFrame)
    view.backgroundColor = .green
    return view
}
```

Но что, если красный квадрат имеет динамические размеры, которые меняются в зависимости от устройства, на котором запущено приложение? Логично предположить, что нам потребуется рассчитывать координаты прямо в коде, используя при этом размеры красного прямоугольника. Для этого мы можем воспользоваться уже известным вам свойством **frame**, которое содержит координаты и размеры (ширину и высоту) вью в его супервью.

- Объявите метод **set(view:toCenterOfView:)** и добавьте его вызов в **setupViews** в соответствии с листингом 16.8.

ЛИСТИНГ 16.8

```
private func setupViews() {
    self.view = getRootView()
    let redView = getRedView()
```



```

let greenView = getGreenView()
set(view: greenView, toCenterOfView: redView)

self.view.addSubview( redView )
redView.addSubview( greenView )
}

private func set(view moveView: UIView, toCenterOfView baseView: UIView){
    // размеры вложенного представления
    let moveViewWidth = moveView.frame.width
    let moveViewHeight = moveView.frame.height

    // размеры родительского представления
    let baseViewWidth = baseView.frame.width
    let baseViewHeight = baseView.frame.height

    // вычисление и изменение координат
    let newXCoordinate = (baseViewWidth - moveViewWidth) / 2
    let newYCoordinate = (baseViewHeight - moveViewHeight) / 2
    moveView.frame.origin = CGPoint(x: newXCoordinate, y: newYCoordinate)
}

```

Теперь, как бы не менялись размеры **redView**, **greenView** всегда будет расположен в его центре (рис. 18.14). Более того, мы разработали прекрасный метод, с помощью которого можем с легкостью совмещать центры двух любых представлений!

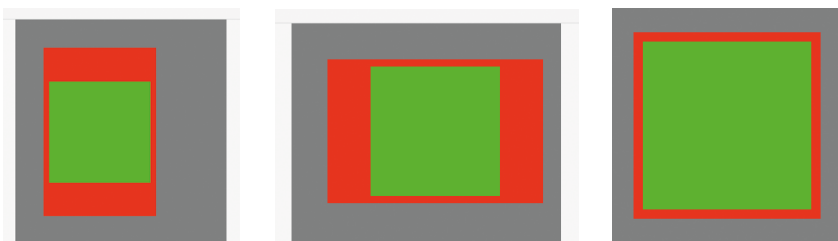


Рис. 16.14. Представление greenView в центре redView

Обратите внимание на следующие моменты.

Во-первых, при изменении координат представления **moveView** в методе **set** (последняя строка) используется свойство **frame.origin**. Данное свойство позволяет получить или изменить координаты верхнего левого угла представления. При работе с ним используется значение типа **CGPoint**, описывающее точку на плоскости (сущность «Точка»). Данный тип входит в состав класса **Core Graphics**. Таким образом, нам не требуется изменять значение свойства **frame**, например, следующим способом:

```
let moveView.frame = CGRect(x: newXCoordinate, y: newYCoordinate, width:
moveViewWidth, height: moveViewHeight)
```

Хотя и этот вариант был бы полностью рабочим. Но вместо того, чтобы лиш-
ний раз передавать длину и ширину, которые не изменяются, мы меняем лишь
координаты представления.

Во-вторых, использование свойства **frame** при получении размеров родитель-
ского представления в данном случае является ошибкой. Хотя такой вариант
и работает в нашем случае, но при определенных условиях он приведет к не-
преднамеренному изменению внешнего вида сцены. Вместо **frame** необходимо
обращаться к **bounds**, но с ним мы еще не знакомы. Поэтому оставим демон-
страцию и решение этой проблемы на потом.

Свойство center

Предположим, что нам необходимо добавить представление белого цвета в
центр красного. Как нам поступить в этом случае? Было бы логично создать
новое вью, добавить его в иерархию и обратиться к методу **set**.

- Создайте метод **getWhiteView** и дополните метод **setupViews** в соответ-
ствии с листингом 16.9.

ЛИСТИНГ 16.9

```
private func setupViews() {
    self.view = getRootView()
    let redView = getRedView()
    let greenView = getGreenView()
    // создаем представление белого цвета
    let whiteView = getWhiteView()

    set(view: greenView, toCenterOfView: redView)
    // выровняем белое представление по центру красного
    set(view: whiteView, toCenterOfView: redView)

    self.view.addSubview( redView )
    redView.addSubview( greenView )
    redView.addSubview( whiteView )
}

// создание представления белого цвета
private func getWhiteView() -> UIView {
    let viewFrame = CGRect(x: 0, y: 0, width: 50, height: 50)
    let view = UIView(frame: viewFrame)
    view.backgroundColor = .white
    return view
}
```

Этот код прекрасно работает, не зря же мы реализовывали метод `set` (рис. 16.15).

Но я бы хотел показать вам другой способ. Каждое представление имеет свойство `center`, которое определяет координаты его центра в системе координат родителя. Так как `greenView` уже расположен в центре `redView`, мы можем получить значение его свойства `center` и проинициализировать его свойству `center` представления `whiteView` вместо вызова метода `set`.

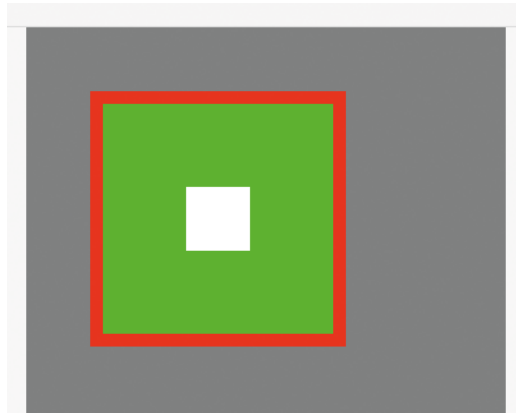


Рис. 16.15. Представление `whiteView` в центре `redView` и `greenView`

► Внесите изменения в метод `setupViews` (листинг 16.10).

ЛИСТИНГ 16.10

```
private func setupViews() {
    self.view = getRootView()
    let redView = getRedView()
    let greenView = getGreenView()
    let whiteView = getWhiteView()

    set(view: greenView, toCenterOfView: redView)
    // позиционируем белое представление с помощью свойства center
    whiteView.center = greenView.center

    self.view.addSubview( redView )
    redView.addSubview( greenView )
    redView.addSubview( whiteView )
}
```

Свойство `center`, так же, как и `origin`, имеет тип `CGPoint`. Стоит отметить, что позиционировать таким образом следует только те представления, которые имеют одного родителя, так как иначе `center` вернет совершенно неподходящее значение. В нашем случае `whiteView` и `greenView` являются дочерними по

отношению к **redView**. На рисунке 16.16 показано, как будет выглядеть сцена, если бы мы использовали выражение

```
whiteView.center = redView.center
```

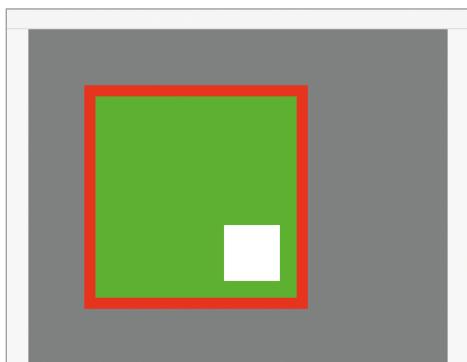


Рис. 16.16. Ошибка в совмещении центров представлений

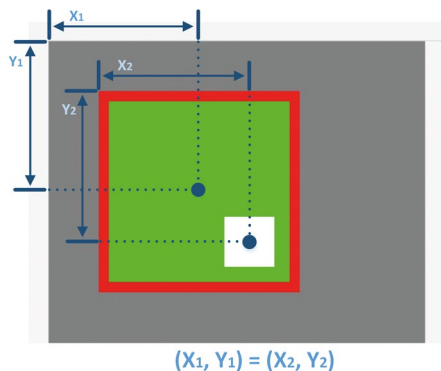


Рис. 16.17. Ошибка в совмещении центров представлений

Подумайте, почему произошло именно так?

Дело в том, что свойство **redView.center** вернуло точку, содержащую координаты центра красного представления в системе координат **rootView**. И точно такие же координаты были выставлены для центра представления **whiteView**, но уже в системе координат **redView** (рис. 16.17).

Стоит обратить внимание, что в iOS-разработке есть два принципиально разных способа позиционирования элементов: с помощью Auto Layout и с помощью фреймов. Auto Layout подразумевает использование ограничений (а также некоторых других механизмов, вроде Size Classes), а в случае с фреймами работа ведется с конкретными координатами и размерами, в том числе с помощью рассмотренных выше свойств.

Вы, как в будущем профессиональный разработчик, должны освоить верстку интерфейса каждым из указанных способов.

16.5 Класс CALayer

В вопросах создания графического интерфейса класс **UIView** безусловно важен, но его возможности довольно ограничены. По этой причине при работе над кастомными элементами интерфейса программистам часто приходится обращаться к фреймворкам **Core Animation** и **Core Graphics**, которые позволяют более гибко управлять графическим контентом.

Как говорилось ранее, даже при создании графики средствами **UIView** на самом деле все операции переадресуются на нижележащие уровни в фреймворки **Core**

Animation и **Core Graphics**. Как бы это не было удивительно, **UIView** вообще не умеет рисовать графические элементы. **UIView** – это в некотором роде контейнер, а большую часть работы по отрисовке интерфейса берет на себя **Core Animation** и его классы.

Core Animation – это не просто фреймворк, который позволяет рисовать интерфейс, это целая инфраструктура компоновки и аппаратного управления графическим содержимым приложения. В основе этой инфраструктуры лежат слои.

Понятие слоя

Одной из базовых сущностей, которая обеспечивает отображение графических элементов, является **слой** (Layer). Он представлен классом **CALayer**. Слой – это неотъемлемая часть любого представления (view). Каждое представление имеет свой собственный корневой слой (root Layer), который как раз и производит всю работу по отрисовке контента. Весь контент, который должен быть выведен в представлении, захватывается слоем и преобразуется в растровое изображение, которым можно легко управлять с помощью аппаратного обеспечения.

Доступ к слою и его возможностям можно получить при помощи свойства **layer**, например:

```
let layer = redView.layer
type(of: layer) // CALayer
```

Действия, которые непосредственно влияют на внешний вид представления, производятся средствами **Core Animation** и, в частности, классом **CALayer**. Например, при изменении цвета фона представления средствами **UIView** (с помощью свойства **backgroundColor**) на самом деле происходит смена цвета фона корневого слоя этого вью:

```
// изначальные значения фонового цвета
redView.backgroundColor // Красный цвет, значение типа UIColor
redView.layer.backgroundColor // Красный цвет, значение типа CGColor

// изменим цвет на другой с помощью возможностей UIView
redView.backgroundColor = .white
redView.backgroundColor // Белый цвет, значение типа UIColor
// значение фонового цвета слоя также изменилось
redView.layer.backgroundColor // Белый цвет, значение типа CGColor
```

Свойство **backgroundColor** класса **UIView** – это лишь посредник, который изменяет соответствующее свойство слоя, при этом автоматически преобразуя значение из типа **UIColor** в тип **CGColor**, используемый для определения цвета слоя.

Но зачем нужен **CALayer**? Почему нельзя было загрузить **UIView** данной функциональностью? Причина кроется в том же, зачем вообще нужен **Core**

Animation. Использование **UIView** – это довольно затратная операция, и создание с его помощью графических элементов расходует больше ресурсов, чем выполнение той же задачи средствами **CALayer**. Это происходит из-за того, что **Core Animation** использует более низкоуровневые, более удобные для «железа» значения, а также умеет кешировать уже отрисованные элементы для их повторного использования. Также слой не нагружен дополнительной функциональностью, вроде обработки событий; у него узкая зона ответственности, узкая специализация, а значит и выполнять свои задачи он может быстрее, чем тяжелый **UIView**.

Разберем, что такое **UIView** и **CALayer** на примере.

Представьте, что перед вами один на другом лежат два листа. В верхнем проделано отверстие, через которое видна часть нижнего листа. Так вот, **UIView** – это отверстие в верхнем листе, выступающее в качестве рамки, а **CALayer** – это нижний лист. И нам видна только та часть **CALayer**, которая попала в отверстие (рис. 16.18). И в зависимости от того, какое значение имеет свойство **clipsToBounds**, верхний лист либо прозрачный, либо нет, то есть мы либо видим то, что находится за границами отверстия, либо нет. Изменяя значение свойства **frame**, мы можем перемещать оба листа по столу, тем самым меняя их положение. Но, также мы можем перемещать только нижний лист (слой), тем самым определяя какая его часть должна быть видна. Для этого используется свойство **bounds**, о котором мы поговорим в следующем разделе.

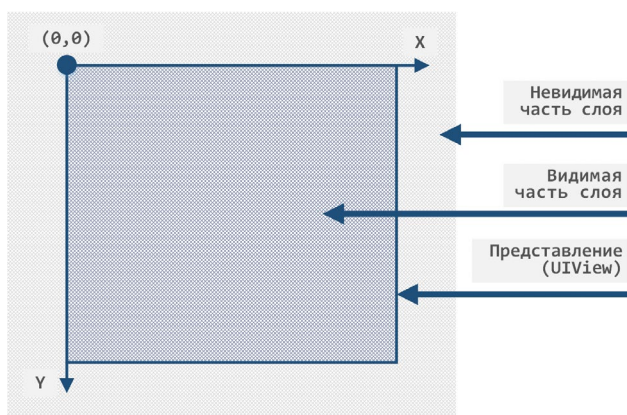


Рис. 16.18. UIView и CALayer

Возможности слоя

Слой (класс **CALayer**) предоставляет более широкие возможности по изменению внешнего вида, чем представление (класс **UIView**). Далее мы рассмотрим основные возможности **CALayer**, для которого создадим еще одно представление, и разместим его ниже на сцене.

► Добавьте на сцену представление розового цвета (листинг 16.11).

ЛИСТИНГ 16.11

```
private func setupViews() {  
  
    // ...  
  
    // добавим розовое представление на сцену  
    let pinkView = getPinkView()  
    self.view.addSubview(pinkView)  
}  
  
// создание представления розового цвета  
private func getPinkView() -> UIView {  
    let viewFrame = CGRect(x: 50, y: 300, width: 100, height: 100)  
    let view = UIView(frame: viewFrame)  
    view.backgroundColor = .systemPink  
    return view  
}
```

Теперь на сцене появилось новый элемент (рис. 16.19).

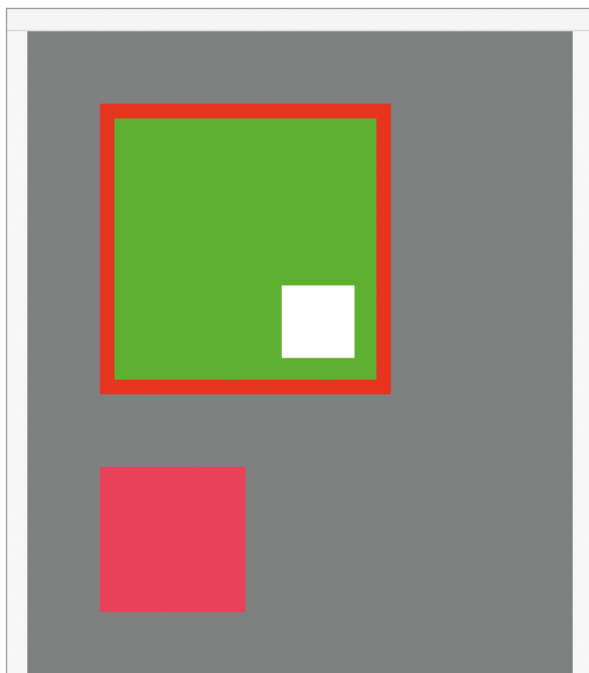


Рис. 16.19. Новый графический элемент на сцене

- Попробуйте каждый из описанных ниже механизмов, для этого потребуется вносить изменения в код метода `getPinkView`.

Толщина границ

Толщина границ регулируется с помощью свойства `borderWidth` (листинг 16.12, рис. 16.20).

ЛИСТИНГ 16.12

```
private func getPinkView() -> UIView {
    let viewFrame = CGRect(x: 50, y: 300, width: 100, height: 100)
    let view = UIView(frame: viewFrame)
    view.backgroundColor = .systemPink

    view.layer.borderWidth = 5

    return view
}
```

Примечание Напоминаю, что для доступа к корневому слою представления необходимо использовать свойство `layer`.

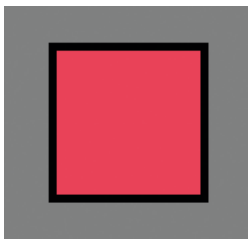


Рис. 16.20. Толщина границ

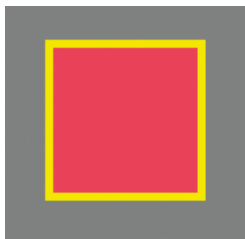


Рис. 16.21. Цвет границ

Цвет границ

Регулируется с помощью свойства `borderColor` (листинг 16.13, рис. 16.21).

ЛИСТИНГ 16.13

```
view.layer.borderColor = UIColor.yellow.cgColor
```

Обратите внимание на то, что свойство `borderColor` принимает значение типа `CGColor`.

Скругление углов

Регулируется с помощью свойства `cornerRadius` (листинг 16.14, рис. 16.22).

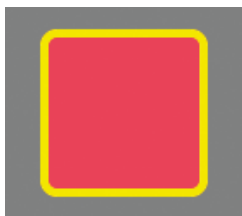


Рис. 16.22. Скругление углов слоя

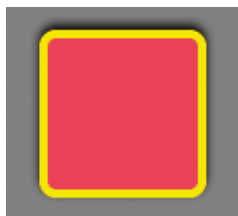


Рис. 16.23. Тень слоя

ЛИСТИНГ 16.14

```
view.layer.cornerRadius = 10
```

Увеличение значения свойства **cornerRadius** делает углы более скругленными, а уменьшение – менее скругленными. По умолчанию слой имеет прямые углы.

Видимость тени

Определяется значением свойства **shadowOpacity** (листинг 16.15, рис. 16.23).

ЛИСТИНГ 16.15

```
view.layer.shadowOpacity = 0.95
```

Значение свойства **shadowOpacity** изменяется от 0.0 (тень не видна), до 1.0 (тень полностью видна). Вы можете указать любое промежуточное значение с плавающей точкой.

Радиус размытия тени – свойство **shadowRadius** (листинг 16.16, рис. 16.24).

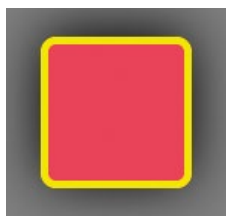


Рис. 16.24. Размытие тени

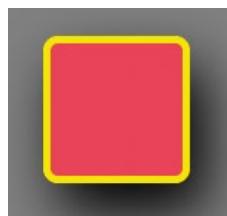


Рис. 16.25. Смещение тени

ЛИСТИНГ 16.16

```
view.layer.shadowOpacity = 1  
view.layer.shadowRadius = 20
```

Увеличение значения свойства **shadowRadius** делает тень более размытой и менее заметной. Меньший радиус делает тень более заметной, более сфокусированной, с четкими выделенными границами. Значение 0 полностью убирает размытие.

Смещение тени – свойство **shadowOffset** (листинг 16.17, рис. 16.25).

ЛИСТИНГ 16.17

```
view.layer.shadowOffset = CGSize(width: 10, height: 20)
```

Для указания размеров, на который необходимо сместить тень, используется значение типа **CGSize**.

Цвет тени

Регулируется с помощью свойства **shadowColor** (листинг 16.18, рис. 16.26).

ЛИСТИНГ 16.18

```
view.layer.shadowColor = UIColor.white.cgColor
```

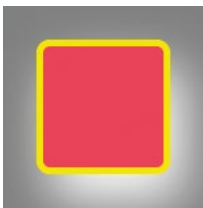


Рис. 16.26. Цвет тени

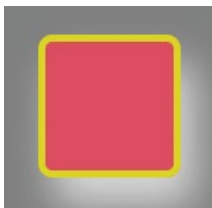


Рис. 16.27. Прозрачность
слоя

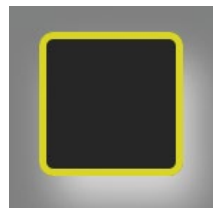


Рис. 16.28. Фоновый
цвет слоя

Прозрачность слоя

Управляется свойством **opacity** (листинг 16.19, рис. 16.27).

ЛИСТИНГ 16.19

```
view.layer.opacity = 0.7
```

Как и в случае с **shadowOpacity**, значение может изменяться от 0.0 (слой полностью прозрачен и не виден) до 1.0 (слой непрозрачен).

Цвет фона

Изменяется с помощью свойства **backgroundColor** (листинг 16.20, рис. 16.28).

ЛИСТИНГ 16.20

```
view.layer.backgroundColor = UIColor.black.cgColor
```

Класс **CALayer** предоставляет нам широкий спектр возможностей для кастомизации внешнего вида графических элементов. В дальнейшем, при разработке собственных приложений, вы будете неоднократно обращаться к ним.

- Измените фоновый цвет представления обратно на розовый.

Иерархия слоев

Вы уже знаете, что представления выстраиваются в иерархию. С помощью такого подхода вы можете отдельно настраивать внешний вид каждого вью для создания более сложных графических элементов, чем просто «цветной квадрат с закругленными углами».

Слой — это не просто элемент представления, этой полноценный самостоятельный функциональный элемент, который предназначен для отрисовки и отображения графического контента. Наравне с представлениями, слои точно также могут быть выстроены в иерархию. Да-да, в такую же иерархию, в какую выстраиваются и представления. Иерархия слоев позволяет управлять графическим контентом точно так же, как это делает иерархия представлений (рис. 16.29).

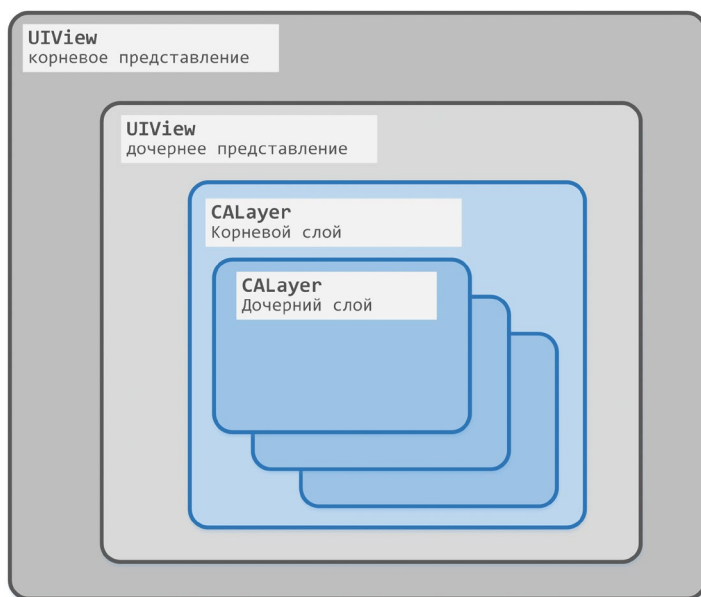


Рис. 16.29. Иерархия представлений и слоев

Таким образом, получается, что существует два механизма для создания графических элементов:

1. иерархия представлений;
2. иерархия слоев.

В ходе изучения материала и в процессе дальнейшей профессиональной деятельности вы будете обращаться к каждому из них, но вопрос в том, какой из способов использовать в каждом конкретном случае.

Рассмотрим пример.

В розовое представление (**pinkView**), которое уже отображено на сцене, требуется добавить цветной круг, расположив его в левом-верхнем углу. Каким образом можно решить эту задачу?

Вариант 1. Создать новое представление, настроить его и добавить в качестве дочернего в розовое представление.

Вариант 2. Создать новый слой, настроить его и добавить в качестве дочернего к корневому слою розового представления.

Первый вариант уже знаком нам, мы создавали представления и ранее, поэтому решим задачу вторым способом, а позже обсудим, чем они отличаются.

► Доработайте метод **getPinkView** в соответствии с листингом 16.21.

ЛИСТИНГ 16.21

```
private func getPinkView() -> UIView {
    // ...
    // код метода из предыдущих листингов
    // ...

    // создание дочернего слоя
    let layer = CALayer()
    // изменение фонового цвета
    layer.backgroundColor = UIColor.black.cgColor
    // изменение размеров и положения
    layer.frame = CGRect(x: 10, y: 10, width: 20, height: 20)
    // изменение радиуса скругления углов
    layer.cornerRadius = 10
    // добавление в иерархию слоев
    view.layer.addSublayer(layer)

    return view
}
```

В результате исполнения кода в розовом квадрате появится черный круг (рис. 16.30).

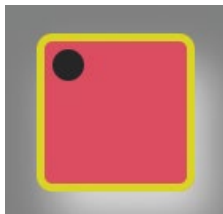


Рис. 16.30. Новый элемент на сцене

Примечание Для того, чтобы добавить слой в качестве дочернего, помимо метода **addSublayer** вы можете использовать группу методов с именем **insertSublayer**, которые имеют более гибкие возможности.

Теперь представление **pinkView** содержит не один, а два слоя, выстроенных в иерархию.

Примечание Иерархия представлений и иерархия слоев – это два различных типа иерархий. При просмотре иерархии представлений в Xcode слои не отображаются.

Теперь поговорим о том, чем использование иерархии слоев отличается от иерархии представлений?

Как говорилось ранее, класс **UIView** – это, в некотором роде, «комбайн». На нем висит довольно широкий круг обязанностей, некоторые из которых попросту не нужны при рисовании отдельных элементов. Так, например, представления могут обрабатывать касания. Реализовав задачу создания круга внутри розового представления с помощью дочернего вью, вместо слоя, у нас появилась бы возможность настроить реакцию на касание отдельно для круга и отдельно для розового квадрата. Но в некоторых случаях такая функциональность является излишней. Если нам требуется просто нарисовать графический элемент, для которого нет необходимости в отдельной обработке касаний, почему бы просто не воспользоваться возможностями **CALayer**, ведь его обработка и отрисовка для системы – значительно более простая задача в сравнении с **UIView**. Слои отрисовываются напрямую на GPU, без использования CPU. В свою очередь, представления используют в процессе отрисовки CPU.

У класса **CALayer** есть множество дочерних классов, каждый из которых предназначен для отображения определенного типа контента. К примеру, **CATextLayer** для текста, **CAShapeLayer** для фигур, **CAGradientLayer** для градиента и многие другие. С некоторыми из них мы познакомимся далее.

16.6 Свойство transform

Экземпляр класса **UIView** имеет свойство **transform**, позволяющее производить графические преобразования представлений в двухмерном пространстве: поворот, масштабирование, перемещение и наклон.

Примечание Преобразования, производимые с помощью свойство **transform** класса **UIView**, называются **аффинными**. Это понятие из математики. Я советую вам самостоятельно почитать о нем в учебниках или сети Интернет.

Свойство **transform** имеет тип **CGAffineTransform**, который входит в состав фреймворка Core Graphics. Для проведения трансформации элемента достаточно создать новое значение данного типа и проинициализировать его свойству **transform**. Рассмотрим каждую из доступных возможностей, при этом все преобразования будем производить над розовым представлением.

Поворот

СИНТАКСИС

Инициализатор `CGAffineTransform(rotationAngle:)`

Позволяет произвести поворот представления по часовой стрелке.

Аргументы

- `rotationAngle: CGFloat` – угол поворота в радианах.

При передаче значения больше 0 будет происходить поворот по часовой стрелке, при передаче значения меньше 0 – против часовой стрелки.

► Дополните метод `getPinkView` в соответствии с листингом 16.22.

ЛИСТИНГ 16.22

```
private func getPinkView() -> UIView {  
  
    // ...  
  
    // вывод на консоль размеров представления  
    print(view.frame)  
    // поворот представления  
    view.transform = CGAffineTransform(rotationAngle: .pi/4)  
    // вывод на консоль размеров представления  
    print(view.frame)  
    return view  
}
```

После запуска кода розовое представление будет повернуто на угол в $\pi/4$ радиан, что соответствует 45° (рис. 18.31).

При создании значения `CGAffineTransform` в качестве аргумента `rotationAngle` передается угол, на который необходимо повернуть изображение. **Данный угол выражен в радианах**, а не в градусах. Это может вызвать некоторые сложности. Так, например, если мы передадим в качестве значения цифру 1:

```
CGAffineTransform(rotationAngle: 1)
```



Рис. 16.31. Разворот представления

представление будет повернуто не на 1° , а на 1 радиан, что соответствует примерно $57,3^\circ$. Постоянная π соответствует углу в 180° , поэтому ее очень удобно использовать для определения градуса, на который необходимо повернуть объект (таблица 16.2).

Таблица 16.2. Соответствие радианов и градусов

Угол поворота	Что передать в качестве значения аргумента
30°	<code>.pi/6</code>
45°	<code>.pi/4</code>
90°	<code>.pi/2</code>
180°	<code>.pi</code>
270°	<code>.pi*3/2</code>

Используя данные из таблицы 16.2 и правила математики, вы сможете с легкостью получить требуемое значение. При этом положительные значения производят поворот по часовой стрелке, а отрицательные – против.

Обратите внимание на вывод на консоли, который мы добавили в предыдущем листинге (рис. 16.32). Возможно вам покажется удивительным, но значение свойства `frame` до преобразования и после него отличаются.

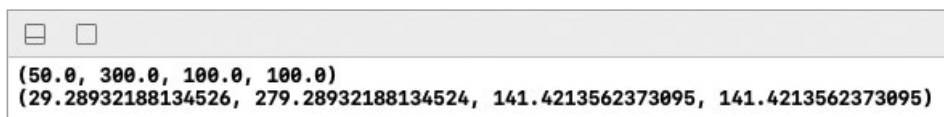


Рис. 16.32. Вывод на консоли

Причина такого поведения кроется в том, что любые преобразования (это касается не только поворотов, но и всех остальных трансформаций) происходят относительно точки привязки (*anchor point*), которой по умолчанию является центр представления. При этом само представление не поворачивается – поворачивается его корневой слой, а представление расширяется (изменяются его координаты и размеры в системе координат родителя). Взгляните на рисунки 16.33 и 16.34.

Свойство **frame** описывает представление в системе координат родителя. А так как с помощью **CGAffineTransform** изменяется не супервью, то и направление осей остается тем же самым. Но при этом внутренняя система координат представления (для работы с ним служит свойство **bounds**) также поворачивается на указанный угол (рис. 16.35), а значит все вложенные представления и слои

будут позиционироваться уже с учетом разворота (именно поэтому вместе с розовым прямоугольником повернулся и черный круг).

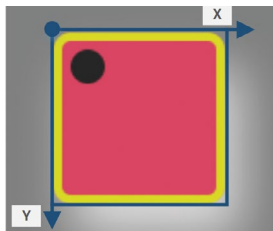


Рис. 16.33. Размеры вью до поворота

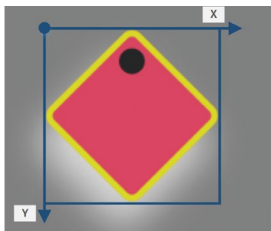


Рис. 16.34. Размеры вью после поворота

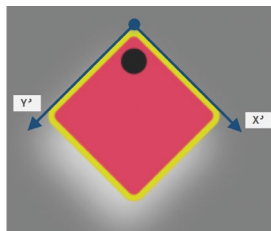


Рис. 16.35. Внутренняя система координат view после поворота

Примечание Вы можете изменить точку привязки (anchor point), относительно которой будет происходить трансформация, с помощью свойства **anchorPoint** корневого слоя, например:

```
view.layer.anchorPoint = CGPoint(x: 0, y: 0)
```

► Уберите код, производящий поворот розового представления.

Растяжение

СИНТАКСИС

Инициализатор `CGAffineTransform(scaleX:y:)`

Позволяет растянуть или сжать представление.

Аргументы

- `scaleX: CGFloat` – во сколько раз увеличить представление по оси X.
- `y: CGFloat` – во сколько раз увеличить представление по оси Y.

При передаче значений больше 1 представление будет растягиваться, при передаче значений меньше 1 – сужаться.

► Дополните метод **getPinkView** в соответствии с листингом 18.23.

ЛИСТИНГ 18.23

```
private func getPinkView() -> UIView {

    // ...

    view.transform = CGAffineTransform(scaleX: 1.5, y: 0.7)
    return view
}
```


Результат исполнения кода показан на рисунке 16.36. Розовое представление увеличило свои размеры в 1.5 раза по оси X (растянулось), и уменьшилось в 0.7 раза по оси Y (сжалось).



Рис. 16.36. Растяжение/сжатие представления

- Уберите код, производящий растяжение/сжатие розового представления.

Смещение

СИНТАКСИС

Инициализатор `CGAffineTransform(translationX:y:)`

Позволяет сместить представление на указанное количество точек.

Аргументы

- `translationX`: `CGFloat` – сдвиг по оси X.
- `y`: `CGFloat` – сдвиг по оси Y.

- Дополните метод `getPinkView` в соответствии с листингом 18.24.

ЛИСТИНГ 18.24

```
private func getPinkView() -> UIView {  
  
    // ...  
  
    view.transform = CGAffineTransform(translationX: 100, y: 5)  
    return view  
}
```

В результате исполнения кода розовое представление переместится на 100 точек вправо и на 5 точек вниз (рис. 16.37).

Множественные преобразования

При необходимости произвести несколько преобразований сразу вы можете воспользоваться цепочкой вызовов. Для этого в классе `CGAffineTransform` для каждого из рассмотренных преобразований есть специальный метод.

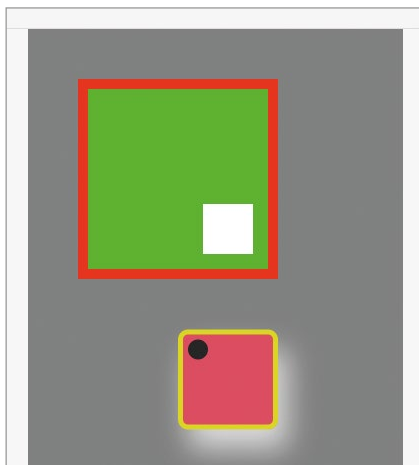


Рис. 16.37. Перемещение представления

В листинге 16.25 показан пример одновременного применения трех различных преобразований.

ЛИСТИНГ 16.25

```
view.transform = CGAffineTransform(rotationAngle: .pi/3).scaledBy(x: 2, y: 0.8).translatedBy(x: 50, y: 50)
```

Результат такого преобразования показан на рисунке 16.38.



Рис. 16.38. Множественное преобразование

Свойство identity

Если свойству **transform** передать значение **CGAffineTransform.identity**, все произведенные над представлением преобразования будут отменены. Данный подход будет использоваться вами при анимировании объектов, когда сперва элемент поворачивается/растягивается/перемещается, а потом с помощью передачи значения **identity** все произведенные изменения отменяются, и разме-

ры с позицией возвращаются к своим исходным значениям. Обратите внимание, что **identity** – это свойство типа **CGAffineTransform**:

```
view.transform = CGAffineTransform.identity
```

Метод **inverted()**

Метод **inverted()** класса **CGAffineTransform** возвращает инвертированное значение аффинного преобразования. Так, например, выражение

```
view.transform = CGAffineTransform(rotationAngle: .pi/4)
```

произведет поворот представления на 45°, а выражение

```
view.transform = CGAffineTransform(rotationAngle: .pi/4).inverted()
```

или

```
view.transform = CGAffineTransform(rotationAngle: .pi/4)
```

```
view.transform = view.transform.inverted()
```

на -45°.

- Удалите из кода все преобразования, произведенные над представлением **pinkView**.

16.7 Свойство **bounds**

Один из наиболее частых вопросов, которые задают на собеседованиях потенциальным сотрудникам: «Чем отличается **frame** от **bounds**?».

Вот ответ на заданный выше вопрос:

«Свойство **frame** определяет, где находится представление в системе координат супервью. Свойство **bounds** определяет внутренние размеры и положение представления относительно собственной системы координат».

Как вам кажется, прост ли приведенный ответ? Не совсем, и чтобы хорошо разобраться в этом, нам пришлось рассмотреть весь предыдущий материал. С понятием **frame** обычно не возникает проблем в понимании, а вот с **bounds** порой приходится повозиться.

Вспомним пример, который мы рассматривали при изучении класса **CALayer**. Представление и слой – это два листа, расположенных друг над другом. В верхнем листе проделано отверстие, и он символизирует представление, нижний лист – это слой.

Благодаря внутренней системе координат представления мы можем изменять координаты верхнего левого угла вью, тем самым словно перемещать нижний

лист, не трогая при этом верхний. Это позволит показать именно ту область внутри вью, которая необходима. Для работы с внутренней системой координат представления используется свойство **bounds**.

Свойство **bounds** имеет тип **CGRect** и устанавливается автоматически вместе с определением значения свойства **frame**. Но **frame** определяет положение и размеры в супервью, а **bounds** – в собственной системе координат.

Предположим, у нас есть представление, в котором отображается дочерний слой с небольшим отступом по краям (рис. 16.39).

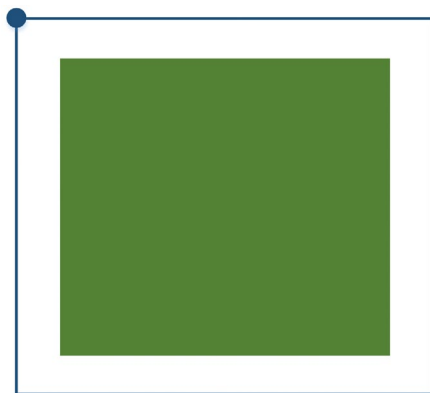


Рис. 16.39. Вложенные представления

Если обратиться к свойствам **frame** и **bounds** старшего представления, мы увидим примерно следующие значения:

```
print(view.frame) // (50.0, 50.0, 100.0, 100.0)
print(view.bounds) // (0.0, 0.0, 100.0, 100.0)
```

То есть, при тех же самых размерах (равная ширина и высота) отличаются координаты.

Координаты **frame** определяют позицию левого верхнего угла представления в системе координат родителя, а координаты **bounds** – позицию левого верхнего угла в собственной системе координат. Звучит довольно сложно, и чтобы лучше разобраться, посмотрите на рисунок 16.40.

При изменении **bounds** мы словно сдвигаем дочерний слой влево-вверх, чтобы верхний-левый угол представления оказался в координатах **(25, 25)** внутренней системы координат представления. При этом значение **frame** не изменяется ни у самого представления, ни у дочерних представлений (если они имеются).

В частности, **bounds** используется в том случае, когда необходимо отобразить конкретную часть изображения. Изменяя **bounds**, мы можем перемещать и отображать необходимый контент (рис. 16.41).

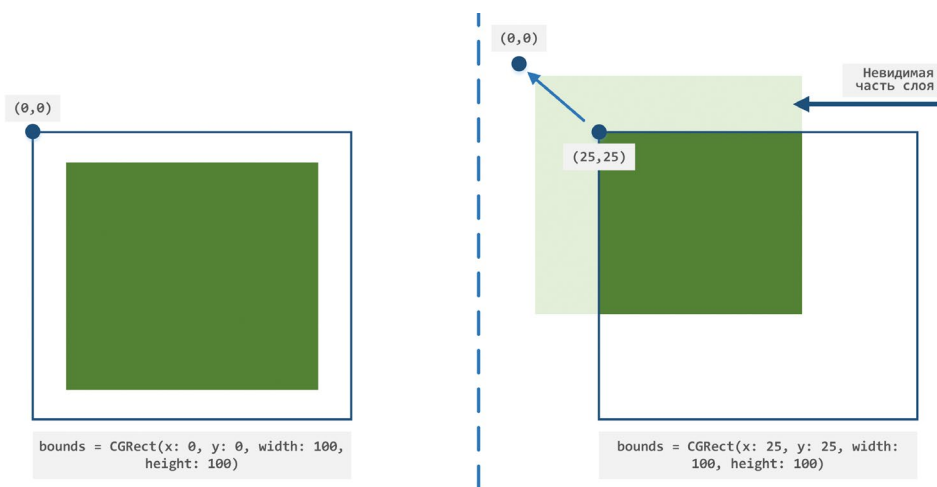


Рис. 16.40. Изменение bounds



Рис. 16.41. Изменение bounds

Также **bounds** спасает в том случае, когда над представлением произведены трансформации с помощью свойства **transform**. Вернитесь к рисунку 16.35 и вспомните то, о чем я говорил: внутренняя система координат преобразуется вслед за изменением всего представления. Таким образом, когда нам необходимо учитывать потенциальные изменения вью для размещения в нем дочерних элементов, необходимо использовать **bounds**.

В начале главы, при написании метода **set**, я обращал ваше внимание на то, что для центрирования прямоугольников лучше использовать свойство **bounds** (если не помните тот комментарий, вернитесь к листингу 16.8 и комментариям после него).

Для демонстрации этой проблемы произведем поворот красного представления и посмотрим на результат (листинг 16.26, рис. 16.42).

ЛИСТИНГ 16.26

```
private func setupViews() {
    self.view = getRootView()
    let redView = getRedView()
    let greenView = getGreenView()
    let whiteView = getWhiteView()

    // поворот красного представления
    redView.transform = CGAffineTransform(rotationAngle: .pi/3)

    set(view: greenView, toCenterOfView: redView)
    whiteView.center = greenView.center
    // ...
}
```

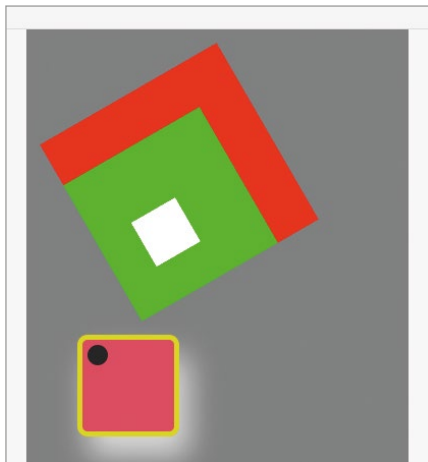


Рис. 16.42. Ошибка при центрировании представлений

Неожиданно, но сейчас красное представление выглядит совершенно не так, как мы этого ожидали. Проблема заключается в том, каким образом были получены координаты центральной точки красного представления в методе `set`. В данный момент там используется свойство `frame`:

```
// размеры базового представления
let baseViewWidth = baseView.frame.width
let baseViewHeight = baseView.frame.height
```

но при повороте представления размеры его `frame` увеличиваются, а значит координаты центральной точки также изменяются. А при их установке в свойство `center` зеленого представления мы получим тот результат, который видим на рисунке: вложенное вью смещено в сторону.

Для решения этой проблемы достаточно заменить использование **frame** на **bounds** при расчете центральной точки базового (в данном случае – красного) представления.

```
// размеры базового представления
let baseViewWidth = baseView.bounds.width
let baseViewHeight = baseView.bounds.height
```

Таким образом, мы получим размеры не всего представления в целом, а только той части, которая соответствует красному квадрату.

► Внесите изменения в метод **set** в соответствии с листингом 16.27.

ЛИСТИНГ 16.27

```
private func set(view moveView: UIView, toCenterOfView baseView: UIView){
    // размеры сдвигаемого представления
    let moveViewWidth = moveView.frame.width
    let moveViewHeight = moveView.frame.height

    // размеры базового представления
    let baseViewWidth = baseView.bounds.width
    let baseViewHeight = baseView.bounds.height
    // ...
}
```

В результате сцена будет выглядеть так, как и требовалось (рис. 16.43).

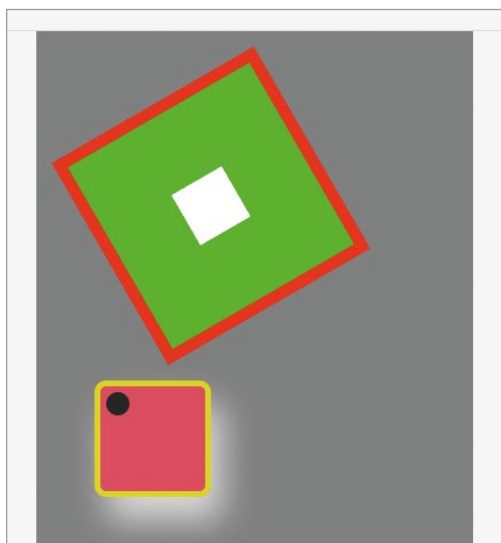


Рис. 16.43. Внешний вид сцены

Свойства minX, minY, midX, midY, maxX, maxY

Работая с классом **CGRect**, который является типом данных для **frame** и **bounds**, вам доступны свойства **minX**, **minY**, **midX**, **midY**, **maxX**, **maxY**, с помощью которых можно получить различные координаты представления. Так, например, **minX** возвращает координаты левого-верхнего угла (без учета трансформации представления) по оси X, а **minY** – по оси Y.

В таблице 16.3 приведены перечень и описание данных свойств для двух вариантов представлений (обычного и развернутого с помощью свойства **transform**) с указанием их расположения на рисунке 16.44.

Таблица 16.3. Соответствие свойств и точек в представлении

Координаты точки	Описание	Номер точки
<code>frame.minX</code> , <code>frame.minY</code>	Левый-верхний угол представления в системе координат супервью	1
<code>bounds.minX</code> , <code>bounds.minY</code>	Левый-верхний угол представления в собственной системе координат	1'
<code>frame.midX</code> , <code>frame.midY</code>	Центр представления в системе координат супервью	2
<code>bounds.midX</code> , <code>bounds.midY</code>	Центр представления в собственной системе координат	2'
<code>frame.maxX</code> , <code>frame.maxY</code>	Правый-нижний угол представления в системе координат супервью	3
<code>bounds.maxX</code> , <code>bounds.maxY</code>	Правый-нижний угол представления в собственной системе координат	2'

Используя эти свойства, мы можем значительно упростить код метода **set**.

► Внесите изменения в метод **set** в соответствии с листингом 16.28.

ЛИСТИНГ 16.28

```
private func set(view moveView: UIView, toCenterOfView baseView: UIView){
    moveView.center = CGPoint(x: baseView.bounds.midX, y: baseView.bounds.midY)
}
```

Для сведения центров двух представлений мы взяли координаты центра внутренней системы координат родительского вью (красного) и присвоили их координатам центра дочернего вью (зеленого). В результате этого метод из многострочного превратился в однострочный!

Очень трудно выделить особенно важные знания для iOS-разработки, так как все они важны. Но **UIView** стоит особняком. Его изучение займет довольно много времени, но потраченные силы вернуться в виде опыта.

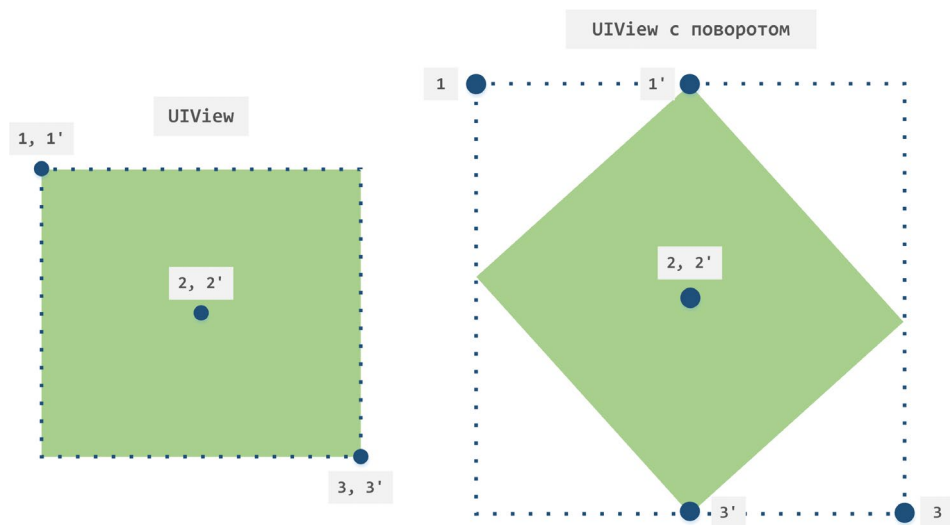


Рис. 16.44. Точки в представлении

Глава 17.

Кривые Безье

В этой главе вы:

- познакомитесь с кривыми Безье;
- рассмотрите, какие возможности для создания графических элементов они предоставляют.

Разработка графического интерфейса – это один из важнейших этапов работы над проектом. В некоторых приложениях используются исключительно стандартные элементы из состава iOS SDK, но иногда программист должен реализовать что-то такое, для чего приходится использовать фреймворки **UIKit**, **Core Animation** и **Core Graphics** на полную мощность.

Одной из интересных возможностей, которую можно применять при создании кастомных элементов интерфейса, являются кривые Безье, о которых мы поговорим в этой главе.

17.1 Что такое кривые Безье

Кривые Безье – это тип алгебраических кривых, названных в честь своего создателя Пьера Безье. Использование таких кривых доступно практически во всех графических редакторах растровой, векторной и даже трехмерной графики, не говоря уже о системах проектирования пользовательского интерфейса приложений (например, Sketch и Figma). При разработке приложений на Swift с помощью кривых Безье у разработчика есть возможность создания фигур произвольной формы.

За работу с кривыми отвечает класс **CGPath**, который входит в состав **Core Graphics**. Как и все остальные классы этого фреймворка, он позволяет достичь высокой скорости отрисовки графики, так как напрямую работает с GPU. Но, как я говорил ранее, не все, что хорошо для «железа», хорошо и для программиста. Для работы с **CGPath** требуется наличие определенных навыков.

По этой причине в большинстве случаев для работы с кривыми используется класс **UIBezierPath**, являющийся по сути надстройкой над **CGPath**, и предостав-

ляющий более дружелюбный интерфейс. Благодаря **UIBezierPath** разработчик получает действительно удобный набор инструментов для создания кривых.

Примечание Swift позволяет вам при необходимости с легкостью конвертировать значение из типа **UIBezierPath** в **CGPath**.

В данной главе мы рассмотрим возможности **UIBezierPath**.

Класс **UIBezierPath** описывает путь (англ. path) линии, соответствующей некоторой фигуре. С помощью **UIBezierPath** вы можете создавать прямые или кривые линии, прямоугольники, круги, овалы или более сложные фигуры. Несколько примеров фигур, которые вы можете реализовать, представлены на рисунке 17.1.

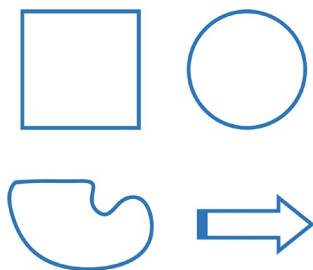


Рис. 17.1. Примеры фигур

Для создания визуальных элементов необходим **графический контекст** или, другими словами, основа, на которой будет происходить отрисовка. Графический контекст представляет собой то же самое, что и холст для художника. Вам доступны несколько вариантов создания контекста, и одним из них является класс **CAShapeLayer**, наследуемый от **CALayer**.

- ▶ В составе проекта «**Cards**» создайте новый playground-файл с именем **BezierExperiments**. В качестве типа файла выберите **Single View Playground**.
- ▶ В новом файле удалите из метода **loadView** класса **MyViewController** весь код, связанный с текстовой меткой, оставив лишь создание корневого вью белого цвета (листинг 17.1).

ЛИСТИНГ 17.1

```
class MyViewController : UIViewController {  
    override func loadView() {  
        let view = UIView()  
        view.backgroundColor = .white  
        self.view = view  
    }  
}
```

Сейчас сцена, отображаемая в Live View, представляет из себя представление белого цвета, на котором мы и будем практиковаться в создании кривых.

17.2 Создание кривых Безье

Перейдем к практике и посмотрим на возможности, которые доступны при работе с кривыми Безье.

Создание слоя

- В классе **MyViewController** объявите метод **createBezier** и добавьте его вызов в **loadView** (листинг 17.2).

ЛИСТИНГ 17.2

```
override func loadView() {
    let view = UIView()
    view.backgroundColor = .white
    self.view = view

    // создаем кривые на сцене
    createBezier(on: view)
}

private func createBezier(on view: UIView) {
    // 1
    // создаем графический контекст (слой)
    // на нем в дальнейшем будут рисоваться кривые
    let shapeLayer = CAShapeLayer()
    // 2
    // добавляем слой в качестве дочернего к корневому слою корневого пред-
    ставления
    view.layer.addSublayer(shapeLayer)
}
```

Метод **createBezier** будет использоваться для создания фигуры с помощью кривых Безье и отрисовки их в переданном в качестве входного параметра представлении. Разберем данный метод.

Шаг 1.

```
let shapeLayer = CAShapeLayer()
```

Создается значение типа **CAShapeLayer**, описывающее слой.

Тип **CAShapeLayer** позволяет работать с фигурами, как с множеством векторов. Из-за этого любая фигура получается независимой от разре-

ния экрана и в любой момент может быть отрисована в виде растровой картинки с необходимым уровнем детализации. Это позволяет одним и тем же фигурам прекрасно смотреться на любых экранах.

Шаг 2.

```
view.layer.addSublayer(shapeLayer)
```

Слой добавляется в качестве дочернего к корневому слою представления.

В качестве входного параметра методу **createBezier** передается представление, на котором будет происходить отрисовка фигуры. У этого представления есть корневой слой. Вы уже знаете, что слои могут создавать иерархию, поэтому с помощью метода **addSublayer** новый слой (он в дальнейшем будет содержать данные о фигуре) добавляется поверх корневого. Позже, когда на новый слой будет добавлена фигура, она отобразится на сцене.

В результате сцена будет содержать одно (корневое) представление, включающее в себя два слоя.

Что такое путь

Любая фигура представляет из себя множество точек, соединенных или не соединенных между собой. Если точки соединены, они образуют отрезки (кривые Безье). Другими словами, фигура – это множество кривых. Такое множество кривых называют **путем**.

Если перевернуть определение, то путь (англ. path) – это множество кривых, описывающих фигуру.

Для того, чтобы создать любую фигуру необходимо описать ее путь, то есть множество кривых, из которых она состоит. На рисунке 17.2 показан пример фигуры «Шалка повара» (или «Брокколи»), которая состоит из 10 кривых двух типов: прямая и дуга. Точки на рисунке помогают визуально отделить одну кривую от другой, они не являются частью фигуры.

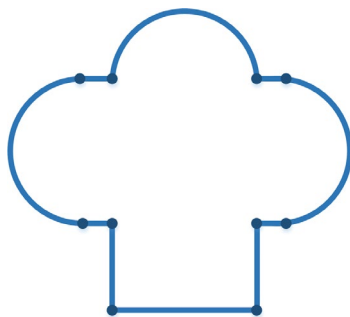


Рис. 17.2. Пример фигуры

Отображение фигуры

Для того, чтобы отобразить фигуру на сцене, необходимо выполнить следующие шаги.

1. Создать фигуру, описав путь с помощью класса **UIBezierPath**, например:

```
var path = UIBezierPath()
```

2. Передать ее контексту, на котором она будет отображена, например:

```
layer.path = path
```

В примере выше создается пустой путь, не содержащий ни одной кривой. Но используя возможности класса **UIBezierPath**, вы можете изменять значение в переменной **path** таким образом, чтобы получить требуемую фигуру.

Далее мы рассмотрим доступные в классе **UIBezierPath** методы.

Перенос указателя

Создание любой фигуры начинается с той точки, где в данный момент находится виртуальный указатель. Указатель можно сравнить с кистью, которую художник водит над полотном, но ничего при этом не рисует на нем. Кисть может перемещаться и в один момент, прикоснувшись к холсту, начать рисовать.

Для изменения позиции указателя используется метод **move**.

СИНТАКСИС

Метод **UIBezierPath.move(to:)**

Перемещает виртуальный указатель над графическим контекстом.

Аргументы

- **to:** **CGPoint** – точка, в которую необходимо переместить указатель.

Пример

```
path.move(to: CGPoint(x: 10, y: 10))
```

Прямая линия

Простейший вариант кривой Безье – это прямая линия. Для создания линии используется метод **addLine**.

СИНТАКСИС

Метод **UIBezierPath.addLine(to:)**

Рисует линию от текущей точки до переданной.

Аргументы

- **to:** **CGPoint** – точка, в которую необходимо провести линию.

Пример

```
path.addLine(to: CGPoint(x: 10, y: 10))
```

- В классе **MyViewController** создайте метод **getPath()** в соответствии с листингом 17.3

ЛИСТИНГ 17.3

```
private func getPath() -> UIBezierPath {  
    // 1  
    let path = UIBezierPath()  
    // 2  
    path.move(to: CGPoint(x: 50, y: 50))  
    // 3  
    path.addLine(to: CGPoint(x: 150, y: 50))  
    return path  
}
```

Разберем тело метода.

Шаг 1.

```
let path = UIBezierPath()
```

Создается значение типа **UIBezierPath**, которое будет описывать фигуру, а точнее ее путь.

Шаг 2.

```
path.move(to: CGPoint(x: 50, y: 50))
```

Указатель перемещается в точку с координатами **(50, 50)**.

Шаг 3.

```
path.addLine(to: CGPoint(x: 150, y: 50))
```

Рисуется линия, которая начинается в точке **(50, 50)** (текущее положение указателя) и заканчивается в точке **(150, 50)**.

Путь, описывающий прямую линию, создан. В данный момент он хранится в константе **path** и содержит одну линию, проведенную от начальной до конечной точки. Теперь созданный путь необходимо передать контексту, на котором будет отображена фигура.

- Дополните метод **createBezier** в соответствии с листингом 17.4.

ЛИСТИНГ 17.4

```
private func createBezier(on view: UIView) {  
    // 1  
    let shapeLayer = CAShapeLayer()  
    view.layer.addSublayer(shapeLayer)
```

```
// 2
// изменение цвета линий
shapeLayer.strokeColor = UIColor.gray.cgColor
// изменение толщины линий
shapeLayer.lineWidth = 5

// 3
// создание фигуры
shapeLayer.path = getPath().cgPath
}
```

Шаг 1.

```
let shapeLayer = CAShapeLayer()
view.layer.addSublayer(shapeLayer)
```

Создается графический контекст и добавляется в иерархию слоев.

Шаг 2.

```
shapeLayer.strokeColor = UIColor.gray.cgColor
shapeLayer.lineWidth = 5
```

Изменяются цвет и толщина рисуемой линии.

Свойство **strokeColor** позволяет указать цвет линий фигуры. Для этого требуется передать значение типа **CGColor**.

Свойство **lineWidth** определяет толщину линий фигуры в точках.

Указанные свойства определяются целиком для слоя, а значит будут применены ко всем фигурам на нем (если мы решим отобразить сразу несколько). Если вам необходимо получить две фигуры различного цвета или толщины, наиболее верным решением станет создание для каждой из них отдельного слоя.

Шаг 3.

```
shapeLayer.path = getPath().cgPath
```

Путь, описывающий фигуру, передается графическому контексту.

Метод **getPath** возвращает путь, который описывает фигуру. Для ее отображения на слое **shapeLayer**, возвращенное значение необходимо инициализировать свойству **path**, тип данных которого **CGPath**. Класс **UIBezierPath** имеет встроенное свойство **cgPath**, с помощью которого значение **UIBezierPath** преобразуется в значение типа **CGPath**.

После запуска playground на сцене вы увидите прямую линию серого цвета (рис. 17.3), которая начинается в точке с координатами **(50, 50)** и завершается в точке **(150, 50)**.



Рис. 17.3. Линия

Открытый путь

Дополним нашу фигуру еще одной линией.

► Дополните метод `getPath` в соответствии с листингом 17.5.

ЛИСТИНГ 17.5

```
let path = UIBezierPath()
path.move(to: CGPoint(x: 50, y: 50))
path.addLine(to: CGPoint(x: 150, y: 50))

// создание второй линии
path.addLine(to: CGPoint(x: 150, y: 150))
return path
```

То, что мы получили в итоге, немного отличается от наших ожиданий: на сцене отображается закрашенный треугольник без одной грани (рис. 17.4). Но почему так получилось?



Рис. 17.4. Фигура на сцене



Рис. 17.5. Фигура зеленого цвета на сцене

У созданной нами фигуры, состоящей из двух линий, начальная и конечная точки не соединены между собой. Путь, описывающий подобные фигуры, называется **открытым**. При этом система закрашивает внутреннее пространство фигуры таким образом, словно ее крайние точки соединены.

Свойство `fillColor`

По умолчанию внутренний фоновый цвет фигуры «непрозрачный черный». С помощью свойства `fillColor` вы можете влиять на цвет фона всех фигур на слое.

СИНТАКСИС

Свойство `CAShapeLayer.fillColor`: `CGColor`

Внутренний фоновый цвет фигуры.

Пример

```
layer.fillColor = UIColor.gray.cgColor
```

Обратите внимание на то, что свойство **fillColor** имеет тип данных **CGColor**, а значит требуется произвести дополнительное преобразование, если вы работаете с **UIColor**.

- ▶ Дополните метод **createBezier**, добавив в него инициализацию значения для свойства **fillColor** слоя (листинг 17.6).

ЛИСТИНГ 17.6

```
let shapeLayer = CAShapeLayer()
view.layer.addSublayer(shapeLayer)

shapeLayer.strokeColor = UIColor.gray.cgColor
shapeLayer.lineWidth = 5
// определение фонового цвета
shapeLayer.fillColor = UIColor.green.cgColor

shapeLayer.path = getPath().cgPath
```

Теперь фигура закрашена в зеленый цвет (рис. 17.5).

В случае, если у фигуры не должно быть фонового цвета, можно использовать один из следующих способов:

1. передать **nil** свойству **fillColor**;
`shapeLayer.fillColor = nil`
2. использовать цвет **UIColor.clear**.
`shapeLayer.fillColor = UIColor.clear.cgColor`

В обоих случаях внутренняя часть фигуры станет прозрачной (рис. 17.6).



Рис. 17.6. Фигура без фоновой заливки

Свойство `lineCap`

Свойство `lineCap` используется для оформления крайних точек линий, из которых состоит фигура.

СИНТАКСИС

Свойство `CAShapeLayer.lineCap`: `CAShapeLayerLineCap`

Стиль оформления крайних точек фигуры.

Доступные значения

- `CAShapeLayerLineCap.butt` – без применения какого-либо стиля, используется по умолчанию.
- `CAShapeLayerLineCap.square` – квадратный.
- `CAShapeLayerLineCap.round` – круглый.

Пример

```
layer.lineCap = .round
```

На рисунке 17.7 показаны соответствия значений свойства `lineCap` внешнему виду линий. Обратите внимание, что при использовании значений `.square` и `.round` на края фигуры словно надевают наконечники либо квадратной, либо круглой формы. При этом результаты `.butt` и `.square` довольно похожи.

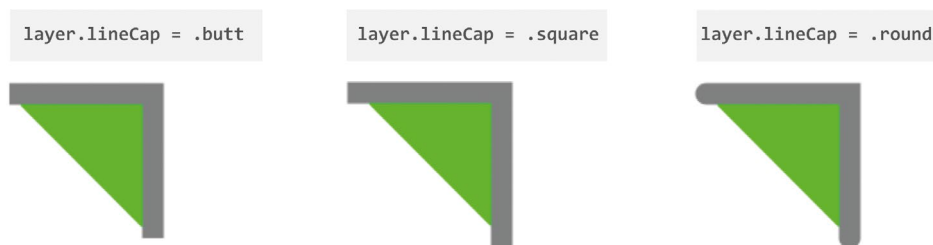


Рис. 17.7. Оформление крайних точек фигуры

Свойство `lineDashPattern`

Свойство `lineDashPattern` позволяет определить шаблон для рисования прерывистой линии.

СИНТАКСИС

Свойство `CAShapeLayer.lineDashPattern`: `[NSNumber]?`

Шаблон, определяющий порядок следования закрашенных и незакрашенных участков прерывистой линии. В качестве значения передается числовой массив или `nil`, если линия не должна быть прерывистой.

Переданный массив может содержать произвольное количество элементов, каждый из которых поочередно определяет размер закрашенного или пустого сегмента.

Пример

```
layer.lineDashPattern = [5,7,1]
```

Рассмотрим несколько примеров использования свойства **lineDashPattern** (рис. 17.8):

- **layer.lineDashPattern = [3]** – поочередно рисует закрашенные и не закрашенные сегменты размером 3 точки.
- **layer.lineDashPattern = [3, 6]** – каждый закрашенный сегмент будет иметь размер 3 точки, а не закрашенный – 6 точек.

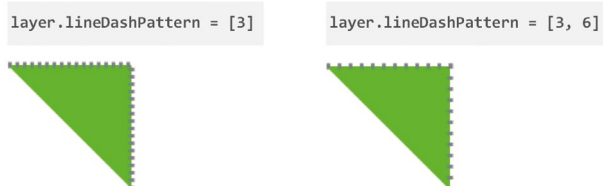


Рис. 17.8. Прерывистые линии

Совместное использование свойств **lineDashPattern** и **lineCap** позволяет оформить края каждого закрашенного сегмента (рис. 17.9).



Рис. 17.9. Прерывистые линии

Примечание С помощью свойства **lineDashPhase** можно изменить стартовое значение паттерна. Так, например код

```
shapeLayer.lineDashPattern = [5]  
shapeLayer.lineDashPhase = 2
```

сделает первый сегмент с длиной 3, а не 5.

Свойства **strokeStart** и **strokeEnd**

Свойства **strokeStart** и **strokeEnd** позволяют указать внутреннее смещение начала и конца линии.

СИНТАКСИС

Свойство `CAShapeLayer.strokeStart`: `CGFloat`

Свойство `CAShapeLayer.strokeEnd`: `CGFloat`

Начальная и конечная позиции линии. В качестве значения передается число с плавающей точкой от 0 до 1, где 0 соответствует началу фигуры, а 1 – ее концу.

По умолчанию `strokeStart` имеет значение 0, а `strokeEnd` равняется 1, то есть линия рисуется целиком.

Пример

```
layer.strokeStart = 0.3
```

На рисунке 17.10 показаны примеры различных значений свойств `strokeStart` и `strokeEnd`.

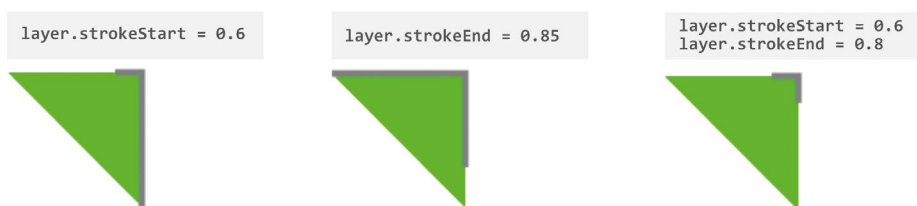


Рис. 17.10. Смещение начала и конца линии

Треугольник и свойство close

Сейчас фигура состоит из двух граней, но закрашиваясь в зеленый цвет, по сути создает треугольник. Далее мы доработаем код таким образом, чтобы у треугольника появилась третья грань.

Для решения этой задачи первое что приходит на ум – использовать метод `addLine` и дорисовать необходимую линию между концом второй линии и началом первой.

► Измените метод `getPath` в соответствии с листингом 17.7.

ЛИСТИНГ 17.7

```
private func getPath() -> UIBezierPath {
    let path = UIBezierPath()
    path.move(to: CGPoint(x: 50, y: 50))
    path.addLine(to: CGPoint(x: 150, y: 50))
    path.addLine(to: CGPoint(x: 150, y: 150))

    // создание третьей линии
    path.addLine(to: CGPoint(x: 50, y: 50))
    return path
}
```

Теперь на сцене отображается треугольник (рис. 17.11), но с ним явно что-то не так: левый-верхний угол словно отщипнули, и выглядит это довольно странно.

Причина такого поведения довольно банальная: начальная и конечная точка фигуры находятся в одних и тех же координатах (50,50), но по сути это две разные точки, в результате чего фигура не считается замкнутой. Такова логика работы с кривыми.



Рис. 17.11. Треугольник с визуальным дефектом



Рис. 17.12. Треугольник без дефектов

Для решения данной проблемы необходимо, чтобы начальная и конечная точки не просто находились в одних координатах, а являлись одной и той же точкой. Для завершения фигуры используется специальный метод `close()`, в результате использования которого фигура замыкается, дорисовывая недостающую линию и оформляя соединение. При этом нет необходимости в том, чтобы самостоятельно рисовать последнюю линию: `close` все сделает автоматически.

СИНТАКСИС

Метод `CAShapeLayer.close()`

Замыкает фигуру, дорисовывая линию от конечной точки к начальной точке.

- В методе `getPath` уберите последний вызов `addLine` и добавьте вызов `close` (листинг 17.8).

ЛИСТИНГ 17.8

```
private func getPath() -> UIBezierPath {
    let path = UIBezierPath()
    path.move(to: CGPoint(x: 50, y: 50))
    path.addLine(to: CGPoint(x: 150, y: 50))
    path.addLine(to: CGPoint(x: 150, y: 150))

    // завершение фигуры
    path.close()
    return path
}
```

Теперь у треугольника нет никаких визуальных дефектов (рис. 17.12). Обратите внимание, что нам потребовалось нарисовать две линии, а третья была создана автоматически с помощью вызова метода `close`.

Свойство `lineJoin`

С помощью свойства `lineJoin` можно определить стиль оформления соединительных точек.

СИНТАКСИС

Свойство `CAShapeLayer.lineJoin`: `CAShapeLayerLineJoin`

Стиль оформления точек, в которых соединяются отдельные линии.

Доступные значения

- `CAShapeLayerLineJoin.miter` – острый, используется по умолчанию.
- `CAShapeLayerLineJoin.bevel` – квадратный/обрезанный.
- `CAShapeLayerLineJoin.round` – скругленный.

Пример

```
layer.lineJoin = .bevel
```

На рисунке 17.13 показаны все доступные варианты использования свойства `lineJoin`.

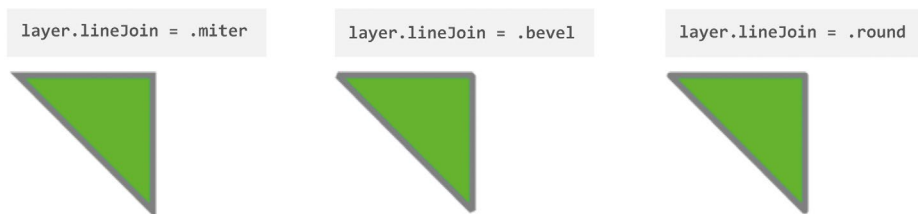


Рис. 17.13. Стиль соединительных точек линий

Многосегментные пути

При создании фигур с помощью кривых вы можете использовать многосегментные пути, то есть такие пути, которые состоят из нескольких отдельных путей.

В любой момент вы можете вновь воспользоваться методом `move` для того, чтобы перенести текущий указатель в точку с другими координатами, после чего вновь начать рисовать новый путь.

- Добавьте в метод `getPath` код для создания второго треугольника в рамках одной фигуры (листинг 17.9).

ЛИСТИНГ 17.9

```
let path = UIBezierPath()

// создание первого треугольника
path.move(to: CGPoint(x: 50, y: 50))
```

```
path.addLine(to: CGPoint(x: 150, y: 50))
path.addLine(to: CGPoint(x: 150, y: 150))
path.close()
```

```
// создание второго треугольника
path.move(to: CGPoint(x: 50, y: 70))
path.addLine(to: CGPoint(x: 150, y: 170))
path.addLine(to: CGPoint(x: 50, y: 170))
path.close()
```

```
return path
```

В результате на сцене отобразятся два несоединенных между собой треугольника (рис. 17.14), но при этом они будут представлять из себя единую фигуру, описанную в рамках одного пути. Обратите внимание, что все свойства, применяемые к графическому контексту (слою, на котором размещены фигуры), будут использоваться для оформления обоих треугольников.



Рис. 17.14. Фигура из двух треугольников



Рис. 17.15. Прямоугольник

Количество сегментов (вложенных путей) не ограничено; вы можете создавать их столько, сколько потребуется.

Прямоугольник

Теперь рассмотрим случай, когда перед вами стоит задача нарисовать прямоугольник. Для это вы можете воспользоваться одним из следующих способов:

- создать несколько линий с помощью метода **addLine** (листинг 17.10):

ЛИСТИНГ 17.10

```
let path = UIBezierPath()
path.move(to: CGPoint(x: 10, y: 10))
path.addLine(to: CGPoint(x: 210, y: 10))
```



```
path.addLine(to: CGPoint(x: 210, y: 110))
path.addLine(to: CGPoint(x: 10, y: 110))
path.close()
```

- сразу создать прямоугольник, используя инициализатор **UIBezierPath (rect:)**:

```
// создание сущности «Прямоугольник»
let rect = CGRect(x: 10, y: 10, width: 200, height: 100)
// создание прямоугольника
let path = UIBezierPath(rect: rect)
```

В обоих случаях мы получим один и тот же прямоугольник (рис. 17.15), однако второй способ явно проще.

- Нарисуйте на слое прямоугольник (листинг 17.11). Вы можете создать для этого отдельный метод или модифицировать метод **getPath**.

ЛИСТИНГ 17.11

```
// создание сущности "Прямоугольник"
let rect = CGRect(x: 10, y: 10, width: 200, height: 100)
// создание прямоугольника
let path = UIBezierPath(rect: rect)
```

При необходимости создания прямоугольника со скругленными углами класс **UIBezierPath** предоставляет специальный инициализатор **UIBezierPath (roundedRect: cornerRadius:)**.

- Измените код создания прямоугольника в соответствии с листингом листинг 17.12.

ЛИСТИНГ 17.12

```
let rect = CGRect(x: 10, y: 10, width: 200, height: 100)
let path = UIBezierPath(roundedRect: rect, cornerRadius: 30)
```

В результате на сцене отобразится прямоугольник со скругленными углами (рис. 17.16).



Рис. 17.16. Прямоугольник со скругленными углами



Рис. 17.17. Прямоугольник с двумя скругленными углами

Аргумент **cornerRadius** позволяет указать радиус скругления, то есть радиус круга, по дуге которого будет нарисован угол прямоугольника.

Также вы можете скруглить только необходимые углы (рис. 17.17):

```
let rect = CGRect(x: 10, y: 10, width: 200, height: 100)
let path = UIBezierPath(roundedRect: rect,
                        byRoundingCorners: [.bottomRight, .topLeft],
                        cornerRadii: CGSize(width: 30, height: 0))
```

Аргумент **byRoundingCorners** принимает список углов, которые необходимо скруглить, а **cornerRadii** – радиус скругления.

Дуга

Дуга – это линия, соответствующая части окружности. Для того, чтобы нарисовать дугу, необходимо определить радиус окружности, а также начальный и конечный угол дуги.

Прежде чем переходить к вопросу создания дуги в Swift, поговорим о том, как определяются углы. В iOS многие вещи перевернуты, в сравнении с тем, что мы привыкли видеть в привычной нам алгебре. К примеру, ось Y системы координат увеличивается вниз, а не вверх. Похожая ситуация обстоит и с отсчетом углов (рис. 17.18). Точка с углом 0 находится справа на окружности и идет по часовой стрелке. Так, например, угол $\pi/2$ (90°) будет соответствовать оси Y, направленной вниз.

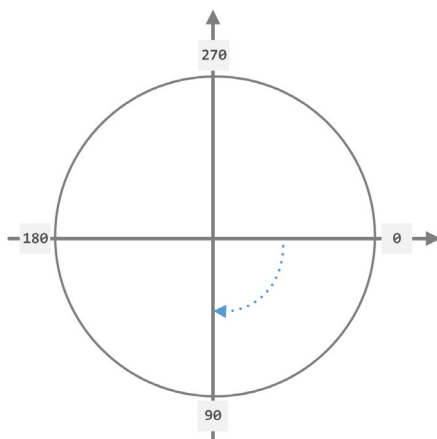


Рис. 17.18. Определение угла в iOS

Для создания дуги в Swift используется специальный инициализатор класса **UIBezierPath**.

СИНТАКСИС

Инициализатор `UIBezierPath(arcCenter: radius: startAngle: endAngle: clockwise:)`

Возвращает фигуру «Дуга».

Аргументы

- `arcCenter: CGPoint` – точка, определяющая центр окружности, по которой описывается дуга.
- `radius: CGFloat` – радиус окружности, по которой описывается дуга.
- `startAngle: CGFloat` – угол начала дуги в радианах.
- `endAngle: CGFloat` – угол конца дуги в радианах.
- `clockwise: Bool` – направление (по часовой стрелке или против).

Пример

```
let path = UIBezierPath(arcCenter: CGPoint(x: 100, y: 100),
                        radius: 50,
                        startAngle: 0,
                        endAngle: .pi,
                        clockwise: false)
```

Рассмотрим пример создания дуги.

- ▶ Нарисуйте на сцене дугу, начало которой лежит в 36° ($\pi/5$), а конец – в 180° (π) (листинг 17.13).

ЛИСТИНГ 17.13

```
let centerPoint = CGPoint(x: 200, y: 200)
let path = UIBezierPath(arcCenter: centerPoint,
                        radius: 150,
                        startAngle: .pi/5,
                        endAngle: .pi,
                        clockwise: true)
```

На рисунке 17.19 показан результат исполнения кода по рисованию дуги. Вам может показаться, что дуга имеет размер в 180° , то есть начинается в 36° , а заканчивается в 216° , но это лишь обман зрения. На рис. 17.20 я указал, где именно находятся центр окружности, по которой рисуется дуга, и углы начала и конца.

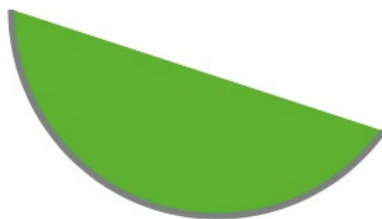


Рис. 17.19. Дуга

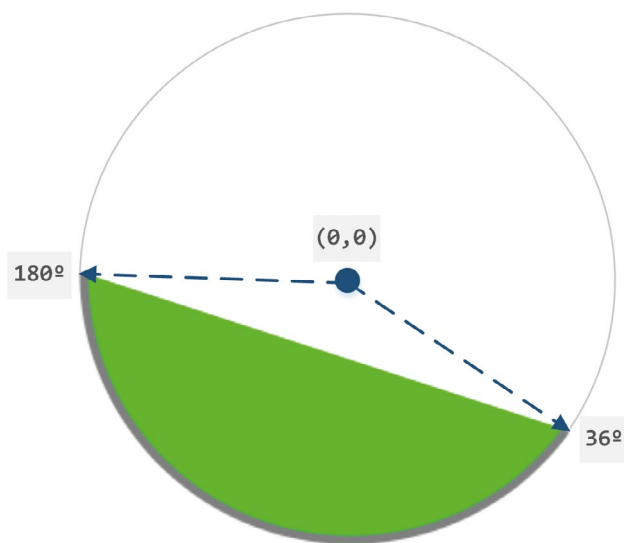


Рис. 17.20. Дуга и окружность, которой она соответствует

Круг и овал

Круг – это овал, имеющий одинаковый радиус для любого угла. Для того, чтобы нарисовать круг в Swift, можно использовать рассмотренный ранее способ создания дуги, т.е. нарисовать дугу размером 360°:

```
let centerPoint = CGPoint(x: 200, y: 200)
let path = UIBezierPath(arcCenter: centerPoint,
                        radius: 150,
                        startAngle: 0,
                        endAngle: .pi*2,
                        clockwise: true)
```

Но как и в случае с прямоугольником, в состав класса **UIBezierPath** входит специальный инициализатор, который позволяет вам с легкостью рисовать овал произвольной формы.

СИНТАКСИС

Инициализатор `UIBezierPath(ovalIn:)`

Возвращает фигуру «Овал», вписанную в заданный прямоугольник.

Аргументы

- `ovalIn: CGRect` – прямоугольник, в который необходимо вписать овал.

- Добавьте на сцену овал с радиусом 200 по горизонтали и 100 точек по вертикали (листинг 17.14).

ЛИСТИНГ 17.14

```
let rect = CGRect(x: 50, y: 50, width: 200, height: 100)
let path = UIBezierPath(ovalIn: rect)
```

Результат исполнения кода показан на рисунке 17.21.



Рис. 17.21. Овал

Если в качестве значения аргумента **ovalIn** передать квадрат, то в результате будет создан круг.

Кривые

Метод **addLine** рисует прямые линии между двумя точками, но помимо прямых вы можете рисовать кривые, определяя не только начальную и конечную точки, но и их кривизну.

СИНТАКСИС

Метод `UIBezierPath.addCurve(to: controlPoint1: controlPoint2:)`

Рисует кривую с заданной кривизной.

Аргументы

- `to: CGPoint` – точка, определяющая конец кривой;
- `controlPoint1: CGPoint` – точка, определяющая кривизну первого изгиба;
- `controlPoint2: CGPoint` – точка, определяющая кривизну второго изгиба.

► Добавьте на сцену кривую из листинга 17.15.

ЛИСТИНГ 17.15

```
let path = UIBezierPath()
path.move(to: CGPoint(x: 10, y: 10))
path.addCurve(to: CGPoint(x: 200, y: 200),
              controlPoint1: CGPoint(x: 200, y: 20),
              controlPoint2: CGPoint(x: 20, y: 200))
```

На рисунке 17.22 показан результат исполнения кода.



Рис. 17.22. Кривая

Кривизна линии задается с помощью двух точек, передаваемых в качестве аргументов метода **addCurve**. Для того, чтобы понять, как они работают, вы можете попробовать создать кривые с различными значениями этих точек. На рисунке 17.23 показан пример того, как точки влияют на размер искривления.

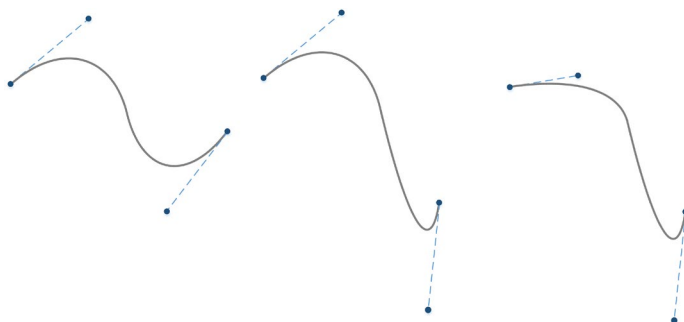


Рис. 17.23. Кривая с точками кривизны

Комбинированные пути

Для создания сложных фигур вы можете комбинировать различные способы создания пути. К примеру, для того, чтобы создать поварскую шапку, показанную в самом начале главы, можно совместно использовать методы **addLine** и **addCurve**.

► Добавьте на сцену фигуру «Поварская шапка» (листинг 17.16).

ЛИСТИНГ 17.16

```
let path = UIBezierPath()
path.move(to: CGPoint(x: 100, y: 100))
```

```
path.addArc(withCenter: CGPoint(x: 150, y: 100),
            radius: 50,
            startAngle: .pi, endAngle: 0, clockwise: true)
path.addLine(to: CGPoint(x: 220, y: 100))
path.addArc(withCenter: CGPoint(x: 220, y: 150),
            radius: 50,
            startAngle: .pi*3/2, endAngle: .pi/2, clockwise: true)
path.addLine(to: CGPoint(x: 200, y: 200))
path.addLine(to: CGPoint(x: 200, y: 260))
path.addLine(to: CGPoint(x: 100, y: 260))
path.addLine(to: CGPoint(x: 100, y: 200))
path.addLine(to: CGPoint(x: 80, y: 200))
path.addArc(withCenter: CGPoint(x: 80, y: 150),
            radius: 50,
            startAngle: .pi/2, endAngle: .pi*3/2, clockwise: true)
path.close()
```

На рисунке 17.24 показан результат исполнения кода листинга.

Вот таким нехитрым способом вы можете создать фигуру любой формы.

На этом мы завершаем свое знакомство с кривыми Безье и основными возможностями класса **UIBezierPath**. В следующих главах мы, используя рассмотренный материал, попробуем создать первые элементы будущей игры «**Cards**».



Рис. 17.24. Поварская шапка, нарисованная при помощи кривых Безье

Глава 18.

Создание игровой карточки.

Разработка кастомных представлений и слоев

В этой главе вы:

- научитесь создавать кастомные слои и кастомные представления;
- продолжите работу над приложением «Cards».

Материал главы будет посвящен созданию первого и основного элемента игры «**Cards**» – игровой карточки. В первую очередь, мы определим требования, предъявляемые к ней, которые в дальнейшем помогут нам реализовать конструктор карточек – класс **CardView**, после чего реализуем несколько вариантов их визуального оформления. Также в ходе изучения материала мы познакомимся с шаблоном проектирования «Фабрика».

18.1 Требования к игровой карточке

Для того, чтобы эффективно вести разработку сущности «Игровая карточка», необходимо разобраться с тем, какую функциональность она должна реализовывать.

Цель игры состоит в поиске пар одинаковых игровых карточек, каждая из которых будет иметь две стороны: обратную и лицевую (рис. 18.1). Лицевая сторона будет содержать одну из четырех цветных фигур (рис. 18.2). Идентичность карточек будет определяться как раз по совпадению цвета и типа изображенных на их лицевых сторонах фигур. Обратная сторона (рубашка) будет оформлена с помощью одного из двух вариантов несложных узоров (рис. 18.3). Она никак не будет влиять на игровой процесс.



Рис. 18.1. Две стороны игровой карточки

Игровое поле по умолчанию будет включать 16 карточек, каждая из которых размещается в случайных координатах (рис. 18.4). При необходимости вы всегда сможете изменить код таким образом, чтобы увеличить количество карточек на поле.

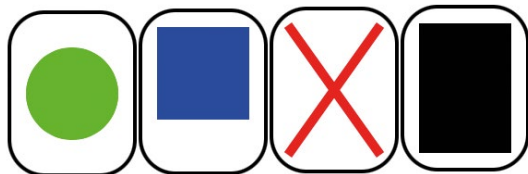


Рис. 18.2. Варианты фигур

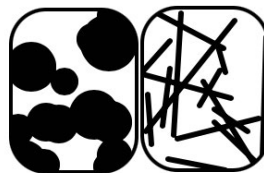


Рис. 18.3. Варианты оформления обратной стороны карточки

Изначально все карточки будут расположены рубашкой вверх с целью скрыть изображенную на них фигуру. При нажатии на любую из карточек будет происходить ее переворот. Если подряд выбраны две одинаковые карточки, то они остаются перевернутыми. В ином случае обе карточки вновь поворачиваются рубашкой вверх.

Для реализации игровых карточек мы будем использовать возможности кривых (класс **UIBezierPath**) и слоев (класс **CALayer**). Таким образом, каждой фигуре будет соответствовать свой слой, содержащий в себе информацию о пути, соответствующем данной фигуре.

Для реализации непосредственно игровой карточки, содержащей некоторую фигуру, мы разработаем кастомное представление **CardView**.

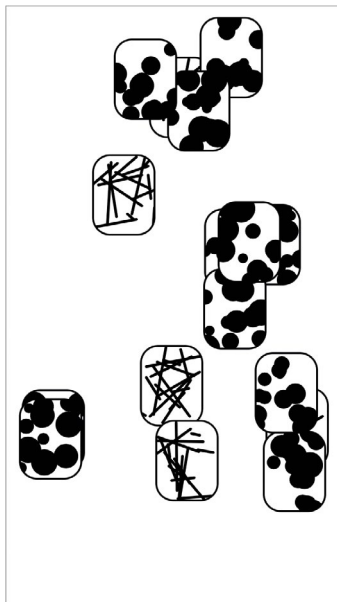


Рис. 18.4. Игровое поле

18.2 Создание кастомных слоев для фигур

Для создания фигуры средствами **UIBezierPath** в качестве графического контекста может выступать экземпляр класса **CAShapeLayer**, в котором будет содержаться вся информация о соответствующем фигуре пути. Поэтому для каждой фигуры мы разработаем отдельный кастомный класс, наследуемый от класса **CAShapeLayer**. В самом простом случае такой класс будет выглядеть следующим образом:

```
class SomeShapeLayer: CAShapeLayer {}
```

Класс **SomeShapeLayer** описывает пустую фигуру, так как в его свойстве **path** не содержится никакой информации о пути. Для того, чтобы описать путь, мы можем переопределить родительский инициализатор (или создать собственный):

```
class SomeShapeLayer: CAShapeLayer {
    // переопределяем родительский инициализатор
    override init() {
        super.init()
        // ...
    }
}
```

```

// создаем собственный инициализатор
init(someValues: SomeType) {
    // ...
    super.init()
    // ...
}

required init?(coder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
}

```

Обратите внимание, что как только определяется инициализатор, Xcode напомним о необходимости переопределения инициализатора **init?(coder:)**.

Для реализации классов будем использовать Playground. Такой подход позволит нам оперативно оценивать получившийся внешний вид созданных элементов с помощью панели Live View.

- В составе проекта создайте новый playground-файл типа **Single View Playground**.

В первую очередь определим протокол, на котором будут основаны все будущие слои.

- В новом playground-файле реализуйте протокол **ShapeLayerProtocol** (листинг 18.1).

ЛИСТИНГ 18.1

```

protocol ShapeLayerProtocol: CAShapeLayer {
    init(size: CGSize, fillColor: CGColor)
}

```

Каждая фигура, а точнее, каждый слой, должен будет вписываться в представление, описывающее игральную карточку. Для того, чтобы делать это корректно, в инициализатор фигуры будет передаваться требуемый размер. Вторым аргументом инициализатора идет цвет, в который фигура будет окрашена.

При этом необходимо позаботиться о том, чтобы разработчик (в данном случае это вы сами) случайно не создал фигуру, используя доступный по умолчанию в классе **CAShapeLayer** пустой инициализатор (не имеющий параметров).

- Добавьте в playground-файл расширение для протокола **ShapeLayerProtocol** (листинг 18.2).

ЛИСТИНГ 18.2

```

extension ShapeLayerProtocol {

```

```
init() {  
    fatalError("init() не может быть использован для создания экзем-  
пляра")  
}  
}
```

Так как все классы фигур будут подписаны на данный протокол, попытка использования пустого инициализатора приведет к фатальной ошибке. При этом в консоль будет выведено соответствующее сообщение, переданное в качестве аргумента в функцию **`fatalError`**.

На этом подготовительные работы завершены. Перейдем к реализации кастомных слоев.

Создание круга

Вы уже знаете, как нарисовать круг с помощью кривых. Сейчас эти знания потребуются обернуть в класс **`CircleShapeLayer`**.

► Добавьте в playground-файл код класса **`CircleShapeLayer`** (листинг 18.3).

ЛИСТИНГ 18.3

```
class CircleShape: CAShapeLayer, ShapeLayerProtocol {  
    required init(size: CGSize, fillColor: CGColor) {  
        super.init()  
  
        // рассчитываем данные для круга  
        // радиус равен половине меньшей из сторон  
        let radius = ([size.width, size.height].min() ?? 0) / 2  
        // центр круга равен центрам каждой из сторон  
        let centerX = size.width / 2  
        let centerY = size.height / 2  
  
        // рисуем круг  
        let path = UIBezierPath(arcCenter: CGPoint(x: centerX, y: centerY),  
                                radius: radius,  
                                startAngle: 0,  
                                endAngle: .pi*2,  
                                clockwise: true)  
  
        path.close()  
        // инициализируем созданный путь  
        self.path = path.cgPath  
        // изменяем цвет  
        self.fillColor = fillColor  
    }  
}
```

```
required init?(coder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
}
```

Процесс создания фигуры (рисования слоя) происходит в инициализаторе при создании экземпляра класса. При этом используются значения переданных аргументов:

- размеры прямоугольника, в который необходимо вписать фигуру;
- цвет фигуры.

Значение входного параметра **size** может соответствовать прямоугольнику, имеющему произвольное соотношение сторон. При этом круг должен оставаться кругом. Для решения этой задачи в коде определяется минимальное из значений параметра **size**, на основании которого и рассчитывается радиус.

- Измените метод **loadView** класса **MyViewController** для отображения созданной фигуры на сцене (листинг 18.4).

ЛИСТИНГ 18.4

```
override func loadView() {
    let view = UIView()
    view.backgroundColor = .white
    self.view = view

    // круг
    view.layer.addSublayer(CircleShape(size: CGSize(width: 200, height:
150), fillColor: UIColor.gray.cgColor))
}
```

На рисунке 18.5 показан пример отображения серого круга, вписанного в прямоугольник с размерами 200 точек по ширине и 150 по высоте.

Создание квадрата

- Добавьте в playground-файл код класса **SquareShape** (листинг 18.5).

ЛИСТИНГ 18.5

```
class SquareShape: CAShapeLayer, ShapeLayerProtocol {
    required init(size: CGSize, fillColor: CGColor) {
        super.init()

        // сторона равна меньшей из сторон
        let edgeSize = ([size.width, size.height].min() ?? 0)
```

```

// рисуем квадрат
let rect = CGRect(x: 0, y: 0, width: edgeSize, height: edgeSize)
let path = UIBezierPath(rect: rect)
path.close()
// инициализируем созданный путь
self.path = path.cgPath
// изменяем цвет
self.fillColor = fillColor
}

required init?(coder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
}

```

На рисунке 18.6 показан внешний вид квадрата на сцене, вписанного в прямоугольник со сторонами 200 на 150 точек.

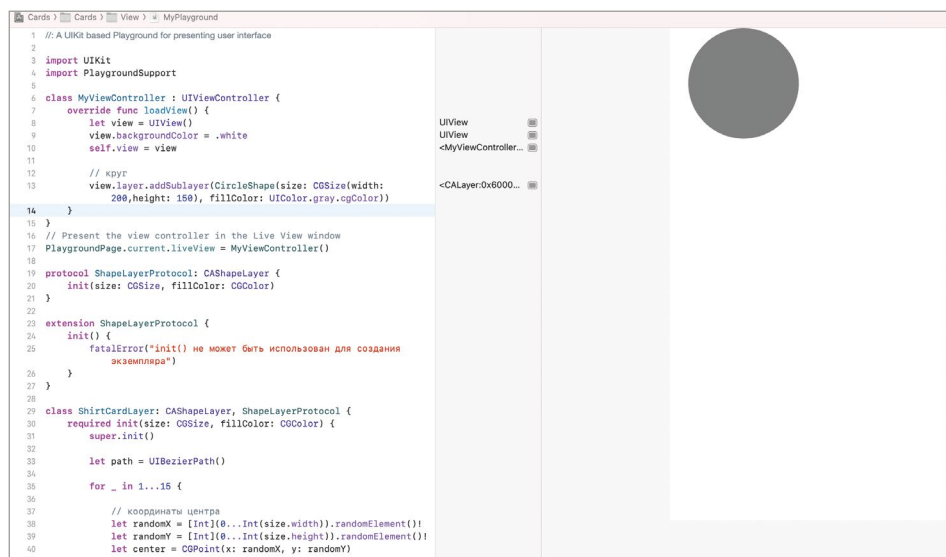


Рис. 18.5. Отображение круга на сцене



Рис. 18.6. Отображение квадрата на сцене

Создание креста

► Добавьте в playground-файл код класса **CrossShape** (листинг 18.6).

ЛИСТИНГ 18.6

```
class CrossShape: CAShapeLayer, ShapeLayerProtocol {
    required init(size: CGSize, fillColor: CGColor) {
        super.init()

        // рисуем крест
        let path = UIBezierPath()
        path.move(to: CGPoint(x: 0, y: 0))
        path.addLine(to: CGPoint(x: size.width, y: size.height))
        path.move(to: CGPoint(x: size.width, y: 0))
        path.addLine(to: CGPoint(x: 0, y: size.height))
        // инициализируем созданный путь
        self.path = path.cgPath
        // изменяем цвет
        self.strokeColor = fillColor
        self.lineWidth = 5
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

На рисунке 18.7 показан внешний вид перекрестия, вписанного в прямоугольник со сторонами 200 на 150 точек.

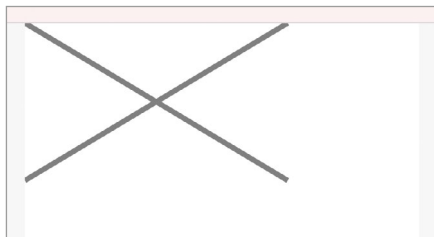


Рис. 18.7. Отображение креста на сцене

Создание закрашенного слоя

Последняя фигура, которая может быть отображена на лицевой стороне карточки – это закрашенный одноцветный прямоугольник, занимающий все пространство слоя.

► Добавьте в playground-файл код класса **FillShape** (листинг 18.7).

ЛИСТИНГ 18.7

```
class FillShape: CAShapeLayer, ShapeLayerProtocol {
    required init(size: CGSize, fillColor: CGColor) {
        super.init()
        let path = UIBezierPath(rect: CGRect(x: 0, y: 0, width: size.width,
height: size.height))
        self.path = path.cgPath
        self.fillColor = fillColor
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

На рисунке 18.8 показан внешний вид полностью закрашенного прямоугольника со сторонами 200 на 150 точек.



Рис. 18.8. Отображение закрашенного прямоугольника

Создание рубашки с кругами

Первый вариант рубашки будет содержать 15 кругов, имеющих случайные радиусы и координаты. Этакий «коровий» узор (рис. 18.9). При каждой генерации рубашки созданный узор будет получаться уникальным. Чем меньше будет размер игровой карточки, тем плотнее будут расположены круги на слое.

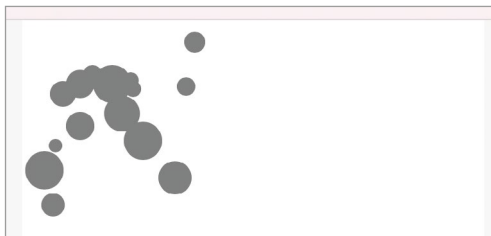


Рис. 18.9. Узор для рубашки карты

► Добавьте в playground-файл код класса **BackSideCircle** (листинг 18.7).

ЛИСТИНГ 18.7

```
class BackSideCircle: CAShapeLayer, ShapeLayerProtocol {
    required init(size: CGSize, fillColor: CGColor) {
        super.init()

        let path = UIBezierPath()

        // рисуем 15 кругов
        for _ in 1...15 {

            // координаты центра очередного круга
            let randomX = Int.random(in: 0...Int(size.width))
            let randomY = Int.random(in: 0...Int(size.height))
            let center = CGPoint(x: randomX, y: randomY)
            // смещаем указатель к центру круга
            path.move(to: center)
            // определяем случайный радиус
            let radius = Int.random(in: 5...15)
            // рисуем круг
            path.addArc(withCenter: center, radius: CGFloat(radius),
startAngle: 0, endAngle: .pi*2, clockwise: true)
        }

        // инициализируем созданный путь
        self.path = path.cgPath
        // изменяем цвет
        self.strokeColor = fillColor
        self.fillColor = fillColor
        self.lineWidth = 1
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

Создание рубашки с линиями

Второй вариант рубашки будет содержать 15 линий, имеющих случайные координаты начала и конца (рис. 18.10). При каждой генерации рубашки созданный узор вновь будет получаться уникальным.



Рис. 18.10. Узор для рубашки карты

► Добавьте в playground-файл код класса **BackSideLine** (листинг 18.8).

ЛИСТИНГ 18.8

```
class BackSideLine: CAShapeLayer, ShapeLayerProtocol {
    required init(size: CGSize, fillColor: CGColor) {
        super.init()

        let path = UIBezierPath()

        // рисуем 15 линий
        for _ in 1...15 {

            // координаты начала очередной линии
            let randomXStart = Int.random(in: 0...Int(size.width))
            let randomYStart = Int.random(in: 0...Int(size.height))
            // координаты конца очередной линии
            let randomXEnd = Int.random(in: 0...Int(size.width))

            let randomYEnd = Int.random(in: 0...Int(size.height))

            // смещаем указатель к началу линии
            path.move(to: CGPoint(x: randomXStart, y: randomYStart))

            // рисуем линию
            path.addLine(to: CGPoint(x: randomXEnd, y: randomYEnd))
        }

        // инициализируем созданный путь
        self.path = path.cgPath
        // изменяем стиль линий
        self.strokeColor = fillColor
        self.lineWidth = 3
        self.lineCap = .round
    }
}
```

```
required init?(coder: NSCoder) {  
    fatalError("init(coder:) has not been implemented")  
}  
}
```

На этом мы завершаем нашу работу со слоями и переходим к разработке кастомного представления для игровой карточки.

18.3 Создание кастомного представления для игровой карточки

В первую очередь хочу отметить, что в данный момент мы занимаемся реализацией внешнего вида, который входит в состав Представления (V в MVC). В составе Модели также будет находиться информация о такой сущности, как «Игровая карточка», так как сравнение идентичности карточек и генерация их списка для игрового поля – это часть бизнес-логики.

Для реализации игровой карточки мы можем использовать один из следующих способов:

- создать отдельный дочерний от **UIView** класс для каждого типа фигуры, например, **CircleView**, **SquareView**;
- создать единый класс, инициализатор которого будет принимать параметр, определяющий тип фигуры, например **CardView(shape: .circle, frame: ...)**;
- создать универсальный тип (дженерик), например **CardView<T>**, где тип **T** определяет фигуру, отображаемую на лицевой стороне карточки.

Первые два способа, с точки зрения возможностей языка, довольно просты в реализации, а вот третий позволит нам потренироваться в использовании универсальных типов, поэтому мы воспользуемся именно им.

► В playground-файле объявите класс **CardView** (листинг 18.9).

ЛИСТИНГ 18.9

```
class CardView<ShapeType: ShapeLayerProtocol>: UIView {}
```

Тип данных, указываемый в угловых скобках, будет использоваться внутри тела класса для создания слоя, соответствующего одной из фигур.

При создании экземпляра класса **UIView** в инициализатор передается аргумент **frame**, который определяет координаты и размеры представления. В на-

шем случае при создании значения типа **CardView** дополнительно потребуется передать цвет фигуры.

- В классе **CardView** объявите инициализатор, принимающий цвет и размеры представления, а также свойство для его хранения (листинг 18.10).

ЛИСТИНГ 18.10

```
class CardView<ShapeType: ShapeLayerProtocol>: UIView {
    // цвет фигуры
    var color: UIColor!

    init(frame: CGRect, color: UIColor) {
        super.init(frame: frame)
        self.color = color
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

Обратите внимание, что при объявлении нового инициализатора вам также необходимо объявить требуемый (**required**) инициализатор **init?(coder:)**. Xcode напомнит вам об этом и предложит сделать данную операцию автоматически.

Теперь поговорим о том, какую структуру будет иметь созданное представление.

Всего в составе вью будет два дочерних вью, одно для лицевой стороны карточки, а другое для обратной:

CardView

- **FaceSideView**
- **BackSideView**

В составе каждого из дочерних представлений, а точнее в составе их дочерних слоев, будет находиться слой, на котором изображена необходимая фигура (для **FaceSideView**) или узор для рубашки (для **BackSideView**).

В зависимости от того, перевернута карточка или нет, вверху иерархии будет находиться то или иное представление.

Функциональность переворота представления, а также его размещения одной из двух сторон к пользователю, будет реализована с помощью протокола.

- Реализуйте протокол **FlippableView** и подпишите на него класс **CardView** (листинг 18.11).

ЛИСТИНГ 18.11

```
protocol FlippableView: UIView {
    var isFlipped: Bool { get set }
    var flipCompletionHandler: ((FlippableView) -> Void)? { get set }
    func flip()
}

class CardView<ShapeType: ShapeLayerProtocol>: UIView, FlippableView {

    var isFlipped: Bool = false
    var flipCompletionHandler: ((FlippableView) -> Void)?
    func flip() {}

    // ...
}
```

Свойство **isFlipped** будет использоваться для того, чтобы определить, расположена ли игральная карточка лицевой стороной вверх или нет.

Метод **flip** в дальнейшем будет использоваться для анимированного переворота карточки.

Замыкание, хранящееся в свойстве **flipCompletionHandler**, позволит выполнить произвольный код после того, как карточка будет перевернута.

Теперь добавим в класс **CardView** код, реализующий представления для лицевой и обратной стороны карточки.

► Добавьте в класс **CardView** код из листинга 18.12.

ЛИСТИНГ 18.12

```
class CardView<ShapeType: ShapeLayerProtocol>: UIView, FlippableView {

    // ...

    // внутренний отступ представления
    private let margin: Int = 10

    // представление с лицевой стороной карты
    lazy var frontSideView: UIView = self.getFrontSideView()
    // представление с обратной стороной карты
    lazy var backSideView: UIView = self.getBackSideView()

    // возвращает представление для лицевой стороны карточки
    private func getFrontSideView() -> UIView {
        let view = UIView(frame: self.bounds)
```

```

        view.backgroundColor = .white

        let shapeView = UIView(frame: CGRect(x: margin, y: margin, width:
Int(self.bounds.width)-margin*2, height: Int(self.bounds.height)-margin*2))
        view.addSubview(shapeView)

        // создание слоя с фигурой
        let shapeLayer = ShapeType(size: shapeView.frame.size, fillColor:
color.cgColor)
        shapeView.layer.addSublayer(shapeLayer)

        return view
    }

    // возвращает вью для обратной стороны карточки
    private func getBackSideView() -> UIView {
        let view = UIView(frame: self.bounds)

        view.backgroundColor = .white

        //выбор случайного узора для рубашки
        switch ["circle", "line"].randomElement()! {
        case "circle":
            let layer = BackSideCircle(size: self.bounds.size, fillColor:
UIColor.black.cgColor)
            view.layer.addSublayer(layer)
        case "line":
            let layer = BackSideLine(size: self.bounds.size, fillColor:
UIColor.black.cgColor)
            view.layer.addSublayer(layer)
        default:
            break
        }
        return view
    }
}

```

Разберем каждый из новых элементов класса **CardView**.

Свойство **margin** определяет внутренний отступ фигуры от краев представления. Это сделано для того, чтобы фигуры на лицевой стороне не граничили с краями представления.

Два ленивых свойства **frontSideView** и **backSideView** используются для хранения вью лицевой и обратной стороны карточки.

Почему мы вообще используем свойства, зачем храним ссылки на представления? Дело в том, что доступ к представлениям нам потребуется в дальнейшем при создании анимации переворота карточки.

Почему для хранения ссылок используются ленивые свойства? Если вы уберете ключевое слово **lazy**, то Xcode сообщит вам об ошибке. Причиной этому является то, что в процессе создания экземпляра все не ленивые свойства должны быть определены, но **self**, с помощью которого производится доступ к методам **getFrontSideView** и **getBackSideView**, еще недоступен (объект еще не создан).

Обратите внимание, что при создании дочерних представлений для определения их размеров используется свойство **self.bounds**. Сейчас для вас это уже не должно быть чем-то удивительным, так как мы подробно разобрали его в предыдущих главах книги.

Также хотелось бы отметить, что лицевое व्यю содержит два व्यю: одно занимает всю площадь карточки, а второе – учитывает отступы (**margin**). Это делается для того, чтобы в случае, когда карточка перевернута, и переднее представление находится сверху, оно должно полностью перекрывать лежащее ниже его заднее. В свою очередь, заднее व्यю содержит всего одно представление.

Также в методах **getFrontSideView** и **getBackSideView** определяется фоновый цвет. Это сделано с целью исключения прозрачности представлений, чтобы они не просвечивали друг через друга.

Теперь необходимо дополнить инициализатор, чтобы в представлении начали отображаться дочерние व्यю.

- Дополните инициализатор класса **CardView** в соответствии с листингом 18.13.

ЛИСТИНГ 18.13

```
init(frame: CGRect, color: UIColor) {
    super.init(frame: frame)
    self.color = color

    if isFlipped {
        self.addSubview(backSideView)
        self.addSubview(frontSideView)
    } else {
        self.addSubview(frontSideView)
        self.addSubview(backSideView)
    }
}
```

Первая версия игровой карточки готова. Отобразим ее на сцене в Live View.

- Дополните код метода **loadView** класса **MyViewController** в соответствии с листингом 18.14.

ЛИСТИНГ 18.14

```
override func loadView() {
    let view = UIView()
    view.backgroundColor = .white
    self.view = view

    // ...

    let firstCardView = CardView<CircleShape>(frame: CGRect(x: 0, y: 0,
width: 120, height: 150), color: .red)
    self.view.addSubview(firstCardView)
}
```

Теперь в зависимости от значения свойства **isFlipped** класса **CardView** на сцене будет отображаться либо узор рубашки, либо красный круг. Обратите внимание, что в некоторых случаях, если **isFlipped** равно **true**, на краях карточки могут быть видны элементы рубашки (рис. 18.11). Это происходит из-за того, что круги или линии на узоре рубашки создаются в случайных местах, а представление **CardView** не обрезает выходящие за его пределы элементы. Далее мы исправим этот момент.

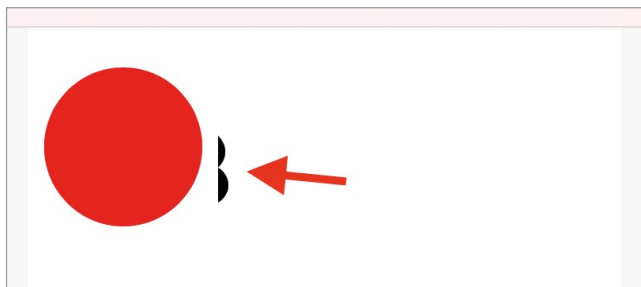


Рис. 18.11. Артефакты при отображении карточки

Создание границ карточки

Пока что карточка визуально неотличима от фона, поэтому нам необходимо добавить для нее визуальные границы.

- Добавьте в класс **CardView** свойство **cornerRadius** и метод **setupBorders**, также добавив его вызов в инициализатор (листинг 18.15).

ЛИСТИНГ 18.15

```
// радиус закругления
var cornerRadius = 20

init(frame: CGRect, color: UIColor) {
```



```
// ...

setupBorders()
}

// настройка границ
private func setupBorders(){
    self.clipsToBounds = true
    self.layer.cornerRadius = CGFloat(cornerRadius)
    self.layer.borderWidth = 2
    self.layer.borderColor = UIColor.black.cgColor
}
```

Свойство **cornerRadius** определяет радиус закругления углов игровой карточки, а метод **setupBorders** обеспечивает настройку всех границ.

Теперь игральная карточка выглядит намного приятнее, так как видны ее границы, а визуальные артефакты отсутствуют (рис. 18.12).

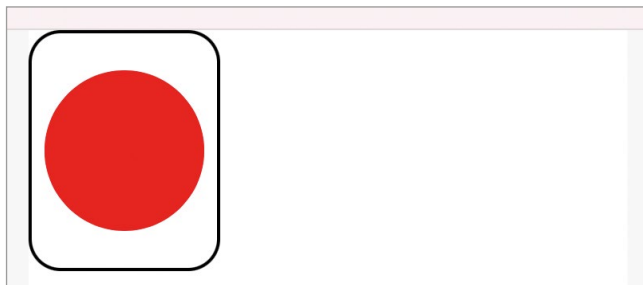


Рис. 18.12. Игральная карточка на сцене

Отображение на сцене второй карточки

Попробуем добавить на сцену вторую карточку. При этом важно, чтобы она имела отличное от первой карточки значение свойства **isFlipped**.

- Измените значение по умолчанию свойства **isFlipped** класса **CardView** на **false**.
- Добавьте в метод **loadView** класса **MyViewController** вторую игральную карточку (листинг 18.16).

ЛИСТИНГ 18.16

```
override func loadView() {

    // ...
```

```
// игральная карточка рубашкой вверх
let firstCardView = CardView<CircleShape>(frame: CGRect(x: 0, y: 0,
width: 120, height: 150), color: .red)
self.view.addSubview(firstCardView)

// игральная карточка лицевой стороной вверх
let secondCardView = CardView<CircleShape>(frame: CGRect(x: 200, y: 0,
width: 120, height: 150), color: .red)
self.view.addSubview(secondCardView)
secondCardView.isFlipped = true
}
```

Несмотря на то, что для второй игровой карточки изменяется значение свойства **isFlipped** на **true**, на сцене обе карточки все равно отображаются рубашкой вверх (рис. 18.13).

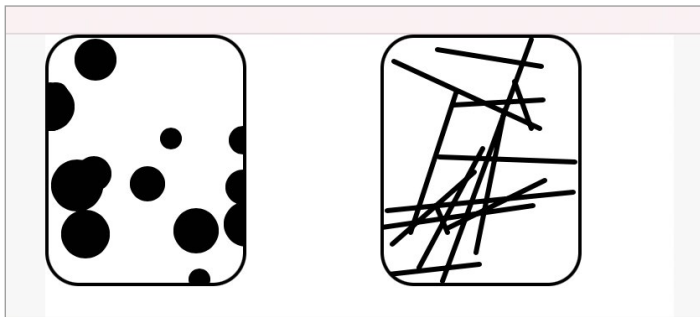


Рис. 18.13. Две игральные карточки на сцене

Но почему так происходит? Об этом мы поговорим в следующем разделе.

18.4 Как представления появляются на экране

iOS и **UIKit** умеют очень эффективно управлять ресурсами устройства, стараясь максимально экономно расходовать их. Эта экономия чувствуется практически во всем, в том числе и в системе генерации и отображения пользовательского интерфейса, о которой мы поговорим в этом разделе.

Как вы знаете, любая сцена состоит из иерархии представлений, что относится и, в том числе, к сцене, отображаемой сейчас в Live View. В процессе подготовки сцены к выводу на экран каждое представление, входящее в его состав, формируется, создается его моментальный снимок (так называемый снапшот, который по сути представляет из себя растровую картинку), после чего выво-

дится на экран. Таким образом, после того, как сцена сформирована и отображена, в памяти хранятся экземпляры классов представлений и их снимшоты.

Несколько раз в секунду (в случае iPhone – это 60 или 120, в зависимости от частоты обновления экрана, измеряемой в герцах) происходит цикл обновления, в ходе которого сцена заново выводится на экран. В ходе **цикла обновления** представления, которые подверглись изменению, заново генерируются, для них повторно создается и сохраняется снимшот и они выводятся на экран. Такой подход позволяет очень эффективно расходовать ресурсы, так как для неизменившихся представлений используется созданный ранее моментальный снимок. В этом случае задача системы состоит в том, чтобы загрузить снимшот из памяти и вывести его на экран.

Повторная генерация представления в ходе цикла обновления производится в следующих случаях:

- вью впервые появляется на экране;
- у вью изменилась область видимости, например, часть вью перекрылась другим вью;
- изменяется значение свойства **isHidden**;
- был вызван метод **setNeedsDisplay()** или метод **setNeedsDisplay(_:)**, отмечающие вью как «требующее обновления».

Именно в таком подходе кроется решение проблемы, из-за которой при изменении значения свойства **isFlipped** для второй игровой карточки она не переворачивается. Дело в том, что при первой генерации карточки используется значение по умолчанию **false** этого свойства. В результате карточка генерируется и помещается в память. Несмотря на то, что мы пытаемся изменить **isFlipped** на **true**, при добавлении вью в иерархию и отображении на сцене используется созданный ранее снимшот, на котором карточка повернута рубашкой вверх. Само по себе изменение **isFlipped** не приводит к перерисовке представления на следующем цикле обновления.

Одной из причин повторной генерации представления, указанной выше, является использование методов **setNeedsDisplay()** и **setNeedsDisplay(_:)**, вызов которых вы можете осуществить самостоятельно в своем программном коде. После того, как один из методов будет вызван, вью будет помечено как «требующее обновления», вследствие чего будет перерисовано в следующем цикле обновления.

Разница методов **setNeedsDisplay()** и **setNeedsDisplay(_:)** заключается в следующем.

- Метод **setNeedsDisplay()** сообщает о том, что вью необходимо перерисовать полностью.
- Метод **setNeedsDisplay(_:)** принимает на вход параметр типа **CGRect**,

содержащий информацию об области вью, требующей обновления. С его помощью вы можете перерисовать только тот прямоугольник, который требуется.

В нашем случае после изменения свойства **isFlipped** карточку необходимо перерисовать полностью, поэтому добавим вызов данного метода в наблюдатель свойства.

► Измените свойство **isFlipped** в соответствии с листингом 18.17.

ЛИСТИНГ 18.17

```
var isFlipped: Bool = false {  
    didSet {  
        self.setNeedsDisplay()  
    }  
}
```

Теперь при каждом изменении значения свойства **isFlipped** вью будет обновляться.

Примечание Хочу отметить, что использование **setNeedsDisplay()** и **setNeedsDisplay(:)** приводит не к моментальному обновлению элемента! Вью перерисовывается только в ходе следующего цикла обновления. Даже если вы вызовете один из методов пять раз подряд, вью будет обновлено всего один раз.

Посмотрим, что изменилось на сцене (рис. 18.14). В общем то ничего – обе карточки как были повернуты рубашкой вверх, так и остались.

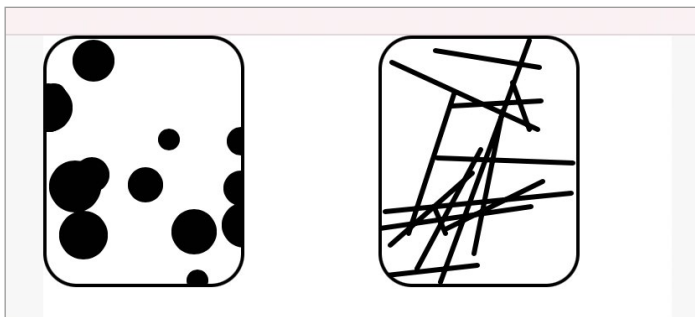


Рис. 18.14. Две игральные карточки на сцене

Посмотрите на код класса **CardView** и скажите, в какой момент происходит формирование внешнего вида представления? В ходе создания экземпляра в инициализаторе – именно там лицевая и обратная стороны добавляются в иерархию представлений! Но при каждой генерации представления применяется другой метод, который пока что не используется в нашем классе.

► В классе **CardView** создайте метод **draw** и перенесите из инициализатора код, добавляющий дочерние представления (листинг 18.18).

ЛИСТИНГ 18.18

```
init(frame: CGRect, color: UIColor) {
    super.init(frame: frame)
    self.color = color

    // ... тут находился код создания дочерних представлений

    setupBorders()
}

override func draw(_ rect: CGRect) {
    // удаляем добавленные ранее дочерние представления
    backSideView.removeFromSuperview()
    frontSideView.removeFromSuperview()

    // добавляем новые представления
    if isFlipped {
        self.addSubview(backSideView)
        self.addSubview(frontSideView)
    } else {
        self.addSubview(frontSideView)
        self.addSubview(backSideView)
    }
}
```

Запустите код на исполнение и посмотрите на панель **Live View** (рис. 18.15). Теперь карточки выглядят именно так, как требуется.

Метод **draw** производит отрисовку элементов внутри представления. Все дочерние элементы, которые должны обновляться в ходе циклов обновления, должны быть отрисованы в данном методе. В качестве входного параметра `rect` данный метод принимает область, требующую обновления. В ходе первой

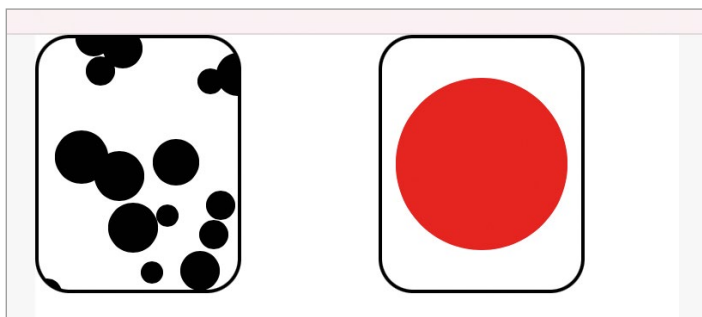


Рис. 18.15. Две игральные карточки на сцене

отрисовки в данном параметре обычно передается прямоугольник, соответствующий видимой части представления, но в последующем он может указывать только на определенную часть кода.

Примечание Никогда не вызывайте метод `draw` напрямую. Если вам требуется обновить вью, вызывайте `setNeedsDisplay()` или `setNeedsDisplay(_:)`.

Создание кастомных представлений с помощью программного кода – это не единственный возможный вариант. Как и в случае с ячейками таблиц, вы можете использовать xib-файлы, что в некоторой степени может облегчить процесс разработки. Но оставим этот материал на будущее. Полученные вами знания позволят вам с успехом в дальнейшем продолжить изучение темы создания кастомных графических элементов.

Глава 19.

События и анимации в iOS

В этой главе вы:

- узнаете, что такое события, с точки зрения iOS;
- познакомитесь с Responder Chain (цепочкой ответчиков);
- реализуете реакцию на события с помощью программного кода;
- создадите несколько анимаций.

События и анимации – с первого взгляда такие несовместимые элементы программирования, которые по какой-то причине оказались вместе в одной главе. В приложении «**Cards**» они будут тесно связаны между собой, так как при возникновении некоторых событий на сцене будут активироваться определенные анимации, например, по нажатию на игральную карточку будет происходить ее анимированный переворот.

Далее в главе мы подробно разберем, что такое события, как они работают и как могут быть использованы для запуска анимаций.

19.1 События

Возможно, в вашем представлении iOS-приложения – это своего рода фабрики. Они работают круглосуточно, без обедов и выходных, в несколько смен, на которых постоянно и скрытно от пользователя решаются некоторые задачи. Вы не видите сам процесс функционирования «завода», но совершенно точно знаете, что его персонал постоянно занят работой.

Но если ваше представление об iOS-приложении именно такое, вы совершенно неправы!

Это может звучать довольно странно, но большую часть времени любое приложение ничего не делает. Совершенно ничего! Оно простаивает! Любое iOS-приложение является **событийно-ориентированным**, т.е. оно ждет, пока произойдет что-то, на что можно отреагировать. Другими словами, оно ждет наступления **события** (event).

В операционной системе iOS существует множество различных событий: тапы, свайпы и жесты, push-уведомления, движения и перевороты устройства, события по таймеру и многие другие. Любые из этих событий могут стать причиной того, чтобы ваше приложение приступило к выполнению реализованных в ней функций.

Примечание **Тапы, свайпы и жесты** – это три типа событий, основанных на взаимодействии пользователя с тачскрином устройства.

Тапы – это клики по экрану одним или несколькими пальцами. Так, вы, к примеру, запускаете приложения.

Свайпы – это непрерывное движения пальца по экрану в одну сторону. Так, вы, к примеру, пролистываете загруженную интернет-страницу в браузере.

Жесты – это сложные движения одним или несколькими пальцами. Так, например, вы увеличиваете масштаб подложки в приложении «Карты» (разводите или сводите два пальца).

Любое приложение состоит из множества типовых элементов, которые взаимодействуют между собой с целью решения определенных задач. Это взаимодействие не происходит просто так, а их причиной становятся те самые события.

Рассмотрим несколько примеров.

Пользователь нажал на иконку приложения на домашнем экране, то есть произошло событие «Запуск приложения». Операционная система iOS обрабатывает прикосновение пальца к экрану, определяет, какое приложение должно быть запущено и передает необходимые данные в фреймворк **UIKit**, вызывая метод **UIApplicationMain** (о нем мы говорили в первой части книги), который возвращает экземпляр класса **UIApplication**.

Получается, что событие «Запуск приложения» привело к поэтапной передаче данных от операционной системы до экземпляра запущенного приложения:

```
iOS -> UIKit -> UIApplication
```

Передаваемые данные (данные о событии) содержали причину запуска (приложение может быть запущено не только при нажатии на иконку на домашнем экране, но и при обработке push-уведомления, выполнении выбранной операции из меню быстрых действий и т.д.), а также другие вспомогательные данные.

В результате проделанных действий приложение было запущено, а его интерфейс отобразился на экране. Далее оно «засыпает» до наступления нового события.

Теперь пользователь нажимает на кнопку, размещенную на сцене. Физически, конечно, никакая кнопка не нажимается, а происходит касание пальца с экраном, то есть происходит событие типа «Касание». Операционная система группирует данные о событии и передает их в фреймворк **UIKit**, а тот в экземпляр **UIApplication** и далее, пока они не достигнут пункта назначения, где смогут быть обработаны, т.е. пока не достигнут нажатой виртуальной кнопки и привязанного к ней обработчика нажатия (например, экшн-метода). Далее, после выполнения определенных разработчиком операций, приложение вновь «засыпает».

Разные элементы приложения могут обрабатывать различные типы событий, например, графические элементы могут обрабатывать касания, а экземпляр приложения – поступающие push-уведомления. В некоторых элементах функция обработки уже встроена (бегунок перемещается по слайдеру, не требуя от разработчика дополнительных действий), а для некоторых вы можете реализовать ее самостоятельно.

19.2 События касания

Для обработки событий, возникающих в процессе жизненного цикла приложения, служат разнообразные механизмы. Так, например, для работы с касаниями используется класс **UIResponder**, для обработки push-уведомлений – Notification Center, для запуска таймера – класс **Timer** и т.д.

В реализуемом нами приложении событием, после которого должен активироваться очередной этап игры, является «Касание» игровой карточки. После его наступления карточка переворачивается, открывая пользователю изображенную на ней фигуру, и запускается код, выполняющий проверку идентичности двух перевернутых карточек. Но игральная карточка в нашем случае реализована с помощью класса **UIView**, который не является кнопкой (**UIButton**), поэтому мы не можем просто привязать к нему экшн-метод так, как мы это делали ранее.

Для обработки касаний используется класс **UIResponder**, который имеет всю требуемую для обработки любых типов касаний, в том числе нажатий, функциональность. Получается, что для обработки касания по игровой карточке нам необходимо совместить **UIView** и **UIResponder**.

Откройте справку к классу **UIView** и обратите внимание на его родительский класс (рис. 19.1).

Класс **UIView** уже является наследником **UIResponder**, а это значит, что он по умолчанию инкапсулирует все его возможности. Помните, в главе про **UIView** было сказано, что представление – это контейнер, наделенный возможностью обработки событий касания? Эта функциональность появляется именно вследствие данного наследования.

Примечание Помимо **UIView** класс **UIResponder** является родительским для многих других классов, в частности, **UIApplication**, **UIApplicationDelegate**, **UIViewController** и **UIWindow**, а также для всех графических элементов, входящих в состав **UIKit**.

Вообще класс **UIResponder** содержит в себе функциональность для реагирования не только на события касания, но и на нажатия физических клавиш, движения устройства (например, встряхивание) и команды от внешних аксессуаров (например, гарнитуры). Это становится возможным благодаря переопределению в дочернем классе (который должен обрабатывать события) специальных методов. К примеру, для обработки касаний используются следующие четыре метода:

- `touchesBegan(_:with:)` – палец коснулся экрана;
- `touchesMoved(_:with:)` – палец движется по экрану;
- `touchesEnded(_:with:)` – палец прекратил касание;
- `touchesCancelled(_:with:)` – касание отменено.

Примечание Данные методы уже были использованы нами в первой книге, когда мы занимались разработкой приложения с перемещением шариков.

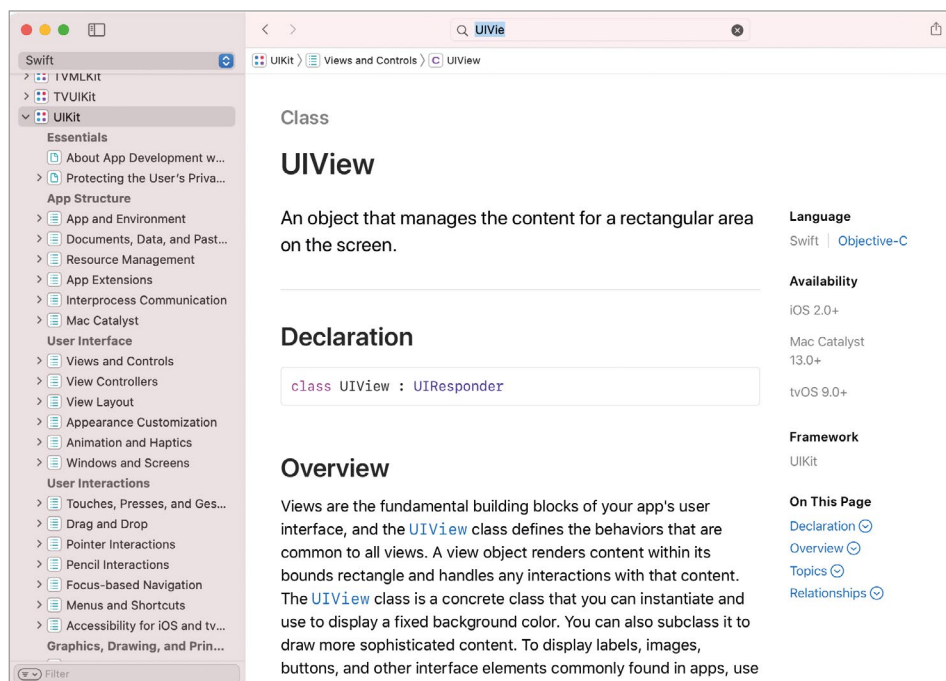


Рис. 19.1. Справка к классу UIView

Для обработки нажатий физических клавиш используются методы **pressesBegan**, **pressesChanged**, **pressesEnded**, **pressesCancelled**, для движений – **motionBegan**, **motionEnded** и **motionCancelled**, а для внешних устройств – метод **remoteControlReceived**.

В приложении «**Cards**» требуется реализовать обработку касаний, поэтому нас будут интересовать методы группы **touches**. Попробуем добавить их в проект.

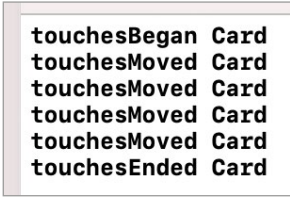
- ▶ Откройте проект «**Cards**».
- ▶ Перейдите к playground-файлу, в котором определен класс **CardView**.
- ▶ Реализуйте в теле класса методы из листинга 19.1.

ЛИСТИНГ 19.1.

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {  
    print("touchesBegan Card")  
}  
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {  
    print("touchesMoved Card")  
}  
  
override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {  
    print("touchesEnded Card")  
}
```

- ▶ Запустите playground-проект.
- ▶ На панели **Live View** найдите графический элемент «Игральная карточка», наведите на него указатель, нажмите левую кнопку мыши, переместите указатель в произвольное место и завершите нажатие. Таким образом, вы симулировали касание пальцем, если бы приложение было запущено на мобильном устройстве.

В процессе выполнения нажатия на консоли появлялись надписи, соответствующие вызовам методов класса **CardView** (пример показан на рисунке 19.2). В начале касания там отобразилось сообщение «**touchesBegan Card**», в процессе перемещения – «**touchesMoved Card**», а после того, как кнопка мыши была отпущена – «**touchesEnded Card**». Чем дальше вы переместите указатель, тем больше сообщений «**touchesMoved Card**» будут выведено.



```
touchesBegan Card  
touchesMoved Card  
touchesMoved Card  
touchesMoved Card  
touchesMoved Card  
touchesMoved Card  
touchesEnded Card
```

Рис. 19.2. Сообщения, информирующие о произошедших событиях

В момент, когда возникло событие «Касание», данные о нем были переданы в экземпляр **CardView**, описывающий игральную карточку, в результате чего был вызван метод **touchesBegan** и выведено сообщение на консоль.

Далее, в ходе перемещения указателя (с зажатой кнопкой) при каждом изменении его координат генерировалось событие «Перемещение», данные о котором вновь доставлялись в экземпляр **CardView**, но при этом теперь вызывался метод **touchesMoved**.

Аналогичная ситуация произошла и в момент окончания касания, вследствие которого был вызван метод **touchesEnded**.

Классы `UIEvent` и `UITouch`

Когда система обнаруживает одно из событий, которое может быть обработано с помощью `UIResponder`, `UIKit` создает экземпляр класса `UIEvent`, описывающий данное событие, после чего он отправляется в приложение. В случае с событиями касания также создается один или несколько экземпляров класса `UITouch` (в зависимости от количества прикосновений), содержащих информацию о касании, включая его координаты.

Взгляните на входные параметры переопределенных в классе `CardView` методов – ими как раз и являются значения типов `UITouch` и `UIEvent`. Таким образом, внутри методов данные значения могут быть использованы для решения необходимых задач, например, для определения координат касания. Далее в главе мы рассмотрим подобный пример. Сейчас же поговорим о том, как данные, «упакованные» в типы `UIEvent` и `UITouch`, путешествуют внутри приложения.

19.3 Responder Chain

Для начала проведем небольшой эксперимент.

- ▶ Из класса `CardView` удалите методы группы `touches`.
- ▶ В классе `MyViewController` объявите метод `touchesBegan` в соответствии с листингом 19.2.

ЛИСТИНГ 19.2

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {  
    print("touchesBegan Controller")  
}
```

- ▶ Запустите playground-проект.
- ▶ На панели **Live View** щелкните по игровой карточке.

Несмотря на то, что вы взаимодействовали с игровой карточкой, при щелчке на нее на консоли отображались сообщения, соответствующие вызову метода `touchesBegan`, объявленного в классе `MyViewController` (рис. 19.3). Случай, когда информация о событии передается старшему элементу, является стандартным для iOS. Причиной такого поведения является Responder Chain (цепочка ответчиков).

Responder Chain – это список вероятных ответчиков, т.е. список объектов, которые потенциально могут обработать возникшее событие. Если очередной элемент списка (цепочки) не может обработать событие, оно передается следующему в списке элементу. Именно так и произошло в примере выше. На рисунке 19.4 показана цепочка ответчиков, используемая при обработке собы-

тий касания игровой карточки. Сама карточка (ее дочерние представления и класс **CardView**) не смогла его обработать, поэтому событие было передано далее, пока не достигло контроллера, который содержит реализацию метода **touchesBegan**.

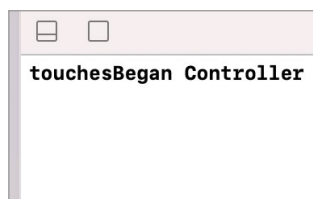


Рис. 19.3. Сообщения, информирующие о произошедших событиях

Примечание Responder Chain – это реализация шаблона разработки «Цепочка обязанностей». Его суть состоит в том, что элементы поочередно опрашиваются на предмет возможности обработки поступивших данных. Если один элемент не смог обработать его, данные о событии передаются следующему элементу в цепочке и т.д.

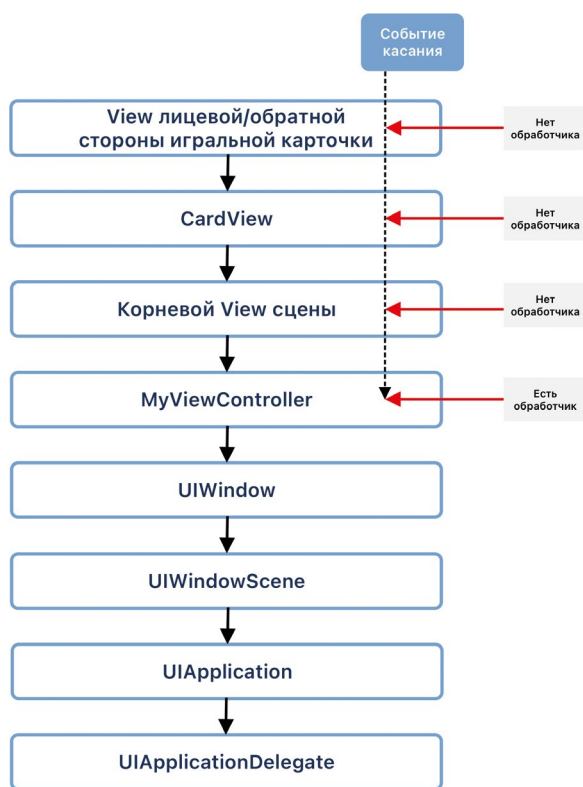


Рис. 19.4. Пример Responder Chain

Теперь поговорим о том, на основе чего строится Responder Chain.

Как вы могли заметить, порядок элементов цепочки очень похож на порядок элементов иерархии представлений, дополненный экземплярами классов **UIApplicationDelegate**, **UIApplication** и **MyViewController**. Тот элемент, по которому происходит касание, назначается **первым ответчиком** (first responder). Если он может обработать событие, он делает это, если нет – передает управление следующему элементу цепочки, и цикл вновь повторяется до тех пор, пока не будет найден элемент, способный обработать событие.

Все элементы цепочки ответчиков являются наследниками класса **UIResponder**.

В зависимости от того, какого элемента на сцене касается пользователь, Responder Chain будет начинаться в том или ином месте и всегда спускаться вниз по структуре. Если произошло касание белого фона сцены, то именно корневое представление будет назначено первым ответчиком, и уже от него начнется создание цепочки.

Переход к следующему элементу цепочки происходит с помощью свойства **next** текущего элемента. Данное свойство определено в классе **UIResponder**, а значит доступно во всех его подклассах. Оно возвращает либо ссылку на следующий ответчик, либо **nil**.

Проведем небольшой эксперимент.

- ▶ В playground-проекте определите расширение для класса **UIResponder**. С его помощью мы сможем отобразить в консоли список элементов цепочки ответчиков (листинг 19.3).

ЛИСТИНГ 19.3

```
extension UIResponder {
    func responderChain() -> String {
        guard let next = next else {
            return String(describing: Self.self)
        }
        return String(describing: Self.self) + " -> " + next.responderChain()
    }
}
```

Примечание Конструкция **Self.self** позволяет получить название типа данных рассматриваемого значения.

- ▶ Удалите из класса **MyViewController** метод **touchesBegan**.
- ▶ В классе **CardView** реализуйте метод **touchesBegan** в соответствии с листингом 19.4.

ЛИСТИНГ 19.4

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {  
    print( self.responderChain() )  
}
```

► Нажмите на игральную карточку на сцене.

Теперь при касании экземпляра класса **CardView** на консоли будет отображаться состав Responder Chain, начиная с данного элемента:

```
CardView<CircleShape> -> UIView -> MyViewController -> UIDropShadowView  
-> UITransitionView -> UIWindow -> UIWindowScene -> UIApplication ->  
XCAppDelegate
```

Примечание Системные классы **UIDropShadowView** и **UITransitionView** уже встречались нам ранее при рассмотрении иерархии представлений. Вы никогда не будете работать с ними напрямую, поэтому я не учитываю их на схеме цепочки ответчиков.

Определение первого ответчика

В зависимости от того, какое именно событие из поддерживаемых классом **UIResponder** произошло, элемент, являющийся первым ответчиком (тем, кто примет данные о событии), может отличаться.

Первый ответчик – это тот элемент, который в данный момент готов принимать данные о произошедшем событии. Обычно это тот элемент, который активировал клавиатуру и ожидает от нее ввод данных, например, текстовое поле.

Но изначально на сцене нет никакого первого ответчика. В случае события «Касание» он определяется следующим образом:

- **UIKit** создает объект типа **UIEvent**, подробно описывающий событие;
- данный объект отправляется в **UIApplication** и, далее, в **UIWindow**;
- **UIWindow** проводит так называемое hit-тестирование, т.е. с помощью поэтапного вызова метода **hitTest(_:with:)** для каждого вложенного представления, определяет, с каким именно графическим элементом взаимодействует пользователь;
- как только данный элемент найден, он назначается **первым ответчиком**, и начиная от него стартует обход **Responder Chain**.

Для всех остальных типов событий (нажатия физических кнопок, движения гарнитуры и команд от внешних устройств) никакого hit-тестирования не проводится. Данные передаются сразу в элемент, который в текущий момент является первым ответчиком (если он был определен ранее).

Вы можете влиять на то, какой элемент должен быть первым ответчиком в данный момент с помощью вызова метода **becomeFirstResponder()**. Напри-

мер, если требуется активировать текстовое поле сразу после отображения сцены на экране, это можно сделать с помощью данного метода, например, во **viewDidAppear** контроллера:

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
    textField.becomeFirstResponder()  
}
```

Теперь текстовое поле является первым ответчиком и будет, в том числе, принимать сообщения о событиях и при необходимости передавать его вглубь цепочки ответчиков.

Хотите скрыть клавиатуру и деактивировать текстовое поле – просто укажите первым ответчиком другой элемент (с помощью нажатия на него или метода **becomeFirstResponder()**) или вызовите метод **resignFirstResponder()** для текущего первого ответчика:

```
textField.resignFirstResponder()
```

Примечание В дальнейшем вы нередко будете встречаться с необходимостью использования методов **becomeFirstResponder** и **resignFirstResponder**. Поэтому возьмите на заметку описанный выше пример их применения для текстового поля.

19.4 Пример обработки событий. Перемещение игровых карточек

Добавим в наш проект интерактивности и реализуем в классе **CardView** обработку событий касания для того, чтобы пользователь мог перемещать игровые карточки по игровому полю.

Все реализованные ранее методы группы **touches** имеют два входных параметра: **touches** типа **UITouch** и **event** типа **UIEvent**. Мы можем использовать их для решения нашей задачи. Параметр **event** описывает произошедшее событие, **touches** содержит множество совершенных касаний. В нашем случае, в нем находится всего один элемент, так как по умолчанию для экземпляра **UIView** отключена поддержка мультитач (одновременное касание несколькими пальцами).

Примечание Для включения поддержки мультитач конкретному выю необходимо установить значение **true** в свойстве **isMultipleTouchEnabled**.

Используя данные из входных параметров, мы можем организовать перемещение карточек по сцене с помощью пальца (или курсора мышки).

- Добавьте в класс **CardView** свойство **anchorPoint** и методы группы **touches** в соответствии с листингом 19.5.

ЛИСТИНГ 19.5

```
// точка привязки
private var anchorPoint: CGPoint = CGPoint(x: 0, y: 0)

override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    anchorPoint.x = touches.first!.location(in: window).x - frame.minX
    anchorPoint.y = touches.first!.location(in: window).y - frame.minY
}

override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    self.frame.origin.x = touches.first!.location(in: window).x -
    anchorPoint.x
    self.frame.origin.y = touches.first!.location(in: window).y -
    anchorPoint.y
}
```

- ▶ Запустите playground-проект, после чего в **Live View** попробуйте переместить игральную карточку, нажав на нее.

Пользователь (в данном случае – это вы) получил возможность перемещать элементы на сцене! При этом совершенно неважно, сколько карточек находится на игровом поле – перемещение каждой из них обрабатывается независимо от других.

Свойство **anchorPoint** хранит координаты первого нажатия и в дальнейшем используется для того, чтобы верно рассчитывать значение свойства **frame**. Без него выражение **self.frame.origin.x = touches.first!.location(in: window).x** в методе **touchesMoved** приводило бы к резкому скачку карточки, совмещая левый верхний угол представления с точкой нажатия.

Метод **location**, применяемый к значению **UITouch**, позволяет получить координаты касания в переданном ему в качестве аргумента представлении. В нашем случае передается **window**, и возвращаются координаты касания в **UIWindow**, в котором отображается игральная карточка.

19.5 Анимации графических элементов

Вы можете использовать события для реализации многих идей в ваших проектах. Например, с их помощью можно создавать игры, требующие перемещения элементов: карты, шахматы и многие другие. Также события могут активировать различные анимации, например, поворота, изменения размеров, перемещения. И все это плавно, без участия пользователя и с очень удобным API.

Примечание Анимации в iOS-разработке живут независимо от событий: вы можете использовать их как вместе, так и раздельно. В данном случае я покажу лишь пример того, что анимации могут быть запущены при определенных событиях.

Создавать анимации на Swift очень легко! Если говорить кратко, вы просто указываете, какое свойство необходимо анимировать, и сколько времени эта анимация должна занять. Хотите изменить позицию представления? Для этого укажите новые координаты, и Swift все сделает сам, плавно передвинув графический элемент.

Для создания анимации используется класс **UIView** и несколько входящих в него статических методов **animate**. Если написать название этого метода в редакторе кода, то в окне автодополнения отобразятся все доступные варианты (рис. 19.5). Каждый из них отличается от остальных набором доступных возможностей.

Примечание В данном разделе мы не будем подробно изучать возможности методов **animate**, ограничившись наиболее простым из них **animate(withDuration:animations:)**. Все остальные методы вы сможете рассмотреть самостоятельно, с вашим навыком работы с документацией это не составит особого труда.

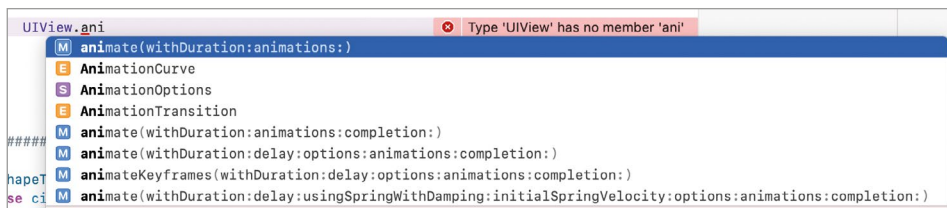


Рис. 19.5. Методы для создания анимации

В общем случае метод **animate** позволяет организовать плавное изменение свойств объекта, например, его координат или размеров. В качестве последнего аргумента (он именуется **animations** или **completion** в зависимости от используемого варианта) данный метод принимает замыкание, содержащее указания на финальные значения свойств, изменения которых необходимо анимировать.

К примеру, следующий код

```
// начальные координаты представления
view.frame.origin = CGPoint(x: 10, y: 10)
// запуск анимации
UIView.animate(withDuration: 1) {
    // конечные координаты представления
    view.frame.origin = CGPoint(x: 100, y: 100)
}
```

плавно, в течение 1 секунды, перемещает представление **view** из точки с координатами **(10, 10)** в точку **(100, 100)**.

Добавим в проект несложную анимацию, возвращающую перемещенную игральную карточку в свою исходную позицию.

► Объявите в классе **CardView** свойство **startTouchPoint** (листинг 19.6).

ЛИСТИНГ 19.6

```
private var startTouchPoint: CGPoint!
```

В данное свойство будут записываться исходные координаты игровой карточки.

- Доработайте методы **touchesBegan** и **touchesEnded** класса **CardView** в соответствии с листингом 19.7.

ЛИСТИНГ 19.7

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {  
    // изменяем координаты точки привязки  
    anchorPoint.x = touches.first!.location(in: window).x - frame.minX  
    anchorPoint.y = touches.first!.location(in: window).y - frame.minY  
  
    // сохраняем исходные координаты  
    startTouchPoint = frame.origin  
}  
  
override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {  
    // анимировано возвращаем карточку в исходную позицию  
    UIView.animate(withDuration: 0.5) {  
        self.frame.origin = self.startTouchPoint  
    }  
}
```

Примечание Метод **touchesMoved**, обеспечивающий перемещение карточки по сцене, был реализован ранее. Удалять его не требуется.

- Запустите playground-проект, переместите карточку в любое место на сцене и завершите касание.

Как только касание будет прекращено, карточка незамедлительно устремится в свою исходную позицию. И для этого нам потребовалось написать всего несколько строчек кода!

При нажатии на карточку срабатывает метод **touchesBegan**, и данные о текущих координатах сохраняются в свойство **startTouchPoint**. После того, как перемещение карточки будет завершено, вызывается метод **touchesEnded**, и с помощью **animate** графический элемент возвращается в исходную позицию. Данный метод анимирует свойство, значение которого указывается в замыкании, передаваемом в качестве последнего аргумента. Анимация длится столько времени, сколько передано в аргументе **withDuration** (в нашем случае – это 0,5 секунды).

С помощью **animate** у вас появляется возможность анимировать различные свойства, не ограничиваясь координатами и размерами. К примеру, можно изменять значение свойства **transform**.

Доработаем анимацию таким образом, чтобы вместе с перемещением в исходную позицию карточка дополнительно переворачивалась на 180°.

► Доработайте метод **touchesEnded** в соответствии с листингом 19.8.

ЛИСТИНГ 19.8

```
override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    // анимировано возвращаем карточку в исходную позицию
    UIView.animate(withDuration: 0.5) {
        self.frame.origin = self.startTouchPoint

        // переворачиваем представление
        if self.transform.isIdentity {
            self.transform = CGAffineTransform(rotationAngle: .pi)
        } else {
            self.transform = .identity
        }
    }
}
```

► Переместите игральную карточку по сцене.

Как только нажатие будет завершено, карточка не просто вернется в исходную позицию, но и перевернется на 180°.

Анимации – это очень круто! Они способны сделать интерфейс приложения по-настоящему интересным. Помните, что у метода **animate** есть множество реализаций, и мы попробовали лишь самую простую из них.

19.6 Анимированные переходы

Анимации плавно изменяют значения указанных свойств представления, но при решении некоторых задач их применение может быть довольно сложным. Например, в проекте «**Cards**» при нажатии на игральную карточку должен происходить ее анимированный переворот, то есть представление, которое в иерархии находится ниже в результате, должно оказаться выше. Использование метода **animate** для реализации такой функциональности будет довольно затруднительно, так как потребуется очень хорошо продумать анимацию, ее тайминги и реализовать плавное многоэтапное изменение целой группы свойств.

Для создания анимированных переходов между представлениями, когда одно должно заменить другое, используются специальный метод **transition**, который, как и **animate**, является статическим методом класса **UIView**. Он «под капотом» уже содержит реализацию сложной анимации, для создания которой вам потребовались бы значительные усилия.

► В классе **CardView** реализуйте тело метода **flip** (листинг 19.9).

ЛИСТИНГ 19.9

```
func flip() {  
    // определяем, между какими представлениями осуществить переход  
    let fromView = isFlipped ? frontSideView : backSideView  
    let toView = isFlipped ? backSideView : frontSideView  
    // запускаем анимированный переход  
    UIView.transition(from: fromView, to: toView, duration: 0.5, options:  
[.transitionFlipFromTop], completion: nil)  
    isFlipped = !isFlipped  
}
```

Примечание Также для изменения значения свойства **isFlipped** можно использовать метод **toggle**.

```
isFlipped.toggle()
```

Свойство **isFlipped**, реализованное в классе **CardView** еще в предыдущей главе, позволяет определить значения параметров **fromView** и **toView**, то есть от какого к какому представлению совершить переход. Данные параметры нужны по причине того, что карточка может находиться либо лицевой, либо обратной стороной к пользователю.

Метод **transition** запускает анимированный переход. В качестве аргументов в него передаются два представления, между которыми необходимо произвести анимированный переход (**from** и **to**), длительность (**duration**), настройки анимации (**options**) и обработчик завершения анимации, выполняемый после окончания перехода.

Свойство **options** может принять произвольное количество значений типа **UIView.AnimationOptions**, определяющих настройки анимации. Так, переданное значение **.transitionFlipFromTop** указывает на то, что переход должен производиться путем переворота представлений вверх.

► Измените тело метода **touchesEnded** в соответствии с листингом 19.10.

ЛИСТИНГ 19.10

```
override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {  
    flip()  
}
```

► Запустите playground-проект и попробуйте переместить игральную карточку.

Как только клавиша мыши будет отпущена, карточка немедленно перевернется. Если попытаться сделать это вновь, эффект будет таким же. И вновь для реализации такой функциональности нам потребовалось всего несколько строк кода.

19.7 Доработка игровой карточки

Исправление артефактов при перевороте

Обратите внимание на то, что при перевороте карточки в ее углах отображаются артефакты (рис. 19.6).

- Если фоновая картинка (круги или линии) заползает на скругленный угол, то в момент переворота скругление исчезает.
- Тень отображается за пределами карточки.

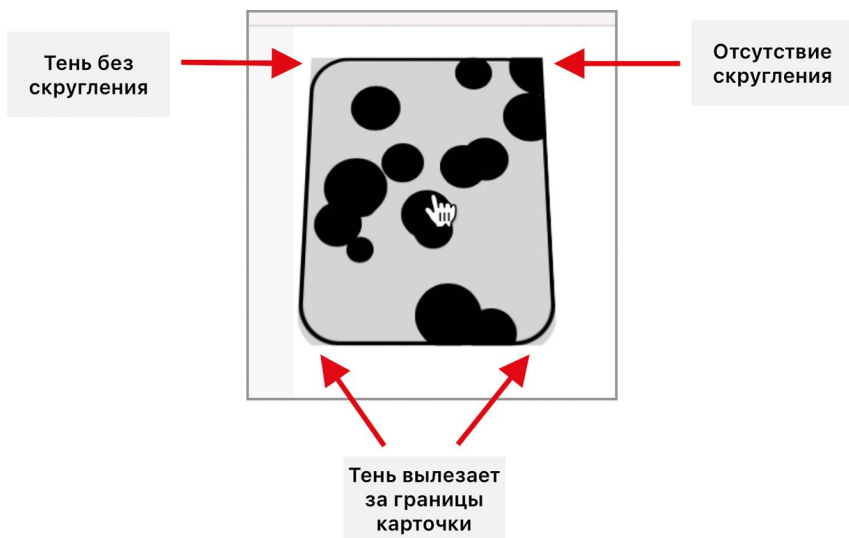


Рис. 19.6. Артефакты в ходе анимации

Дело в том, что в процессе перехода корневое представление класса **CardView** не влияет на дочерние представления (лицевую и обратную стороны), а ведь именно для него определяются скругления углов. Для решения данной проблемы необходимо скруглить углы обоих дочерних представлений.

► Дополните методы в соответствии с листингом 19.11.

ЛИСТИНГ 19.11

```
private func getFrontSideView() -> UIView {  
  
    // ...  
  
    // скругляем углы корневого слоя  
    view.layer.masksToBounds = true  
    view.layer.cornerRadius = CGFloat(cornerRadius)
```

```

    return view
}

private func getBackSideView() -> UIView {

    // ...

    // скругляем углы корневого слоя
    view.layer.masksToBounds = true
    view.layer.cornerRadius = CGFloat(cornerRadius)

    return view
}

```

► Попробуйте перевернуть карточку на сцене.

Теперь при перевороте карточки больше нет никаких артефактов (рис. 19.7).

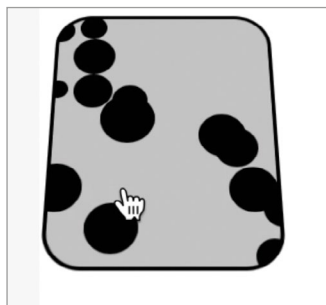


Рис. 19.7. Анимация переворота без артефактов

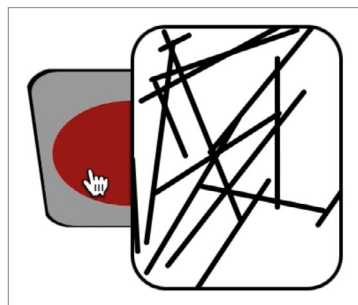


Рис. 19.8. Неверная иерархия представлений

Изменение иерархии представлений

Второй недостаток текущей реализации проявляется в том случае, когда на сцене находятся несколько игровых карточек. Если одна из них находится ниже в иерархии представлений, и мы активируем ее переворот, то карточка так и остается снизу, а это выглядит несколько странно (рис. 19.8).

Эту проблему можно решить, сделав так, чтобы карточка не просто переворачивалась, но и перемещалась вверх иерархии представлений. Вся сложность данной задачи состоит в том, что событие касания доставляется непосредственно в игровую карточку и обрабатывается только там, а иерархией карточек управляет корневое представление сцены (родительский к карточке элемент). Нам требуется реализовать такой механизм, при котором в определенный момент данные о событии будут передаваться наверх, где и будет производиться перемещение карточек.

Для решения данной проблемы воспользуемся созданным ранее свойством **flipCompletionHandler** класса **CardView** и организуем в нем изменение иерархии представлений так, чтобы игральная карточка оказывалась в самом ее верху.

- Добавьте в метод **flip** класса **CardView** вызов обработчика завершения переворота (листинг 19.12).

ЛИСТИНГ 19.12

```
func flip() {
    let fromView = isFlipped ? frontSideView : backSideView
    let toView = isFlipped ? backSideView : frontSideView
    UIView.transition(from: fromView, to: toView, duration: 0.5, options:
[.transitionFlipFromTop], completion: { _ in
        // обработчик переворота
        self.flipCompletionHandler?(self)
    })
    isFlipped.toggle()
}
```

Теперь после того, как анимация переворота карточки завершится, будет вызван соответствующий обработчик, если конечно свойство **flipCompletionHandler** не **nil**.

- Убедитесь, что на сцене находятся минимум две игральные карточки. При необходимости добавьте их в методе **loadView** класса **MyViewController**.
- Для каждой из игровых карточек инициализируйте значение свойству **flipCompletionHandler** так, как это показано в листинге 19.13.

ЛИСТИНГ 19.13

```
firstCardView.flipCompletionHandler = { card in
    card.superview?.bringSubviewToFront(card)
}
```

Метод **bringSubviewToFront** осуществляет перенос представления на передний план, то есть поднимает карточку в иерархии таким образом, чтобы мы могли увидеть ее полностью.

- Попробуйте переместить карточки так, чтобы они перекрывали друг друга.

Теперь если игральная карточка находится ниже при перемещении, то сперва она будет перевернута, а уже после этого перенесена вверх по иерархии представлений.

Примечание Логичным было бы реализовать механизм, при котором игральная карточка сперва переносилась вверх по иерархии, а уже после этого переворачивалась. Но при таком подходе есть одна проблема: обработчик **flipCompletionHandler** помимо переноса представления на передний план в дальнейшем также будет про-

изводить и определение идентичности перевернутых карточек. И если эту операцию производить до переворота, одинаковые карточки будут исчезать с поля еще до того, как будут перевернуты.

Для решения этой проблемы было бы логичным создать два обработчика: один вызывается до переворота, а другой – после. В первом мы могли бы переносить карточку вверх, а во-втором – производить проверку на идентичность.

В нашем случае мы оставим один обработчик с целью не усложнять проект, но в дальнейшем вы можете самостоятельно произвести требуемую доработку.

Переворот при клике

Сейчас игральная карточка осуществляет переворот в конце перемещения, независимо от того, перемещали мы карточку или просто кликнули по ней. Но что, если пользователь не стремится переворачивать ее, а просто хочет переместить ее в другую позицию?

Для этого переработаем обработчики касания класса **CardView** таким образом, чтобы перед переворотом происходила проверка того, была ли перемещена карточка (изменились ли ее исходные координаты). Если перемещения не было (то есть был просто клик), то карточка переворачивается.

► В классе **CardView** измените тело метода в соответствии с листингом 19.14.

ЛИСТИНГ 19.14

```
override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {  
    if self.frame.origin == startTouchPoint {  
        flip()  
    }  
}
```

Теперь карточка будет переворачиваться только при клике по ней. В ином случае она будет оставаться в текущем состоянии.

На этом мы завершаем работу над игровой карточкой. Мы реализовали все необходимое для генерации ее внешнего вида и, самое важное, узнали новый для вас материал о событиях и создании анимаций. В следующей главе мы займемся созданием игрового поля и бизнес-логики программы, после чего вас ждет самостоятельная работа.

Глава 20.

Разработка приложения

В этой главе вы:

- научитесь верстке, основанной на фреймах, с помощью программного кода;
- разработаете Модель и Представление приложения «Cards»;
- познакомитесь с шаблоном проектирования «Простая фабрика»;
- проведете несколько доработок созданной игры.

К настоящему моменту вы прошли внушительный путь, состоящий из огромного количества практического и теоретического материала, который, я надеюсь, позволил вам открыть для себя много нового. Вообще, в моем случае путь становления в качестве iOS-разработчика стал одним из самых интересных путешествий в жизни. Наша специализация интересна тем, что в какую-то тему не копни, всегда можно найти что-то новое. И кажется, только ты изучал алгоритм «Бинарный поиск», как внезапно понял, что уже пытаешься реализовать на Swift структуру данных «Бинарное дерево», а еще через мгновение понимаешь, что уже изучаешь порядок хранения различных типов данных в оперативной памяти (Swift Memory Layout).

У Swift всегда есть что-то, чем вас удивить!

Но вернемся к учебному материалу.

В процессе работы над книгой я предполагал, что данная глава станет одной большой самостоятельной работой, в которой вы с минимальной поддержкой с моей стороны привели бы проект в чувство: разработали структуру файлов, разнесли по ним типы, реализовали Модель, отображали Представление и заставили бы Контроллер управлять приложением. Такая работа стала бы своеобразным выходным экзаменом, довольно сложным, но очень эффективным. Тем не менее, для меня стало очевидным, что все эти задачи лучше выполнить совместно, ведь именно так я смогу контролировать тот факт, что вы движетесь в правильном направлении, а значит в результате добьетесь максимального эффекта.

В этой главе мы совместно завершим разработку приложения «**Cards**». Вся работа будет построена по следующему плану.

1. Перенос реализованных типов из playground-файла в файлы проекта.

Это необходимо сделать в по той причине, что классы, перечисления и структуры сейчас доступны только внутри playground и не видны в самом проекте.

2. Реализация Модели и Представления проекта.

При этом мы будем активно дорабатывать Контроллер, так как именно он обеспечивает связь этих элементов проекта.

3. Проведение вами самостоятельной доработки проекта.

Реализованная нами функциональность будет представлять из себя самый минимум, которым должна обладать подобная программа. По этой причине не будет лишним обсудить и исправить «недоработки», тем самым расширив возможности игры.

20.1 Распределение типов по проекту

Playground-файл, в котором проводилась вся разработка в последних главах, содержит значительное количество кастомных типов данных, описывающих такие сущности, как «Игральная карта», «Фигуры» и т.д. Для того, чтобы созданные нами классы, структуры и перечисления могли быть использованы в проекте, их необходимо перенести из playground. Сделаем это с учетом используемой архитектуры MVC.

- ▶ Создайте папку **View**.
- ▶ Создайте файл **/View/Shapes.swift**.
- ▶ Создайте файл **/View/Cards.swift**.
- ▶ Осуществите перенос типов в соответствии с данными, приведенными в таблице 20.1

Примечание В некоторых случаях вам потребуется подключить (импортировать) библиотеку **UIKit**, чтобы обеспечить доступ к используемым типам. Помните, что **UIKit** автоматически подключает **Foundation**.

Считайте это небольшой самостоятельной работой.

Таблица 20.1. Перенос типов внутри проекта

Элемент playground-проекта	Место назначения
protocol ShapeLayerProtocol	/View/Shapes.swift
extension ShapeLayerProtocol	/View/Shapes.swift
class BackSideCircle	/View/Shapes.swift

class BackSideLine	/View/Shapes.swift
class FillShape	/View/Shapes.swift
class CrossShape	/View/Shapes.swift
class CircleShape	/View/Shapes.swift
class SquareShape	/View/Shapes.swift
class CardView	/View/Cards.swift
protocol FlippableView	/View/Cards.swift

- ▶ Удалите из проекта все playground-файлы.
- ▶ Создайте папку **/View/Storyboards**.
- ▶ Перенесите все storyboard-файлы в созданную папку.
- ▶ Удалите файл **ViewController.swift**.
- ▶ Создайте в проекте папку **Controller**.
- ▶ В новой папке создайте новый файл, содержащий класс **BoardGameController**. Данный класс должен быть дочерним по отношению к **UIViewController**.
- ▶ Свяжите вью контроллер, размещенный на сториборде **Main.storyboard** с классом **BoardGameController**.
- ▶ Создайте папку **Model**.

Теперь используемая структура файлов и папок, а также порядок распределения по ним созданных типов позволят нам проводить дальнейшую разработку. Но прежде поговорим о текущем состоянии проекта с точки зрения MVC.

На рисунке 20.1 показана текущая схема элементов в составе проекта, разделенных по ролям в паттерне MVC.

Примечание В первую очередь, хочу обратить ваше внимание на то, что на текущем этапе разработки проект будет включать всего одну сцену, которая уже находится в файле **Main.storyboard**. Данные элементы не отражены в схеме на рисунке 20.1.



Рис. 20.1. Структура проекта

20.2 Разработка Модели

В текущем состоянии Модель в проекте состоит только из папки **Model**. Да – пустого прямоугольника на рисунке 20.1 (шутка:). Говоря иначе, Модель отсутствует как класс. А ведь именно ее данные являются первичными для всего приложения. На основе массива случайных карточек будет генерироваться игровое поле.

Примечание Цель, которая стоит перед нами, состоит в том, чтобы сделать структуру проекта максимально простой и доступной. Мы не станем создавать большое количество сущностей в составе Модели, а просто инкапсулируем всю логику в едином классе. Далее, при необходимости, вы сможете самостоятельно провести рефакторинг так, как мы делали это ранее для проекта «**Right on target**».

В самом простом случае Модель должна выполнять две основные задачи:

- генерировать множество случайных карточек;
- проверять две карточки на эквивалентность.

Основной сущностью, которой будет управлять Модель, является «Игральная карточка». Подождите, но ведь «Игральная карточка» уже реализована нами с помощью класса **CardView**, который входит в состав Представления! Получается, что один и тот же тип будет использоваться и в Модели, и в Представлении?

Вовсе нет! Хотя работа с карточками, с одной стороны, относится к Представлению, так как говорит о их размещении на сцене, с другой стороны – это Модель, так как на основе этой сущности строится вся бизнес-логика игры. Но по требованиям MVC для реализации этой логики нельзя использовать уже созданные классы Представления, так как Представление и Модель должны быть максимально независимы.

Получается, что в Модели должна присутствовать собственная реализация сущности «Игральная карточка», включающая множество типов, отличных от тех, что используются в Представлении.

Примечание Разделим сущности «Игральная карточка» для Модели и Представления:

- когда я буду говорить о сущности «**Карточка**», я буду подразумевать «Игральную карточку» в составе Модели;
- когда я буду говорить о сущности «**Игральная карточка**», я буду подразумевать данную сущность в составе Представления.

Реализация сущности «Карточка»

С точки зрения Модели сущность «Карточка» должна определять всего два свойства: тип фигуры на лицевой стороне и ее цвет. Этих данных вполне достаточно для того, чтобы понять, идентичны ли две карточки между собой. Никаких других знаний о карточке Модели не требуется. Помните, что вопрос отрисовки игральных карточек на сцене не относится к Модели.

Создадим два перечисления, содержащих множество значений, определяющих типы фигур и цвета, а также класс, описывающий сущность «Карточка».

- ▶ Создайте файл **/Model/Card.swift**.
- ▶ Добавьте в файл код из листинга 20.1.

ЛИСТИНГ 20.1

```
import UIKit

// типы фигуры карт
enum CardType: CaseIterable {
    case circle
    case cross
    case square
    case fill
}

// цвета карт
enum CardColor: CaseIterable {
    case red
    case green
    case black
    case gray
    case brown
    case yellow
    case purple
    case orange
}

// игральная карточка
typealias Card = (type: CardType, color: CardColor)
```

Игральная карточка представлена в виде псевдонима типа **(type: CardType, color: CardColor)**, который является кортежем. Такой способ позволит не усложнять проект, создавая дополнительный тип, так как кортеж вполне может справиться с возложенной на него задачей хранения информации о карточке и при этом позволит сравнивать карточки между собой.

Используемые в кортеже перечисления **CardType** и **CardColor** содержат список типов фигур и их цветов. Обратите внимание, что при их определении используется протокол **CaseIterable**, благодаря которому у перечисления появляется свойство **allCases**, возвращающее коллекцию всех его элементов. Это наделяет перечисление дополнительными возможностями, например:

```
// обход элементов перечисления
for enumElement in CardType.allCases {
```

```
// ...  
}  
  
// получение случайного элемента перечисления  
let randomEnumelement = CardColor.allCases.randomElement()!
```

Протокол **CaseIterable** очень полезен. Запомните его и применяйте в разработке!

На этом работа над сущностью «Карточка» завершена. Уверен, что это было намного проще, чем вы ожидали. Старайтесь не искать сложных решений – в первую очередь, используйте максимально простые варианты реализации.

Сравнение карточек

Одной из основных операций, которая будет использоваться в процессе игры, является сравнение игровых карточек между собой. Сами карточки будут переворачиваться с помощью функций Представления (на игровом поле), а вот непосредственно их сравнение должно производиться в Модели.

Тип **Card** является псевдонимом к кортежу, который включает в себя два перечисления. Перечисления, в свою очередь, по умолчанию поддерживают функцию сравнения элементов:

```
CardColor.red == CardColor.green // false  
CardType.circle == CardType.circle // true
```

Поэтому для проверки идентичности карточек мы просто будем сравнивать два кортежа:

```
(type: .circle, color: .green) == (type: .circle, color: .yellow) // false
```

Чуть позже мы инкапсулируем данную логику в специальный класс, отвечающий за «Игру».

Сущность «Игра» в Модели

По аналогии с проектом «**Right on target**», для управления всей бизнес-логикой игры создадим специальный класс **Game**.

- ▶ В папке **Model** создайте файл **Game.swift**.
- ▶ В новом файле объявите класс **Game** (листинг 20.2).

ЛИСТИНГ 20.2

```
class Game {}
```

Каким минимальным набором функций должен обладать класс **Game**? Очевидно, что в его функции должна входить возможность генерации массива случайных карточек и его хранения для последующей передачи в Представление и сравнения отдельных элементов.

Примечание На данном этапе мы пропустим функциональность подсчета совершенных действий, а также другие возможности игры. В первую очередь, необходимо реализовать MVP (минимальную жизнеспособную версию продукта) – это понятие, с которым мы уже встречались ранее.

Значимость реализации MVP в повседневной разработке тяжело переоценить, так как излишек функций на первом этапе разработки может сыграть с вами злую шутку, и в результате вы рискуете так и не завершить проект.

► Доработайте код класса **Game** в соответствии с листингом 20.3.

ЛИСТИНГ 20.3

```
class Game {
    // количество пар уникальных карточек
    var cardsCount = 0
    // массив сгенерированных карточек
    var cards = [Card]()

    // генерация массива случайных карт
    func generateCards() {
        // генерируем новый массив карточек
        var cards = [Card]()
        for _ in 0...cardsCount {
            let randomElement = (type: CardType.allCases.randomElement()!,
color: CardColor.allCases.randomElement()!)
            cards.append(randomElement)
        }
        self.cards = cards
    }

    // проверка эквивалентности карточек
    func checkCards(_ firstCard: Card, _ secondCard: Card) -> Bool {
        if firstCard == secondCard {
            return true
        }
        return false
    }
}
```

Свойство **cardsCount** содержит общее количество пар уникальных карточек. Данное свойство используется в методе **generateCards** для создания массива случайных карточек.

Свойство **cards** содержит массив пар карточек, который создается с помощью метода **generateCards**.

На рисунке 20.2 показано текущее распределение элементов в проекте по ролям MVC.



Рис. 20.2. Элементы в составе проекта

Вся требуемая функциональность Модели создана. И вновь процесс ее реализации оказался довольно простым.

Связь Модели и Контроллера

В текущем состоянии проект содержит полнофункциональную Модель, отдельные элементы Представления, а также пока еще пустой Контроллер. Сейчас все перечисленные компоненты совершенно ничего не знают друг о друге. В первую очередь мы произведем подключение Модели к Контроллеру.

► Добавьте в класс **BoardGameController** код из листинга 20.4.

ЛИСТИНГ 20.4

```
// количество пар уникальных карточек
var cardsPairsCounts = 8
// сущность "Игра"
lazy var game: Game = getNewGame()

private func getNewGame() -> Game {
    let game = Game()
    game.cardsCount = self.cardsPairsCounts
    game.generateCards()
    return game
}
```

Ленивое свойство **game** будет использоваться для хранения настроенного экземпляра игры. По вашему мнению, почему оно является ленивым (**lazy**)?

Во-первых, в момент создания экземпляра класса контроллера **BoardGameController** значение данного свойства не используется – в нем нет никакой необходимости. Это значит, что «ленивый вариант» позволит слегка увеличить скорость создания экземпляра класса, отложив инициализацию значения типа **Game** до момента первого доступа к нему.

Во-вторых, такой подход позволил нам вынести код генерации «Игры» в отдельный метод **getNewGame**.

20.3 Разработка Представления

Одной из «особенностей» шаблона MVC от Apple является очень высокий уровень связи между Представлением и Контроллером (контроллер даже называется View Controller, то есть контроллер представления). В чем же она выражается? Дело в том, что представления не могут существовать без Контроллера, и именно Контроллер обеспечивает отображение представлений на сцене, в том числе за счет методов жизненного цикла.

Как бы вы не хотели разделить Представление и Контроллер, но на данном этапе это станет чрезвычайно сложной задачей. Позже, когда вы изучите другие шаблоны проектирования, позволяющие выделять в проекте иные функциональные роли, вы узнаете варианты, способы и решения максимального выноса задач отображения из вью контроллера.

Для чего я это сказал? Дело в том, что в результате высокой связанности Представления и Контроллера большую часть кода, реализующего отображение элементов (который по факту должен быть частью Представления), в действительности мы будем писать в классе Контроллера.

Варианты создания графического интерфейса

Вообще реализация графического интерфейса – это один из наиболее творческих процессов в разработке мобильных приложений. И не только потому, что всегда хочется создать одновременно простой и функциональный интерфейс, а еще и потому, что визуально одинаковый результат может быть достигнут разными методами. Вспомните материал про создание ячеек табличных представлений, когда один и тот же вариант оформления создавался двумя способами: горизонтальным стеком и констрейнтами.

В общем случае можно выделить следующие способы разработки интерфейса.

1. По типу процесса верстки – определяет каким образом ведется размещение графических элементов на сцене:
 - a. с помощью программного кода;
 - b. с помощью **Interface Builder**.
2. По системе позиционирования – определяет, каким образом вычисляются позиции графических элементов на сцене:
 - a. Auto Layout (ограничения, Size Classes и др.);
 - b. фреймы.
3. По используемым элементам – одни и те же задачи отображения данных можно решить с помощью различных элементов: таблиц, коллекций, стеков, обычных представлений и т.д.

Выбор какого-либо из указанных способов не говорит о том, что всю разработку в проекте вы должны проводить исключительно с помощью него. Довольно часто разработчики применяют комбинированный стиль, когда работа в **Interface Builder** смешивается с версткой кодом, а констрейнты используются совместно с фреймами. Такой подход не всегда очевиден и понятен, но довольно гибок. Со временем вы получите достаточно опыта, чтобы применять любой из них в своей практике разработки.

Подавляющее большинство Junior Swift Developer (начинающие разработчики) при верстке графического интерфейса используют **Interface Builder**. И это вовсе не удивительно, так как позиционировать вьюшки с помощью мышки очень удобно и самое важное – наглядно. Но моя цель – развивать в вас различные навыки, связанные с разработкой. Поэтому мы будем верстать сцену исключительно средствами программного кода, а в качестве системы позиционирования используем фреймы.

О системах позиционирования мы уже неоднократно говорили. Так, ограничения позволяют задавать правила размещения элементов, а использование фреймов (свойства **frame** представления) требует предварительного расчета координат и размеров графического элемента.

Структура интерфейса игры «Cards»

На рисунке 20.3 показан интерфейс приложения «**Cards**», который будет разработан нами в этой главе. В его верхней части расположена кнопка запуска/перезапуска игры, а ниже нее – игровое поле. Таким образом, сцена будет иметь следующую иерархию представлений.

View – корневое представление сцены.

- **UIButton** – кнопка запуска/перезапуска игры.
- **UIView** – игровое поле.
 - **[CardView]** – множество игровых карточек, состоящих из лицевой и обратной сторон.

Размещение кнопки запуска игры

Обычно для размещения кнопки на сцене мы совершали следующие операции:

1. производили поиск кнопки в библиотеке объектов;
2. перемещали кнопку на сцене и позиционировали ее;
3. проводили визуальное оформление кнопки;
4. реализовывали экшн-метод и связывали его с кнопкой.

Создание кнопки средствами программного кода будет отличаться, так как никаких библиотек, storyboard и экшн-методов в данном случае нет.

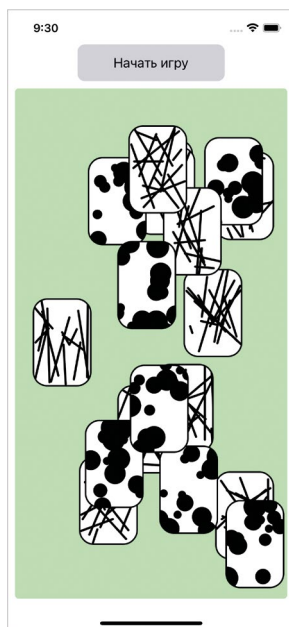


Рис. 20.3. Интерфейс приложения «Cards»

- ▶ Откройте файл **BoardGameController.swift**.
- ▶ Добавьте в класс код из листинга 20.5.

ЛИСТИНГ 20.5

```
// кнопка для запуска/перезапуска игры
lazy var startButtonView = getStartButtonView()

private func getStartButtonView() -> UIButton {
    // 1
    // Создаем кнопку
    let button = UIButton(frame: CGRect(x: 0, y: 0, width: 200, height: 50))
    // 2
    // Изменяем положение кнопки
    button.centerX = view.centerX

    // 3
    // Настраиваем внешний вид кнопки

    // устанавливаем текст
    button.setTitle("Начать игру", for: .normal)
    // устанавливаем цвет текста для обычного (не нажатого) состояния
```

```
        button.setTitleColor(.black, for: .normal)
        // устанавливаем цвет текста для нажатого состояния
        button.setTitleColor(.gray, for: .highlighted)
        // устанавливаем фоновый цвет
        button.backgroundColor = .systemGray4
        // скругляем углы
        button.layer.cornerRadius = 10

    return button
}
```

Свойство **startButtonView** будет хранить в себе ссылку на кнопку запуска игры. Оно является ленивым, а значит его значение будет создано при первом доступе к нему.

Метод **getStartButtonView** возвращает сконфигурированную кнопку, готовую к размещению на сцене. Разберем данный метод по шагам.

Шаг 1. Создание кнопки

На первом шаге создается экземпляр класса **UIButton**, описывающий кнопку. Изначально его координаты определяются как (0, 0). Это говорит о том, что кнопка будет находиться в левом верхнем углу родительского представления. Размеры (ширина и высота) кнопки подобраны опытным путем так, чтобы они могли вмещать весь ее текст.

Шаг 2. Изменение координат кнопки

Кнопка перемещается в центр горизонтальной оси родительского представления. Таким образом, она займет требуемую нам позицию, а не будет расположена в углу представления.

Шаг 3. Изменение внешнего вида

Методы **setTitle** и **setTitleColor** класса **UIButton** позволяют установить текст и цвет кнопки для ее различных состояний. Состояние – это значение типа **UIControl.State**, определяющее текущий статус элемента (не нажат, нажат, выделен и др.). Остальные операции данного шага уже хорошо вам известны.

Загрузка представлений из состава сцены происходит в методе **loadView** вью контроллера. Его также желательно использовать и для размещения кастомных представлений, которыми в нашем случае являются кнопка и игровое поле с карточками.

► В классе **BoardGameController** добавьте метод **loadView** (листинг 20.6).

ЛИСТИНГ 20.6

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

```
// добавляем кнопку на сцену
view.addSubview(startButtonView)
}
```

► Произведите запуск приложения на симуляторе.

Обратите внимание на внешний вид сцены и ошибку позиционирования кнопки. По какой-то причине она размещается вплотную к верхней границе сцены, перекрывая статус-бар (рис. 20.4). А это явно не то, что нам требовалось получить.

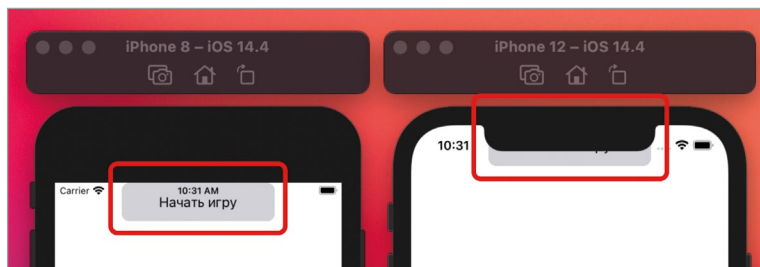


Рис. 20.4. Ошибка позиционирования кнопки

Причина довольно банальна: при размещении кнопки мы добавляли его в корневое представление, которое занимает всю площадь окна (**window**). При этом совершенно не использовали Safe Area, который как раз и применяется для того, чтобы элементы интерфейса не перекрывали системные элементы.

Для решения данной проблемы необходимо обратиться к экземпляру окна и получить верхний отступ Safe Area, после чего использовать его при определении координат кнопки.

► Дополните метод **getStartButtonView** в соответствии с листингом 20.7.

ЛИСТИНГ 20.7

```
private func getStartButtonView() -> UIButton {
    let button = UIButton(frame: CGRect(x: 0, y: 0, width: 200, height:
50))
    button.center.x = view.center.x

    // получаем доступ к текущему окну
    let window = UIApplication.shared.windows[0]
    // определяем отступ сверху от границ окна до Safe Area
    let topPadding = window.safeAreaInsets.top
    // устанавливаем координату Y кнопки в соответствии с отступом
    button.frame.origin.y = topPadding

    // ...
}
```

► Запустите проект на симуляторе.

Теперь кнопка находится в верхней части сцены и не перекрывается статус-баром (рис. 20.5).

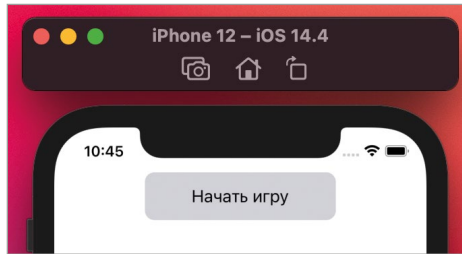


Рис. 20.5. Корректная позиция кнопки

Связь кнопки и метода

При верстке интерфейса в **Interface Builder** для связи кнопки с кодом используются экшн-методы. В нашем случае графический редактор интерфейса не используется, поэтому потребуется найти иной способ запуска программного кода по нажатию на кнопку.

Один из вариантов вы уже знаете: использование событий типа «Касание». Но класс **UIButton** является наследником класса **UIControl**, а значит поддерживает специальный метод **addTarget**.

СИНТАКСИС

Метод `UIControl.addTarget(_:action:for:)`

Привязывает событие `for` к вызову некоторого метода `action`.

Аргументы

- `_`: Any? – целевой объект, в котором будет происходить поиск метода `action`; если указать `nil`, то метод будет искаться по всей цепочке ответчиков (Responder Chain).
- `action`: Selector – селектор, идентифицирующий вызываемый метод.
- `for`: UIControl.Event – действие, к которому привязывается вызов метода.

Пример

```
button.addTarget(nil, action: #selector(startGame(_:)), for: .touchUpInside)
```

Прежде чем привязать к нажатию на кнопку вызов метода, объявим его.

► Добавьте в класс **BoardGameController** метод **startGame** (листинг 20.8).

ЛИСТИНГ 20.8

```
@objc func startGame(_ sender: UIButton) {
    print("button was pressed")
}
```

Обратите внимание на новый для вас элемент – атрибут **@objc**. Он используется в тех случаях, когда отмеченный им элемент должен быть доступен в Objective-C и его среде выполнения.

Но что это значит? Ведь наш проект написан полностью на Swift и не использует ни строчки кода на Objective-C.

В этом вы полностью правы, но все дело в историческом развитии Swift. Долгое время он создается поверх уже реализованного с помощью Objective-C окружения, и большая часть фреймворков из состава iOS SDK все еще написаны на Objective-C, а значит приложение по факту работает в среде Objective-C.

Метод **addTarget** вызывается в ходе исполнения приложения (runtime) и обращение к нему происходит средствами Objective-C, а не Swift. Именно по этой причине и используется атрибут **@objc**. Он позволяет сделать метод видимым для Objective-C.

► Дополните метод **getStartButtonView** в соответствии с листингом 20.9.

ЛИСТИНГ 20.9

```
private func getStartButtonView() -> UIButton {  
  
    // ...  
  
    // подключаем обработчик нажатия на кнопку  
    button.addTarget(nil, action: #selector(startGame(_)), for:  
    .touchUpInside)  
  
    return button  
}
```

Для создания ссылки на метод используется **селектор** – значение типа **Selector**, определяемое с помощью следующего синтаксиса:

- первым указывается ключевое слово **#selector** (с префиксом **#**);
- далее в скобках – название вызываемого метода.

Вы можете использовать несколько вариантов написания имени метода в селекторе:

```
#selector(startGame)  
#selector(startGame(_))  
#selector(BoardGameController.startGame)
```

Теперь проверим созданную связь.

► Запустите приложение и нажмите на кнопку, размещенную на сцене.

При каждом нажатии кнопки на консоль будет выводиться сообщение, соответствующее вызову метода **startGame** (рис. 20.6).

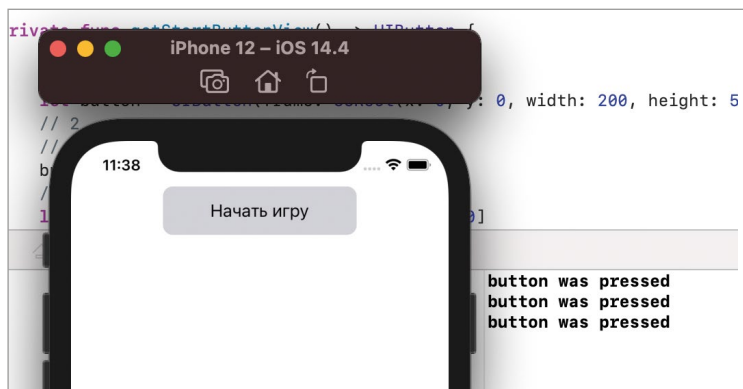


Рис. 20.6. Вывод на консоли

Примечание В iOS 14 появились «свифтовые» аналоги метода **addTarget** – методы, основанные на использовании типа **UIAction**.

Если ваш код должен поддерживать исключительно iOS 14+, то теперь обработчик для той же кнопки можно реализовать следующим образом:

```
button.addAction(UIAction(title: "", handler: { action in
    print("Button was pressed")
}), for: .touchUpInside)
```

В момент написания книги предыдущие версии iOS все еще довольно активно используются, поэтому использование **addTarget**, а значит и понимание связанных с ним концепций, являются важным навыком.

Размещение игрового поля

Следующим графическим элементом, который необходимо добавить, является игровое поле – класс **BoardGameView**.

► Добавьте в класс **BoardGameController** код из листинга 20.10.

ЛИСТИНГ 20.10

```
// игровое поле
lazy var boardGameView = getBoardGameView()

private func getBoardGameView() -> UIView {
    // отступ игрового поля от ближайших элементов
    let margin: CGFloat = 10

    let boardView = UIView()
```

```

    // указываем координаты
    // x
    boardView.frame.origin.x = margin
    // y
    let window = UIApplication.shared.windows[0]
    let topPadding = window.safeAreaInsets.top
    boardView.frame.origin.y = topPadding + startButtonView.frame.height +
margin

    // рассчитываем ширину
    boardView.frame.size.width = UIScreen.main.bounds.width - margin*2
    // рассчитываем высоту
    // с учетом нижнего отступа
    let bottomPadding = window.safeAreaInsets.bottom
    boardView.frame.size.height = UIScreen.main.bounds.height - boardView.
frame.origin.y - margin - bottomPadding

    // изменяем стиль игрового поля
    boardView.layer.cornerRadius = 5
    boardView.backgroundColor = UIColor(red: 0.1, green: 0.9, blue: 0.1,
alpha: 0.3)

    return boardView
}

```

Процесс создания представления аналогичен созданию кнопки в методе **getStartButtonView**, за тем исключением, что экземпляр типа **UIView** здесь инициализируется без указания каких-либо координат и размеров. Данные параметры рассчитываются и указываются далее в коде.

Теперь разместим игровое поле на сцене.

► Дополните метод **loadView** в соответствии с листингом 20.11.

ЛИСТИНГ 20.11

```

override func loadView() {
    super.loadView()

    // добавляем кнопку на сцену
    view.addSubview(startButtonView)
    // добавляем игровое поле на сцену
    view.addSubview(boardGameView)
}

```

► Произведите запуск приложения.

Все графические элементы расположены на своих местах (рис. 20.7), а это значит, что мы можем приступить к размещению игровых карточек в соответствии с данными Модели.

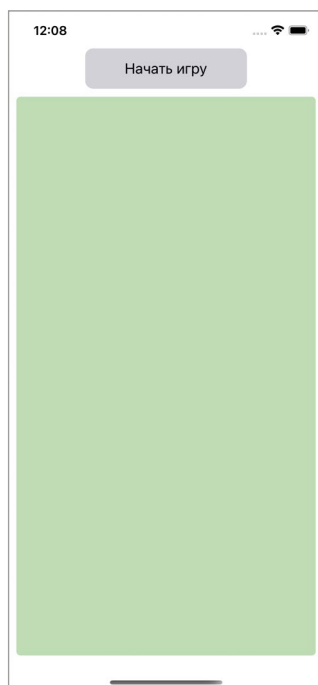


Рис. 20.7. Графический интерфейс приложения

20.4 Шаблон проектирования «Фабрика». Фабрика фигур

При попытке создания представлений карточек вы столкнетесь со следующей проблемой: у вас есть данные Модели, описывающие карточки. Но как эти данные преобразовать в графический элемент, который может быть размещен на сцене?

Конечно, можно решать задачу в лоб, и в коде строчка за строчкой генерировать нужное количество фигур, определяя их типы и цвета, тем самым создавая излишнюю связанность между Моделью, Контроллером и Представлением.

Каждая игральная карточка состоит из основного класса представления, двух классов, описывающих две ее стороны, и двух классов, определяющих фигуру на лицевой стороне и узор на обратной. В результате мы имеем небольшой «зоопарк» типов, с которым нужно как-то управляться. А что, если в приложении появятся 10, 20 или 30 различных узоров и фигур? А что, если для некоторых

классов будут использоваться иные способы создания экземпляров, требующие передачи других аргументов? Тогда процесс работы с классами станет значительно сложнее. Вы можете просто-напросто запутаться в том, где какой класс использовать. Не для этого мы изучаем MVC.

Описанная проблема является довольно типичной для программистов, вы не первые и не последние, кто столкнулся с ней. Для ее решения мы вновь обратимся к шаблонам проектирования, а если точнее, к порождающим шаблонам.

Порождающие шаблоны проектирования – это шаблоны, описывающие способ и порядок создания объектов. С их помощью можно добиться того, чтобы элементы проекта оставались независимыми от способа создания экземпляров, их количества и возможных вариаций.

Наиболее известными из порождающих шаблонов являются:

- «Простая фабрика»;
- «Абстрактная фабрика»;
- «Фабричный метод»;
- «Синглтон»;
- «Прототип»;
- «Строитель» и многие другие.

Выбор порождающих шаблонов по-настоящему большой. Со временем вы изучите многие из них, так как в процессе разработки приложений встретитесь с самыми разнообразными проблемами, требующими интересных и эффективных решений.

Одним из упомянутых выше порождающих шаблонов является «Простая фабрика» (также известный, как «Фабрика») – именно им мы и воспользуемся для решения проблемы создания классов, используемых для описания игровой карточки.

Простая фабрика предусматривает инкапсуляцию в одном объекте (данный объект называется фабрикой) логики по созданию других объектов без привязки к их конкретным типам. Проще говоря, у вас есть один класс с одним или несколькими методами, при вызове которых передаются параметры, определяющие, какой именно объект необходимо создать и вернуть. В некотором смысле с помощью «Фабрики» определяется единая точка создания некоторой группы объектов. Именно то, что и требуется в случае нашего «зоопарка».

Примечание Еще одной аналогией для объекта-фабрики является фабрика или завод из реальной жизни. На завод поступают различные элементы, из которых собирается готовое изделие.

Ранее нами были созданы перечисления **CardType** и **CardColor**, содержащие возможные варианты фигур и их цветов. Данные типы используются в Модели

для описания игровой карточки. Мы можем передавать эти данные в фабрику, чтобы в результате получать представление карточки.

К чему отнести фабрику, к Модели, Представлению или Контроллеру – вопрос довольно сложный. Я считаю, что Фабрика – это не часть Представления, а вспомогательный элемент приложения. Она не определяет и не описывает внешний вид. Она говорит о том, как преобразовать типы и значения Модели в типы и значения Представления. Скорее всего этот элемент может быть отнесен к Контроллеру, но для его хранения будем использовать отдельную папку **Helpers**.

- ▶ В структуре проекта создайте папку **Helpers**.
- ▶ В новой папке создайте файл **CardViewFactory.swift**.
- ▶ В созданном файле реализуйте класс **CardViewFactory** (листинг 20.12).

ЛИСТИНГ 20.12

```
import UIKit

class CardViewFactory {
    func get(_ shape: CardType, withSize size: CGSize, andColor color:
CardColor) -> UIView {
        // на основе размеров определяем фрейм
        let frame = CGRect(origin: .zero, size: size)
        // определяем UI-цвет на основе цвета модели
        let viewColor = getViewColorBy(modelColor: color)

        // генерируем и возвращаем карточку
        switch shape {
        case .circle:
            return CardView<CircleShape>(frame: frame, color: viewColor)
        case .cross:
            return CardView<CrossShape>(frame: frame, color: viewColor)
        case .square:
            return CardView<SquareShape>(frame: frame, color: viewColor)
        case .fill:
            return CardView<FillShape>(frame: frame, color: viewColor)
        }
    }

    // преобразуем цвет Модели в цвет Представления
    private func getViewColorBy(modelColor: CardColor) -> UIColor {
        switch modelColor {
        case .black:
            return .black
        case .red:
```

```

        return .red
    case .green:
        return .green
    case .gray:
        return .gray
    case .brown:
        return .brown
    case .yellow:
        return .yellow
    case .purple:
        return .purple
    case .orange:
        return .orange
    }
}
}

```

Код класса **CardViewFactory** довольно прост для понимания. На основании значения параметров **shape**, **size** и **color** метод **get** возвращает представление, содержащее требуемую фигуру.

Фабрика – это один из моих любимых шаблонов, так как он позволяет добиться высокого уровня удобства работы с кодом. Обязательно после завершения изучения книги вернитесь к нему, а также к другим порождающим шаблонам, и подробно изучите их.

20.5 Размещение игровых карточек на игровом поле

Следующая задача, которая стоит перед нами – по нажатию кнопки получить на основе данных Модели массив представлений, соответствующих карточкам, после чего разместить их на игровом поле.

- В классе **BoardGameController** реализуйте метод **getCardsBy** (листинг 20.13).

ЛИСТИНГ 20.13

```

// генерация массива карточек на основе данных Модели
private func getCardsBy(modelData: [Card]) -> [UIView] {
    // хранилище для представлений карточек
    var cardViews = [UIView]()
    // фабрика карточек
    let cardViewFactory = CardViewFactory()
    // перебираем массив карточек в Модели

```

```

    for (index, modelCard) in modelData.enumerated() {
        // добавляем первый экземпляр карты
        let cardOne = cardViewFactory.get(modelCard.type, withSize:
cardSize, andColor: modelCard.color)
        cardOne.tag = index
        cardViews.append(cardOne)

        // добавляем второй экземпляр карты
        let cardTwo = cardViewFactory.get(modelCard.type, withSize:
cardSize, andColor: modelCard.color)
        cardTwo.tag = index
        cardViews.append(cardTwo)
    }
    // добавляем всем картам обработчик переворота
    for card in cardViews {
        (card as! FlippableView).flipCompletionHandler = { flippedCard in
            // переносим карточку вверх иерархии
            flippedCard.superview?.bringSubviewToFront(flippedCard)
        }
    }
    return cardViews
}

```

Метод **getCardsBy** обрабатывает переданные в него данные Модели и возвращает подготовленный к отображению массив представлений игровых карточек. Для каждой карточки в Модели генерируются по две идентичные карточки в Представлении (параметры **cardOne** и **cardTwo**).

Для создания представлений используется разработанная ранее фабрика (класс **CardViewFactory**).

Особое внимание обратите на то, что свойству **tag** каждой карточки присваивается индекс карточки в массиве **modelData**. С помощью этой операции мы решаем проблему связи карточки в Модели и карточки в Представлении. Когда пользователь осуществит переворот карточки на игровом поле, мы сможем получить значение свойства **tag** и по нему загрузить элемент Модели, соответствующий перевернутой карточке, а значит сможем провести операцию сравнения двух карточек.

Теперь объявим несколько свойств, которые будут определять размеры и позиции карточек.

► Дополните класс **BoardGameController** свойствами из листинга 20.14.

ЛИСТИНГ 20.14

```

// размеры карточек
private var cardSize: CGSize {

```

```

        CGSize(width: 80, height: 120)
    }

    // предельные координаты размещения карточки
    private var cardMaxXCoordinate: Int {
        Int(boardGameView.frame.width - cardSize.width)
    }
    private var cardMaxYCoordinate: Int {
        Int(boardGameView.frame.height - cardSize.height)
    }
}

```

Свойство **cardSize** определяет размеры представлений игровых карточек. Вы можете включить сюда любую логику их расчета, основанную на размере экрана, фазе луны или другом значении. Я решил оставить карточки статичными (80 на 120 точек).

Свойства **cardMaxXCoordinate** и **cardMaxYCoordinate** будут использоваться для того, чтобы игровые карточки не могли быть размещены за пределами игрового поля. Они определяют максимальную координату по осям X и Y, которую может иметь левый верхний угол представления игровой карточки.

► Реализуйте в классе **BoardGameController** код из листинга 20.15.

ЛИСТИНГ 20.15

```

// игральные карточки
var cardViews = [UIView]()

private func placeCardsOnBoard(_ cards: [UIView]) {
    // удаляем все имеющиеся на игровом поле карточки
    for card in cardViews {
        card.removeFromSuperview()
    }
    cardViews = cards
    // перебираем карточки
    for card in cardViews {
        // для каждой карточки генерируем случайные координаты
        let randomXCoordinate = Int.random(in: 0...cardMaxXCoordinate)
        let randomYCoordinate = Int.random(in: 0...cardMaxYCoordinate)
        card.frame.origin = CGPoint(x: randomXCoordinate, y:
randomYCoordinate)
        // размещаем карточку на игровом поле
        boardGameView.addSubview(card)
    }
}
}

```


Свойство **cardViews** хранит массив представлений игровых карточек и используется в методе **placeCardsOnBoard** для удаления старых игровых карточек с игрового поля перед размещением новых. Сам метод **placeCardsOnBoard** принимает коллекцию игровых карточек, и размещает каждый ее элемент в представлении, соответствующему игровому полю. Для каждой карточки генерируются случайные координаты.

Теперь используем реализованный код для размещения карточек на сцене.

- Измените код метода **startGame** в соответствии с листингом 20.16.

ЛИСТИНГ 20.16

```
@objc func startGame(_ sender: UIButton) {  
    game = getNewGame()  
    let cards = getCardsBy(modelData: game.cards)  
    placeCardsOnBoard(cards)  
}
```

- Запустите проект и несколько раз нажмите на кнопку на сцене.

Каждое нажатие кнопки приводит к тому, что на сцене появляется новый комплект игровых карточек, то есть это условно запускает новый раунд игры.

- Переместите и переверните несколько карточек на сцене.

Карточки успешно переворачиваются, перемещаются в иерархии и по игровому полю, но мы все еще не реализовали проверку карточек на эквивалентность, а также их обратный переверт (в случае несовпадения) и удаление с игрового поля (в случае совпадения).

Для решения этой задачи нам потребуется доработать обработчик, хранящийся в свойстве **flipCompletionHandler** игровых карточек.

- Добавьте в класс **BoardGameController** свойство **flippedCards** (листинг 20.17).

ЛИСТИНГ 20.17

```
private var flippedCards = [UIView]()
```

В свойстве **flippedCards** будут храниться ссылки на перевернутые в данный момент игровые карточки. То есть, как только карточка переворачивается рубашкой вниз, она помещается в данное свойство, а при повторном перевероте – удаляется. Значение в данном свойстве будут использоваться для сравнения идентичности карточек, а также их обратного переверота или удаления с игрового поля.

- Доработайте тело метода **getCardsBy** в соответствии с листингом 20.18.

Примечание Метод `getCardsBy` получился довольно сложным и однозначно требует переработки. В дальнейшем я предлагаю вам самостоятельно произвести его рефакторинг, разделив его на несколько более простых функций.

ЛИСТИНГ 20.18

```
private func getCardsBy(modelData: [Card]) -> [UIView] {

    // ...

    // добавляем всем картам обработчик переворота
    for card in cardViews {
        (card as! FlippableView).flipCompletionHandler = { [self] flippedCard
in
            // переносим карточку вверх иерархии
            flippedCard.superview?.bringSubviewToFront(flippedCard)

            // добавляем или удаляем карточку
            if flippedCard.isFlipped {
                self.flippedCards.append(flippedCard)
            } else {
                if let cardIndex = self.flippedCards.firstIndex(of:
flippedCard) {
                    self.flippedCards.remove(at: cardIndex)
                }
            }

            // если перевернуто 2 карточки
            if self.flippedCards.count == 2 {
                // получаем карточки из данных модели
                let firstCard = game.cards[self.flippedCards.first!.tag]
                let secondCard = game.cards[self.flippedCards.last!.tag]

                // если карточки одинаковые
                if game.checkCards(firstCard, secondCard) {
                    // сперва анимировано скрываем их
                    UIView.animate(withDuration: 0.3, animations: {
                        self.flippedCards.first!.layer.opacity = 0
                        self.flippedCards.last!.layer.opacity = 0
                    })
                    // после чего удаляем из иерархии
                    self.flippedCards.first!.removeFromSuperview()
                    self.flippedCards.last!.removeFromSuperview()
                    self.flippedCards = []
                }
            }
        }
    }
}
```

```
        // в ином случае
    } else {
        // переворачиваем карточки рубашкой вверх
        for card in self.flippedCards {
            (card as! FlippableView).flip()
        }
    }
}
}
}
return cardViews
}
```

► Запустите проект на симуляторе и попробуйте сыграть.

Если вы перевернете две разные карточки, они автоматически перевернутся обратно. Если вы найдете две одинаковые карточки, они плавно исчезнут с игрового поля! MVP-версия приложения готова – мы достигли того списка возможностей, которым должна обладать игра.

Далее мы обсудим несколько доработок, которые вам необходимо сделать самостоятельно.

20.6 «Cards», версия 1.1. Самостоятельная работа

То, что мы вместе создали – это MVP будущего проекта. В нем реализован минимальный набор функциональности, при этом проект получился очень интересным, так как позволил вам попробовать в действии новые возможности Swift, а также оставил простор для реализации ваших идей.

В этом разделе мы обсудим, какие доработки можно произвести в первую очередь.

Ошибка размещения игровых карт и hit-тестирование

- Начните новую игру и переместите любую из игровых карт так, чтобы она частично выходила за пределы игрового поля (рис. 20.8).
- Нажмите на игровую карточку в той области, в которой она попадает в зону игрового поля.

В следствии проделанного действия, как и планировалось, карточка перевернется согласно реализованной в игре логике.

- Нажмите на игровую карточку в той области, где она находится вне игрового поля.

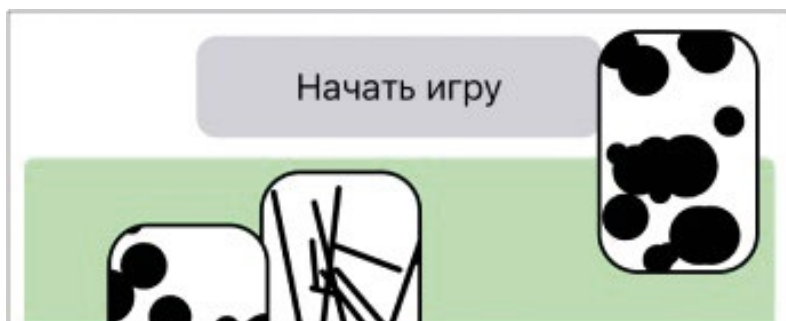


Рис. 20.8. Размещение игровой карточки

Сколько бы раз вы не нажимали в указанной выше области, карточка так и останется перевернутой. А еще хуже то, что такое поведение может привести к невозможности использования игровой карточки и, соответственно, завершению игры, так как игральная карточка, находящаяся за пределами поля, прекращает реагировать на нажатия пользователя.

Причина такого поведения в уже упомянутой ранее системе обхода представления – hit-тестировании. Дело в том, что игральная карточка является дочерним представлением для игрового поля, но при нажатии за его пределами первым ответчиком становится вовсе не карточка, так как hit-тестирование происходит по следующему порядку.

1. Проверяется, попадают ли координаты нажатия в **UIWindow** – попадают.
2. Рассматривается дочерний элемент **UIWindow** – корневое представление сцены. Проверяется, попадают ли координаты нажатия в корневое представление – попадают.
3. Далее рассматривается дочерний элемент корневого представления – кнопка. Координаты нажатия не соответствуют этому элементу.
4. И, наконец, рассматривается дочерний элемент корневого представления – игровое поле. Координаты нажатия не соответствуют и ему, а это значит, что до карточек hit-тестирование даже не дойдет.

Для решения этой задачи мы можем запретить перемещать карточки за пределы игрового поля.

Примечание Есть и альтернативные варианты устранения этой проблемы.

1. Переопределение логики обхода дочерних представлений, используемой при hit-тестировании. Это довольно сложный, на данном этапе, вариант решения.
2. Автоматически (возможно, анимировано) возвращать карточку в область игрового поля, если она была перемещена за нее.

Задание Доработайте приложение таким образом, чтобы игральные карточки не могли быть перемещены за пределы игрового поля или возвращались в него.

Новая фигура

Для того, чтобы поддерживать интерес пользователя к игре, в ней периодически должны появляться новые варианты фигур.

Задание Добавьте в игру новую фигуру «Не закрашенный круг» для лицевой стороны карточки.

Кнопка переворота всех карточек

Задание Добавьте на сцену кнопку, которая обеспечивает переворот всех карточек на лицевую или обратную сторону. При этом переворот всех карточек должен производиться на одну сторону. То есть, например, если хотя бы одна из карточек перевернута лицом вверх, все остальные карточки переворачиваются лицом вверх. Если все карточки перевернуты лицом вверх, то все они переворачиваются лицом вниз.

Экран входа

Задание Добавьте в приложение экран входа, с которого можно перейти к экрану игры. В дальнейшем на этот экран будут добавлены несколько дополнительных кнопок.

Экран настроек

Задание Добавьте в приложение один или несколько экранов настроек, на которых можно произвести следующие операции:

- выбрать количество пар одинаковых карт;
- выбрать типы карт (фигуры), используемые в игре;
- выбрать цвета карт, используемые в игре;
- выбрать узоры обратной стороны карт, используемые в игре.

Вход на экран настроек должен происходить либо с экрана входа, либо с экрана текущей игры.

Подсчет выполненных переворотов

Задание Реализуйте в приложении подсчет совершенных переворотов, которые потребуются игроку на поиск всех пар игровых карточек. Текущее количество шагов должно всегда отображаться в верхней части сцены, а после нахождения всех карточек должно отображаться всплывающее окно с итоговым счетом и предложением начать новую игру.

Сохранение прогресса

Задание Реализуйте функцию сохранения прогресса игры. Она должна сохранять текущее состояние игрового процесса (координаты и количе-

ство карточек, текущее количество выполненных переворотов и т.д.) после каждого действия. Продолжить предыдущую игру можно нажатием кнопки «Продолжить игру» на главном экране приложения.

ИТОГИ ЧЕТВЕРТОЙ ЧАСТИ КНИГИ

Трудно сказать, какая из частей книги была наиболее важной, но четвертая однозначно стала наиболее интересной, так как познакомила вас с интерактивными элементами, которые вы можете использовать при разработке собственных приложений. Вы освоили настолько важные базовые навыки, значение которых сложно переоценить, так как профессиональный iOS-разработчик должен знать работу системы и языка «изнутри».

Заклучение

Вот вы и подошли к концу еще одного этапа обучения. Я надеюсь, что смог рассказать вам что-то новое и полезное, и вы стали еще на один шаг ближе к своей цели стать первоклассным iOS-разработчиком. Я желаю вам не останавливать свое обучение и использовать все доступные сегодня учебные материалы: различные книги и статьи (в том числе на иностранных языках), видео- и онлайн-курсы.

Помните, что для собственного профессионального роста очень помогает наличие пет-проекта¹. Если вы все еще не обзавелись им – самое время!

И в заключении, если книга оказалась вам полезно, то я буду рад прочитать ваш отзыв.



Оставить отзыв можно перейдя по следующей ссылке

<https://swiftme.ru/ostavte-svoj-otzyv/>

О вашей поддержке и будущих книгах

Рано или поздно электронный вариант каждой книги начинает активно тиражироваться в сети, в том числе через торрент-трекеры, мессенджеры и т.д. В этом нет ничего плохого, это естественный жизненный цикл совершенно любой книги. Основной плюс в том, что это позволяет привлекать новых членов в сообщество разработчиков на Swift!

Если вы скачали книгу в сети и считаете, что мой труд достоин вашей поддержки, то можете купить ее на сайте swiftme.ru или перевести **произвольную** сумму на один из следующих кошельков:

¹ Пет-проект (от англ. Pet – домашнее животное, любимец, питомец) – это личный (иногда групповой) проект, который вы делаете/разрабатываете с особой старательностью и заботой для собственного удовольствия и развития профессиональных качеств.

**Bitcoin (BTC)**

<https://1Pyo9q5ejxan88p8X3tRrf8vUoFX5EqEcP>

**Ethereum (ETH)**

<https://0x3dD7f5BaDAC2138bBa67F6d1d211C952bb04E294>

**Bitcoin Cash (BCH)**

<https://qr7pp9jgxuxg8xu2cwc5pf9q7ldywvlr9gjq5p480n>

**ЮMoney (Яндекс.Деньги)**

По номеру кошелька 41001436666857
или по ссылке [https:// yoomoney.ru/to/41001436666857](https://yoomoney.ru/to/41001436666857)

**Qiwi**

По никнейму SWIFTMERU
или по ссылке <https://qiwi.com/n/SWIFTMERU>

Эта работа делается не ради денег, но именно ваша финансовая поддержка помогает мне продолжать активно развивать мой стартап и создавать новые учебные материалы и книги.

